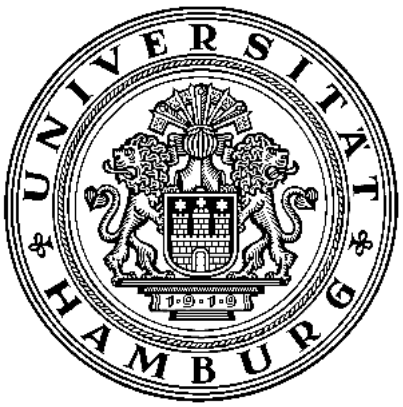


Universität Hamburg
FB Informatik
Datenbanken und
Informationssysteme

DIPLOMARBEIT

# Generische Visualisierung von multimedialen Dokumenten in Online-Redaktionssystemen

vorgelegt am
28. August 2000
von

Christian Pesch
Monikastraße 7
22089 Hamburg

Tel.: 040 / 670 41 79

**Betreuer:**
Prof. Dr. Joachim W. Schmidt
Prof. Dr. Wolfgang Menzel

# Inhaltsverzeichnis


**1 Einleitung und Motivation** **1**

1.1 Aufgabenstellung . . . . . 1

1.2 Struktur der Arbeit . . . . . 3

**2 Kontext der Arbeit** **5**

2.1 Multimediale Dokumentenverarbeitung . . . . . 6

2.1.1 Eigenschaften . . . . . 7

2.1.2 Einsatzgebiete . . . . . 9

2.2 Online-Redaktionssysteme . . . . . 10

2.2.1 Anforderungen . . . . . 10

2.2.2 Redaktionelle Prozesse . . . . . 11

2.2.3 Eigenschaften . . . . . 15

2.2.4 Technologische Basis . . . . . 17

2.2.5 Beispiele für Systeme . . . . . 20

2.3 Gestaltung der Schnittstelle zum Benutzer . . . . . 24

2.3.1 Terminal . . . . . 25

2.3.2 Graphische Benutzungsoberfläche . . . . . 25

2.3.3 HTML-Formulare . . . . . 27

**3 Grundlagen der generischen Visualisierung** **29**

3.1 Dokumentenmodelle . . . . . 29

3.1.1 Elektronische Dokumente . . . . . 30

3.1.2 Bestandteile von Dokumenten . . . . . 30

3.1.3 Beispiele für Dokumentenmodelle . . . . . 32

3.2 Typsysteme . . . . . 38

3.2.1 CoreMedia . . . . . 39

3.2.2 Programmiersprachen . . . . . 39

3.2.3 Datenbanken . . . . . 40

3.3 Verarbeitungsweise . . . . . 41

3.3.1 Ebenen der Visualisierung . . . . . 41

3.3.2 Auswahl der Visualisierungsebene . . . . . 42

3.4 Dokumentzustände . . . . . 43

**4 Software-Architektur und Alternativen** **45**
4.1 Architektur von CoreMedia . . . 46
4.1.1 Überblick . . . 46
4.1.2 Model-View-Controller . . . 49
4.1.3 Abstraktes Verteilungsmodell . . . 51
4.2 Architektur eines Systems zur generischen Visualisierung . . . 54
4.2.1 Verarbeitungsregeln . . . 55
4.2.2 Aggregationshierarchie durch Typen . . . 56
4.2.3 Klassifikation der Alternativen . . . 56
4.2.4 Schema für die Einteilung . . . 59
4.3 Kodierung . . . 59
4.3.1 Direkte Kodierung . . . 60
4.3.2 Entwurf eines Rahmenwerks . . . 60
4.3.3 Nutzung von Standardwerkzeugen . . . 62
4.3.4 Übersetzung und Einbindung von Quelltext zur Laufzeit . . . 63
4.4 Generierung . . . 64
4.4.1 Gegenstand der Generierung . . . 65
4.4.2 Generierung von Quelltext . . . 65
4.4.3 Layout-Generierung . . . 66
4.5 Generierung mit Nachbearbeitung . . . 66
4.5.1 Regelbasierte Nachbearbeitung . . . 67
4.5.2 Interaktive Nachbearbeitung . . . 67
4.6 Eingrenzung . . . 68

**5 Systemimplementierung** **71**
5.1 Model-View-Controller-Implementierungen . . . 72
5.2 Implementierung des Rahmenwerks . . . 73
5.2.1 Abstraktionsstufen . . . 73
5.2.2 Strukturierung des Rahmenwerks . . . 74
5.2.3 Zentrale Komponenten . . . 76
5.2.4 Konfiguration der Sichten auf Dokumente . . . 80
5.3 Implementierung der Sichten . . . 81
5.3.1 Sichten für den Dokumentenspeicher . . . 81
5.3.2 Sichten auf einzelne Dokumente . . . 84
5.3.3 Sichten auf Attribute . . . 87
5.4 Generierung von Dokumentsichten . . . 89
5.4.1 Layout-Generator . . . 89
5.4.2 Regelbasierte Nachbearbeitung . . . 91
5.4.3 Interaktive Nachbearbeitung . . . 93

**6 Zusammenfassung und Ausblick** **95**
6.1 Zusammenfassung . . . 95
6.2 Bewertung der Ergebnisse . . . 96
6.3 Bewertung der eingesetzten Werkzeuge . . . 98
6.4 Ausblick . . . 99

**A Dokumenttypen** **102**
A.1 XML-DTD . . . . . 102
A.2 XML-Dokument . . . . . 103

**B Layout-Definitionen** **104**
B.1 XML-DTD . . . . . 104
B.2 XML-Dokument . . . . . 105

**Literaturverzeichnis** **106**

# Abbildungsverzeichnis


2.1 Aktoren und Anwendungsfälle eines redaktionellen Prozesses . . . . . 13
2.2 Aktivitäten eines redaktionellen Prozesses . . . . . 14

3.1 Dokumente und ihre Bestandteile . . . . . 31
3.2 Beziehungen zwischen SGML, XML und HTML . . . . . 34
3.3 Beispiel für ein HTML-Dokument . . . . . 35
3.4 Baumstruktur des HTML-Dokuments . . . . . 35
3.5 Beispiel für ein XML-Dokument . . . . . 36
3.6 Baumstruktur des XML-Dokuments . . . . . 37
3.7 Objektdiagramm eines CoreMedia-Dokuments . . . . . 38
3.8 Speicherungsstrukturen eines CoreMedia-Dokuments . . . . . 42
3.9 Zustände von Dokumenten . . . . . 44

4.1 Systemarchitektur . . . . . 47
4.2 Model-View-Controller . . . . . 49
4.3 Systemgrenzen zur Verteilung bei Model-View-Controller . . . . . 50
4.4 Klassendiagramm zur Verteilung von Objekten . . . . . 51
4.5 Klassendiagramm zur Aktualisierung der verteilten Objekte . . . . . 52
4.6 Interaktionsdiagramm zur Initialisierung der verteilten Zustände . . . . . 53
4.7 Interaktionsdiagramm zum Abgleich der verteilten Zustände . . . . . 54
4.8 Beziehung von Dokumenttyp und Dokument . . . . . 55
4.9 Beziehung von Attributtyp und Dokumenttyp . . . . . 55
4.10 Aggregationshierarchie durch Typen . . . . . 57
4.11 Autor-Kritiker-Zyklus . . . . . 61
4.12 Anpassung der generischen Visualisierung . . . . . 69

5.1 Drei Abstraktionsstufen . . . . . 74
5.2 Hierarchie der Pakete . . . . . 75
5.3 Hierarchie der Editor-Klassen . . . . . 76
5.4 Die Klasse AbstractClient und ihre Subklassen . . . . . 77
5.5 Hierarchie der Klassen für Dokumentsichten . . . . . 79
5.6 Die Klasse AbstractView . . . . . 80
5.7 Die Klasse AbstractDocumentTableView . . . . . 82
5.8 Die Klasse LayoutTableDefinition . . . . . 83
5.9 Die Klasse BasicDocumentView und ihre Super- und Subklassen . . . . . 86
5.10 Hierarchie der Attributsichten . . . . . 88
5.11 Bildschirmfoto zur graphischen Struktur (Poem) . . . . . 92
5.12 Bildschirmfoto zur graphischen Struktur (Author) . . . . . 92

# Tabellenverzeichnis

2.1 Vergleich der Online-Redaktionssysteme . . . . . 23

4.1 Einordung der Entwürfe . . . . . 59

5.1 Abbildung von Attributsichten auf Serverobjekte . . . . . 90
5.2 Parameter einer Layout-Definition . . . . . 91

# Kapitel 1

# Einleitung und Motivation

Die zunehmende Digitalisierung der Informationsmedien bedeutet eine Herausforderung für die Produzenten der Medieninhalte. Die Verarbeitung multimedialer Dokumente, die wie Hypertexte miteinander verwoben sind, erfordert die Unterstützung durch neue Software-Systeme. Mit Systemen zur multimedialen Dokumentenverarbeitung wird versucht, die Produktion umfangreicher, multimedial aufbereiteter Informationsangebote zu realisieren. Dabei gibt es in den traditionellen Medien wie Film, Funk und Fernsehen bereits Vorgehensweisen und Methoden, die sich für die Produktion qualitativ hochwertiger Inhalte eignen. Die relativ junge Klasse der Online-Redaktionssysteme überträgt die redaktionellen Prozesse der tradionellen Medien auf die digitale Welt und integriert die Leistung eines Systems zur multimedialen Dokumentenverarbeitung. Durch die multimedialen Inhalte, die Unterstützung redaktioneller Prozesse, aufwendige Arbeitsabläufe, viele gleichzeitige Benutzer, große Dokumenten- und Datenmengen und eine redaktionell vorgeprägte Benutzergruppe gehören Online-Redaktionssysteme zu einer komplexen Problemklasse. Deshalb muß die Benutzungsoberfläche für ein solches System besonderen Anforderungen genügen, um diese Komplexität für den Benutzer handhabbar zu machen. Die Implementierung eines graphischen Bearbeitungswerkzeuges zur Visualisierung und Bearbeitung multimedialer Dokumente hat sich der einem solchen System inhärenten Komplexität zu stellen. Ein generischer Ansatz kann dabei helfen, handhabbare Komponenten zu erstellen und diese Komponenten an die gestellten Anforderungen anzupassen. Es wird untersucht, inwieweit die Konfiguration einer generischen Lösung im Vergleich zu einer Neuimplementierung bestehen kann, wenn man Aufwand, Flexibilität, Mächtigkeit und Bedienbarkeit betrachtet.

## 1.1 Aufgabenstellung

Während der Entwicklung großer Software-Systeme tritt häufig die Problemstellung auf, mit den aufwendigen Anpassungen für die Nutzer des Systems umzugehen. Die dafür notwendigen Änderungen oder Anpassungen betreffen alle Komponenten des Systems. Eine Lösung für dieser Problemstellung besteht darin, einen generischen Kern des Systems zu entwerfen, der gemäß den Anforderungen der Nutzer konfiguriert wird. Bei den Komponenten, die eine graphische Benutzungsoberfläche realisieren, ist dieser Lösungsweg jedoch im Vergleich zu den anderen Komponenten sehr aufwendig. Ohne eine generische Lösung für die graphische Benutzungsoberfläche ist für jeden Nutzer des Systems eine vollständig neue Implementierung notwendig.

In einer Reihe von Projekten mit Nutzern aus verschiedenen Einsatzgebieten sind umfangreiche Erfahrungen gesammelt worden. Sie zeigen, daß die Erstellung und Änderung der Komponenten für die graphische Benutzungsoberfläche

- aufwendig,
- statisch, d.h. schlecht zu ändern und
- fehlerträchtig

ist. Es dauert in Regel eine Personenwoche, bis eine Anpassung der graphischen Benutzungsoberfläche nach den Vorgaben der späteren Nutzer implementiert ist. Danach erfordern Änderungen wieder einen Zyklus mit Spezifikation, Implementierung, Test und Installation. Diese Schritte sind zeitintensiv und umständlich zu organisieren. Der Zeitraum, bis eine Änderung für den Nutzer wirksam wird, kann relativ lang werden. Unter den derzeitigen Rahmenbedingungen, unter denen die Zeit bis zum Markteintritt (*Time to market*) zu einem entscheidenden Faktor geworden ist, bedeutet dies einen großen Nachteil.

Mittels geeigneter Software-Techniken kann der gemeinsame Teil aller durch Anpassungen entstehenden Varianten deutlich vergrößert werden. Es zeigt sich, daß die für ein Software-System charakteristischen häufigen Anpassungen trotz solcher Techniken zu einem erheblichen Aufwand führen. Der typische Arbeitsablauf beim Kodieren der Anpassungen mit den Schritten Bearbeiten, Übersetzen, Starten, Beurteilen und Verbessern ist sehr langwierig. Diesen Arbeitsablauf zu verkürzen, ist ein wichtiger Teil der Motivation für diese Arbeit.

Der Kontext der Arbeit stellt eine deutliche Einschränkung dar. Ein Vergleich zeigt, daß es im Bereich der Online-Redaktionssysteme keine Systeme gibt, die eine generische Lösung für eine graphische Benutzungsoberfläche besitzen. Die meisten Systeme verarbeiten ausschließlich unstrukturierte Daten. Für eine generische Visualisierung sind jedoch Strukturinformationen über die Daten erforderlich, um eine ansprechende Darstellung und eine geeignete Bearbeitung der Daten zu ermöglichen. Die Realisierung einer generischen Lösung ist nur mit solchen Online-Redaktionssystemen möglich, die z.B. über eine entsprechende Typisierung solche Strukturinformationen zur Verfügung stellen. Der Vergleich von Online-Redaktionssystemen ergibt, daß die technologische Basis des CoreMedia [Higher-Order Informations- und Kommunikationssysteme GmbH 99] Online-Redaktionssystems am besten geeignet ist, um eine generische Visualisierung multimedialer Dokumente zu implementieren. Innerhalb dieses Kontextes werden eine Reihe von breit gestreuten Lösungsansätzen entworfen, die einen möglichst großen Raum von Lösungen überdecken. Ein vielversprechender Weg wird ausgewählt und die Gründe für die Entscheidung dargestellt. Der gewählte Lösungsweg wird implementiert und die Erfahrungen, die dabei gesammelt wurden, werden beschrieben.

Am Ende dieser Arbeit steht eine wartungsfreundliche, generische Lösung für eine graphische Benutzungsoberfläche. Dies bedeutet, daß nach der Anpassung des CoreMedia Online-Redaktionssystems der Zyklus mit den Schritten Implementierung, Test und Installation entfällt. Sofort nach der Anpassung des Systems kann mit der graphischen Benutzungsoberfläche gearbeitet werden. Unter Umständen ist eine Konfiguration erforderlich, um die Anforderungen zu erfüllen. Dieser Schritt darf in seinen Möglichkeiten nicht zu beschränkt sein oder einen unangemessenen Aufwand erfordern, um im Vergleich zur neu kodierten Variante zu bestehen. Der große Aufwand für die Implementierung einer generischen Lösung soll mehr als aufgewogen werden durch den geringeren Aufwand bei der Konfiguration der generischen Version im Vergleich zu einer neu implementierten Variante.

Im Bereich der Dokumentenmanagement-Systeme gibt es einige generische Lösungen. Die Anforderungen an Dokumentenmanagement-Systeme sind jedoch andere als die an Online-Redaktionssysteme. Ein Lösungsansatz könnte darin bestehen, ein Online-Redaktionssystem auf Basis eines Dokumentenmanagement-Systems zu erstellen und so dessen

graphische Benutzungsoberfläche zu nutzen. Der Aufwand hierfür ist aufgrund der unterschiedlichen Anforderungsprofile deutlich höher als die Implementierung einer generischen Lösung auf Basis eines Online-Redaktionssystems. Der Markt für Online-Redaktionssysteme selbst ist noch recht jung, so daß noch kein Standardprodukt vorliegt, das als Basis für die generische Visualisierung dienen könnte. Eine Implementierung einer generischen Lösung basiert also immer auf einem proprietären System.

In der Forschung gibt es Projekte zur Generierung von Benutzungsoberflächen. Beim BOSS-Projekt [Schreiber 94] der TU München wird versucht, aus einer abstrakten Beschreibung der Daten und ihrer Eigenschaften eine graphische Benutzungsoberfläche zu erzeugen. Anfang der 80er Jahre hat sich ein Teil der Forschung im Bereich der Büroautomatisierung mit der Konfiguration von Prozessen, die Dokumente in eine graphische Repräsentation überführen, beschäftigt. Die Systeme MINOS und MULTOS [Tsichritzis 85] sind hier als Beispiel zu nennen. Die Architekten dieser Systeme arbeiteten später im Bereich von Hypertext und Hypermedia [Lucarella et al. 92; Corders, Streitz 92] und erstellten Werkzeuge, um durch Hypertext-Dokumente zu navigieren. Die Aufgabenstellung, eine graphische Repräsentation der Knoten des Hypertext-Dokuments zu generieren, ist mit dieser Arbeit vergleichbar.

Eine ähnliche Problemstellung findet man bei den Hypertext-Dokumenten im World Wide Web. Die für die Navigation eingesetzten Web-Browser müssen logische Strukturen von Dokumenten in graphische umsetzen. In jüngster Zeit wurden solche Layout-Techniken im Rahmen der *eXtensible Stylesheet Language* (XSL) [DuCharme 99] entworfen. Hier wird eine Entwicklung fortgeführt, die mit der *Document Style Semantics and Specification Language* (DSSSL) [ISO10179 96] begonnen hat. Auch kommerzielle Software wie FrameMaker+SGML [Gulbins, Obermayr 99] realisiert eine Umsetzung logischer in graphische Strukturen, in diesem Fall allerdings mit einer proprietären Technik. Der Forschungsprototyp *Grif* [Quint, Vatton 92] bietet in diesem Bereich einen Funktionsumfang, der deutlich über den kommerzieller Software hinausgeht. Im Rahmen des Projekts *Individualised Electronic Newspaper* [Hülser, Weber 92] ist ein Prototyp entwickelt worden, der viele Merkmale von Online-Redaktionssystem besitzt. Da für jeden Leser eine individuelle Zeitung erstellt werden soll, muß die graphische Struktur für jeden Leser einzeln aus einem redaktionellen Inhalt und einer Layout-Beschreibung generiert werden.

## 1.2 Struktur der Arbeit

Diese Arbeit befaßt sich mit der generischen Visualisierung multimedialer Dokumente in einem Online-Redaktionssystem. Kapitel 2 beschreibt den Kontext der Arbeit mit einem Überblick über den Bereich der multimedialen Dokumentenverarbeitung, die Teilmenge der Online-Redaktionssysteme und die redaktionellen Prozesse, die diese System unterstützen. Es werden die Eigenschaften dieser Systeme, die Leistungen, die sie erbringen, und ihre Einsatzgebiete betrachtet. Die Gestaltung der Schnittstelle zum Benutzer bei solchen Systemen wird diskutiert. Als Basis für die folgenden Kapitel werden in Kapitel 3 die Grundlagen der generischen Visualisierung, angefangen bei Dokumentenmodellen und Typsystemen, die versuchen, Dokumente zu definieren, über verschiedene Visualisierungsebenen, die Trennung von logischen und graphischen Strukturen, bis zur Verarbeitung der Dokumente für die Visualisierung beschrieben. In Kapitel 4 werden Software-Architekturen zur generischen Visualisierung entworfen, Alternativen zu ihnen erstellt und die Entscheidung für einen Entwurf vertreten. Die Erfahrungen, Erkenntnisse und Schwierigkeiten bei der Implementierung werden in Kapitel 5 vorgestellt. Die Zusammenfassung und der Ausblick in Kapitel 6 bilden den Abschluß dieser Arbeit.

# Kapitel 2

# Kontext der Arbeit

Eine Möglichkeit, ein elektronisches Dokument zu erstellen, besteht darin, ein Papierdokument in eine für Rechner handhabbare, strukturierte Form zu übertragen. Dies erlaubt gegenüber der Arbeit mit Papierdokumenten die kostengünstige und schnelle Vervielfältigung und Verarbeitung der Dokumente. Unter Verarbeitung ist hier z.B. die Bearbeitung des Inhalts der Dokumente durch menschliche Bearbeiter oder Programme, der Austausch der Dokumente mit Dritten oder die Recherche in einer Menge von Dokumenten, d.h. ganz allgemein die Abwicklung eines Bearbeitungsprozesses auf den Dokumenten zu verstehen. Für diese Arbeit interessieren die näheren Umstände für die Verarbeitung der Dokumente nicht. Entscheidend ist, daß die technischen Voraussetzungen für eine Verarbeitung vorhanden sind.

Liegt ein Dokument in einer für Rechner handhabbaren Form vor, liegt es nahe, die Dokumente in einer Datenbank zu speichern. Dadurch können die Vorteile genutzt werden, die ein Datenbankmanagementsystem bietet. Die wichtigsten dieser Vorteile sind:

- die Dauerhaftigkeit von Änderungen,
- die Erholung nach Fehlern,
- die zentralisierte Sicherung der Daten,
- die Möglichkeit zur parallelen Bearbeitung der Daten und
- eine kalkül- oder algebrabasierte Anfragesprache.

Ein Dokumentenverarbeitungssystem baut auf einem Datenbankmanagementsystem auf und bietet darüber hinaus Leistungen für die Verarbeitung der Dokumente an. Dazu gehören beispielsweise Recherchemöglichkeiten, Funktionen zum Import und Export von Dokumenten sowie Mechanismen zur Kooperation der Benutzer für die gemeinsame Bearbeitung der Dokumente.

Für den Multimedia-Einsatz müssen die Dokumente um die Verwendung von Farbe, die Integration von Grafiken, Audio- und Video-Sequenzen sowie die Einbettung und Verknüpfung anderer Dokumente erweitert werden. Diese Entwicklung wird gefördert durch die wachsende Verbreitung und die ständige Verbesserung der Leistungsfähigkeit der Informationstechnik. Damit lassen sich Dokumente mit immer aufwendigerer und qualitativ hochwertigerer Gestaltung erstellen und verarbeiten. Parallel dazu sorgt das rasante Wachstum des World Wide Web für einen breiten Einsatz von Hypertext- und Hypermedia-Dokumenten. Als Hypertext bezeichnet man eine Menge von Dokumenten,

die strukturiert werden durch Verweise. Die Verweise schaffen einen hohen Grad an Vernetzung, der das nichtlineare Lesen, das Navigieren, in der Menge von Dokumenten fördert. Hypermedia ist eine Erweiterung von Hypertext, die multimediale Elemente in die Hypertext-Dokumente integriert. Die beiden Faktoren Multimedia und Hypertext haben zu einer weiten Verbreitung von multimedialen Dokumenten und zu steigenden Anforderungen bei der Produktion und Verarbeitung dieser Dokumente geführt.

Online-Redaktionssysteme erlauben auf Basis von Dokumentenverarbeitungssystemen die Abbildung redaktioneller Prozesse. Solche Prozesse sind nötig, um aufwendige, multimediale Dokumente mit inhaltlichem und gestalterischem Qualitätsanspruch herzustellen. Online-Redaktionssysteme bieten damit Dienste für redaktionelle Prozesse, die beispielsweise in Bibliotheken bei der Buchbeschaffung, bei der Erstellung von wissenschaftlichen Arbeiten, im Pressewesen und in Standardisierungsgremien genutzt werden. Die redaktionellen Prozesse in den genannten Anwendungsfeldern erlauben es, eine Qualitätskontrolle für die produzierten Inhalte zu realisieren. Online-Redaktionssysteme bieten diese Prozesse für digitale Medien z.B. zur Erstellung von Internet Websites an.

Mit einer intuitiv bedienbaren Benutzungsschnittstelle für ein solches System vergrößert sich der Kreis der Benutzer um diejenigen Benutzer, die wenig Erfahrung und Übung im Umgang mit dem System haben. Für die Darstellung und Bearbeitung von multimedialen Inhalten sind Schnittstellen zum Benutzer nötig, mit denen die Inhalte effizient bearbeitet werden können. An die Benutzungsschnittstelle eines Online-Redaktionssystems werden diese beiden Anforderungen gleichzeitig gestellt. Zum einen arbeitet mit Redakteuren eine Benutzergruppe mit dem System, die ihre Hauptkompetenz nicht im Bereich der Informationstechnik liegt. Zum anderen soll das System diesen Benutzern die komfortable Erstellung und Verarbeitung multimedialer Inhalte ermöglichen.

Dieses Kapitel stellt die Problemklasse der redaktionellen Prozesse bei der Produktion multimedialer Dokumente vor. Es beschreibt zunächst die multimediale Dokumentenverarbeitung im allgemeinen und wendet sich dann der Teilmenge der Online-Redaktionssysteme zu. Dabei werden die Anforderungen an Online-Redaktionssysteme, die Eigenschaften, die sich aus den Anforderungen ergeben sowie die technologische Basis, mit deren Hilfe die Anforderungen realisiert werden, erläutert.

## 2.1 Multimediale Dokumentenverarbeitung

Der Begriff der multimedialen Dokumentenverarbeitung ist aufgrund der weitverbreiteten und werbewirksamen Nutzung des Wortes Multimedia und der sehr vielfältigen Deutungen für das Wort Dokument recht unklar. In diesem Abschnitt werden die Begriffe

- Multimedia und
- Dokument

diskutiert. Es wird beschrieben, was rechnergestützte, multimediale Dokumentenverarbeitung leisten kann. Dazu wird erläutert, wie, in welchen Bereichen und für welche Szenarien Systeme zur multimedialen Dokumentenverarbeitung eingesetzt werden.

Das Wort *Multimedia* ist 1995 das Wort des Jahres geworden. Es ist seit Jahren in aller Munde und prangt heutzutage auf einer großen Anzahl von Produkten. Die 19. Ausgabe des großen Brockhaus [Brockhaus 86] definiert Multimedia wie folgt:

> „aufeinander abgestimmte Verwendung versch. (vieler) Medien, Medienverbund insbes. in der Unterhaltungsbranche (M.-Show) [..] In der Publizistik meint M. Unternehmen oder Unternehmensgruppen, die mit mehreren auch unterschiedl. Medienbetrieben, z.B. mit Buch- und Presseverlagen, Film-, Video- und Rundfunkgesellschaften, tätig sind."

Dennoch mangelt es an einem einheitlichen Verständnis, was Multimedia ist, und deshalb sind die Interpretationen vielfältig. Unter Multimedia ist im Zusammenhang mit der multimedialen Dokumentenverwaltung ganz allgemein die Möglichkeit zu verstehen, mehr als ein Medium verarbeiten zu können. Bei Rechnern ist das immer vorhandene Medium der Text. Es entspricht der Praxis, daß häufig bereits von Multimedia gesprochen wird, wenn die Möglichkeit besteht, über die Verarbeitung von Texten hinaus Grafiken, zwei- oder dreidimensionale Bilder, Videos, Klänge, Sprache oder beliebige Datenstrukturen [Meyer-Wegener 91] zu verarbeiten. In letzter Zeit taucht der Begriff Multimedia verstärkt in Verbindung mit Begriffen der neuen Medien auf. Dabei wird unter Multimedia die digitale Präsentation von Grafik, Tönen und bewegten Bildern verstanden [Blittkowski 97].

Ebenso wie beim Begriff Multimedia gibt es eine breit gestreute Anzahl von Definitionen für den Begriff *Dokument*. Die folgenden, recht allgemein gefaßten Definitionen reichen für die Beschreibung der multimedialen Dokumentenverarbeitung aus. Eine ausführliche Diskussion des Begriffs Dokument, der Online-Redaktionssystemen zu Grunde liegt, findet sich im Abschnitt 3.1. In der 19. Ausgabe des großen Brockhaus [Brockhaus 86] ist ein Dokument definiert als

> „mlat. beweisende Urkunde, eigtl. das zur Belehrung über etwas oder zur Erhellung von etwas Dienliche, zu lat. docere lehren; Urkunde, amtl. Schriftstück; Beweismittel, Beleg".

Der Brockhaus beschreibt die althergebrachte Bedeutung des Wortes Dokument, die die physische Natur eines Dokumentes betont. Wilhelm und Heckmann [Wilhelm, Heckmann 96] beziehen den virtuellen Aspekt elektronischer Dokumente mit ein und erweitern den Begriff des Dokuments auf weitere Medien. Sie bezeichnen Dokumente als

> „Bündel von Informationen, die in der Absicht erzeugt wurden, daß sie von Menschen wahrgenommen werden. Beispiele dafür sind Reden, Filme, Bilder und Schriftdokumente."

Die DIN-Norm [DIN31631 84] dagegen formuliert technischer und betont dabei den materiellen, d.h. physischen Charakter eines Dokuments:

> „Ein Dokument ist die materielle Einheit eines Trägers bibliographischer und dokumentarischer Daten."

Für die Charakterisierung der multimedialen Dokumentenverarbeitung reicht die Einordnung aus, daß es sich um virtuelle Dokumente handelt, die per Rechner verarbeitet werden. Das wesentliche Merkmal dieser Dokumente ist, daß sie aufgezeichnete Information darstellen, die durch entsprechende Techniken wieder physisch repräsentiert werden kann.

### 2.1.1 Eigenschaften

Ausgehend von diesen Definitionen der Begriffe Multimedia und Dokument lassen sich die Eigenschaften eines Systems zur rechnergestützten, multimedialen Dokumentenverarbeitung charakterisieren. Es bietet:

- die persistente Speicherung multimedialer Dokumente,
- die Verwaltung einer großen Anzahl von Dokumenten,
- die Verarbeitung der Dokumente,

- Recherchemöglichkeiten,
- den Austausch von Dokumenten,
- den parallelen Betrieb mit mehreren Benutzern,
- die Unterstützung der Kooperation der Benutzer und
- eine graphische Benutzungsoberfläche.

Unter *persistenter Speicherung der Dokumente* ist die Ablage der Dokumente in einer Datenbank zu verstehen. Die gespeicherten Dokumente können Daten aus einer Reihe unterschiedlicher Medien wie z.B. Texte, Bilder und Klänge aufnehmen. Es ist möglich, mit einer *großen Anzahl von Dokumenten* zu arbeiten. Mit „groß“ sind hier Größenordnungen von mehr als $10^5$ Dokumenten bezeichnet.

Mit der *Verarbeitung der Dokumente* ist die technische Möglichkeit gemeint, an den Inhalt der Dokumente heranzukommen und ihn zu manipulieren. Dies kann durch menschliche Bearbeiter oder durch Programme geschehen, die innerhalb eines Verarbeitungsprozesses mit den Dokumenten arbeiten. Der Grund, weshalb ein Dokument verarbeitet wird, liegt außerhalb des Systems und wird in dieser Arbeit ausgeklammert. Entscheidend ist, daß die technischen Voraussetzungen für die Verarbeitung vorhanden sind.

Die *Recherchemöglichkeiten* haben das Ziel, aus einer großen Anzahl von Dokumenten die für den Benutzer nützlichen Dokumente herauszufiltern, ohne dabei ein relevantes Dokument zu übersehen. Aufgrund des umfangreichen Dokumentenbestandes kann der Benutzer nicht jedes Dokument im System kennen, so daß dieser Funktion eine entscheidende Bedeutung zukommt. Beim *Information Retrieval* [Steffens 97] gibt der *Rücklauf* (*Recall*) die Fähigkeit eines Verfahrens an, nützliche Dokumente zu finden. Mit *Genauigkeit* (*Precision*) wird bezeichnet, wie konsequent „nutzlose“ Dokumente ausgeschlossen werden. Im Idealfall gibt es Verfahren mit großem Rücklauf und großer Genauigkeit für alle unterschiedlichen Medien. Zufriedenstellend funktioniert dies meist nur für Texte. Die Suche in visuellen Medien ist unter den Namen *Image-Retrieval* [Aksoy, Haralick 98; Srihari, Zhang 98] und *Video-Retrieval* [Choi et al. 98] noch Gegenstand der Forschung und bisher selten anzutreffen. Bei der Suche in Audio-Daten befaßt man sich noch mit Grundlagenforschungen und es gibt nur Lösungsansätze [Jajodia et al. 98].

Ein Dokumentenverarbeitungssystem ist offen für den *Austausch von Dokumenten*. Es bietet die Möglichkeit, bestehende Dokumentenmengen aus alten Systemen weiterzuverwenden. Dies ist die Voraussetzung dafür, daß alte von neuen Systemen abgelöst werden können. Darüber hinaus ist es in der Lage, mit anderen Systemen zusammenzuarbeiten. Der Austausch von Dokumenten wird häufig über den Im- und Export oder über Programmierschnittstellen realisiert. Daneben gibt es noch weitere Möglichkeiten für den Austausch von Dokumenten, doch diese beiden sind die gebräuchlichsten.

Beim *Im- und Export* gelangen die Dokumente durch einen Importprozeß ins System, der externe Daten einliest, diese bei Bedarf ins interne Format des Systems transformiert und speichert. Danach können die importierten Dokumente im System verarbeitet und an andere Systeme exportiert werden. Somit besteht die Möglichkeit, auf der Basis bekannter Dokumentstrukturen die Dokumente extern weiterzuverarbeiten. Dies ist mit genormten Dokumentformaten wie z.B. der *Office Document Architectur* (ODA) [ISO8613 89], auf Basis von Dokumenttypdefinitionen in der *Standardized General Markup Language* (SGML) [ISO8879 86] oder in der *eXtensible Markup Language* (XML) [DuCharme 99] möglich. Diese Dokumentformate werden in Abschnitt 3.1 detailliert behandelt.

Bei der Nutzung von *Programmierschnittstellen* für den Austausch von Dokumenten rufen externe Programme Funktionen des Systems auf, mit denen Dokumente im System gespeichert oder aus dem System heraus abgerufen und verändert werden können. Dieser

Ansatz hat den Vorteil, daß durch eine umfangreiche Programmierschnittstelle die Möglichkeit besteht, die Funktionalität des Systems zu erweitern. Die Offenheit des Systems durch Programmierschnittstellen vermeidet die Nachteile proprietärer Insellösungen wie mangelnde Erweiterbarkeit, schlechte Skalierbarkeit und Bindung an einen Hersteller.

Die im System gespeicherten Dokumente können im *parallelen Betrieb mit mehreren Benutzern* [Gibbs, Verrijn-Stuart 90] verarbeitet werden. Die Verteilung der Arbeit auf mehrere Benutzer ist die Voraussetzung dafür, um mit einer großen Dokumentenmenge umgehen zu können. Die *Kooperation der Benutzer* wird durch Rollen, Zuständigkeiten, Arbeitsabläufe (*Workflow*) und Bearbeitungszustände unterstützt. Wie dies für redaktionelle Prozesse genutzt wird, ist in Abschnitt 2.2.2 beschrieben.

Es wird eine *graphische Benutzungsoberfläche* [Voss, Nentwig 98] mit der weitverbreiteten *Windows, Icons, Menus, Pointer* (WIMP) Metapher [Wessel 98] eingesetzt, um Benutzern mit geringen Vorkenntnissen die Arbeit mit dem System zu ermöglichen. Die Eigenschaften dieser Schnittstelle des Systems werden in Abschnitt 2.3 näher erläutert.

Wie sich zeigt, unterscheiden sich die Leistungen einer rechnergestützten, multimedialen Dokumentenverarbeitung nicht weit von den Leistungen, die Systeme zur Verarbeitung von elektronischen Dokumenten erbringen. Der wesentliche Unterschied besteht in der Unterstützung weiterer Medien, der Offenheit zu anderen Systemen sowie der Notwendigkeit, graphische Benutzungsoberflächen zu verwenden.

### 2.1.2 Einsatzgebiete

Die Einsatzgebiete für Systeme zur multimedialen Dokumentenverarbeitung sind vielfältig, da der Begriff des Dokumentes als aufgezeichnete Information sehr allgemein ist. Eine wesentliche Eigenschaft der Informationstechnologie ist die Speicherung und Verarbeitung von Daten, so daß sich mit einem Dokument sehr viele Gegenstände der realen Welt abbilden lassen. Beispielhaft sind einige Bereiche aufgezählt, in denen Systeme zur multimedialen Dokumentenverarbeitung eingesetzt werden. Diese sind:

- Redaktion über Inhalte,
- Vorfall- oder Sachbearbeitung,
- Wissensverwaltung und
- Archivierung.

Die *Redaktion über Inhalte* findet vor allem Verwendung im Medien- und Nachrichtenbereich, wo aktuelle Inhalte möglichst schnell verteilt und veröffentlicht werden sollen. Zusätzlich bieten Archive die Möglichkeit, auf weniger aktuelle und veraltete Inhalte zuzugreifen. Dieser Bereich wird in Abschnitt 2.2 noch weiter vertieft.

Die *Vorfall- oder Sachbearbeitung* ist ein klassischer Prozeß, den die Büroautomatisierung [Tsichritzis 85] schon Mitte der achtziger Jahre erforscht hat. Er besteht zu einem großen Teil aus immer wiederkehrenden Vorgängen, die einfach zu automatisieren sind. In der Folgezeit wurde versucht, immer mehr von den nicht klar strukturierbaren Arbeitsabläufen zu automatisieren [Khoshafian et al. 92]. Systeme, die diese Arbeitsabläufe unterstützen, werden als *Workflow Management*-Systeme bezeichnet. Sie existieren in unterschiedlichen Varianten bei allen größeren Organisationen wie z.B. Versicherungen, Banken, Call-Centern, öffentlichen Verwaltungen oder Produktionsbetrieben.

Unter dem Titel *Wissensverwaltung* (*Knowledge Management*) wird alles zusammengefaßt, was über das bloße Speichern und Aufsuchen von Daten hinausgeht. Der zentrale Gedanke besteht darin, die im Geschäftsbetrieb einer größeren Organisation anfallenden

Dokumente als organisationsweite Wissensbasis zu nutzen. Es existiert ein Interesse an Metainformationen wie z.B. der Entstehungsgeschichte von Dokumenten oder der Vernetzung von Informationen durch die Verknüpfung von Dokumenten. Mit der Erstellung und Verwaltung von Metainformationen wird versucht, die Qualität von Anfrageergebnissen zu verbessern, um besseren Nutzen aus der Wissensbasis zu ziehen.

Bei der *Archivierung* von Dokumenten muß aufwendige und/oder umfangreiche Dokumentation aus rechtlichen oder wirtschaftlichen Gründen lange Zeit aufbewahrt werden. Die archivierten Daten können z.B. Register, Grundbücher und Patente in Behörden, Rechnungen in Buchhaltungen, Buchungsdaten aus Warenwirtschaftssystemen, Prima Noten in Banken oder Dokumentation bei Flug- oder Fahrzeugherstellern sein.

In den Einsatzgebieten, in denen Systeme für rechnergestützte, multimediale Dokumentenverarbeitung Verwendung finden, kommen ihre Vorteile nicht immer zur Geltung. Problematisch sind vor allem die Defizite bei den Recherchemöglichkeiten in einigen Medien, die in Abschnitt 2.1.1 beschrieben wurden. Trotz dieser Einschränkungen stellen die genannten Einsatzgebiete einen potentiellen Anwendungsbereich dar, sofern geeignete Techniken zur Recherche in absehbarer Zeit zur Verfügung stehen. Bis dahin können Metainformationen für alle anderen Medien außer dem Textmedium hinterlegt werden. Diese Metainformationen müssen zeitaufwendig manuell oder halbautomatisch erfaßt werden. Die Qualität der Suche hängt entscheidend von der Qualität und der Quantität der Metainformationen ab.

## 2.2 Online-Redaktionssysteme

Online-Redaktionssysteme stellen redaktionelle Prozesse zur Erstellung von Publikationen in den neuen Medien bereit. Beispiele sind die Produktion von Internet Websites, die Beschickung mobiler Empfänger, die Weiterverarbeitung in nachgelagerten Systemen etc. In diesem Umfeld ist die Aktualität der Informationen von herausragender Bedeutung und die redaktionellen Prozesse sollen trotz des Zeitdrucks bei der Produktion eine Qualitätskontrolle für die produzierten Inhalte ermöglichen. Unter diesen Aspekt ist auch die Möglichkeit zu sehen, die produzierten Inhalte in mehreren, unterschiedlichen Medien weiterzuverwenden.

Dieser Abschnitt stellt zunächst die Anforderungen an Online-Redaktionssysteme vor und erläutert die besondere Rolle der redaktionellen Prozesse. Anschließend werden die Eigenschaften von Online-Redaktionssystemen und Kriterien für die Bewertung ihrer technologischen Basis beschrieben. Damit können einige Beispiele für Online-Redaktionssysteme verglichen werden, um das System auszuwählen, das als Basis für die Implementierung einer generischen Visualisierung multimedialer Dokumente am besten geeignet ist.

### 2.2.1 Anforderungen

Mit der Tauglichkeit für den Einsatz in den neuen Medien und den starke Aktualitätsbezug der produzierten Inhalte wurden bereits in der Einleitung dieses Abschnitts zwei wesentliche Eckpunkte für die Anforderungen an Online-Redaktionssysteme erwähnt. Zu den Anforderungen an ein Online-Redaktionssystem zählen:

- die Erstellung von Publikationen in den neuen Medien,
- eine schnelle Produktion hochaktueller Inhalte,
- eine Qualitätskontrolle für die produzierten Inhalte,
- die Durchführung redaktioneller Prozesse und
- die Bedienung mehrerer, unterschiedlicher Medien.

Für die *Erstellung von Publikationen in den neuen Medien* müssen multimediale Dokumente erstellt, verwaltet und verteilt werden können. Die Inhalte der Publikationen haben einen starken Aktualitätsbezug, der durch die technischen Eigenschaften der neuen Medien bedingt ist und ein hohes Tempo bei der Produktion der Publikationen erfordert. Ein Online-Redaktionssystem muß dazu die *schnelle Produktion hochaktueller Inhalte* ermöglichen und unterstützen.

Gleichzeitig muß eine *Qualitätskontrolle der produzierten Inhalte* helfen, die Qualität der Publikationen trotz des hohen Produktionstempos aufrecht zu erhalten. Dazu bietet sich der - auch außerhalb der Online-Redaktionssysteme erprobte - *Einsatz redaktioneller Prozesse* an. Die redaktionellen Prozesse werden ausführlich in Abschnitt 2.2.2 vorgestellt.

Schließlich sollte ein Online-Redaktionssystem *mehrere, unterschiedliche Medien* bedienen können. Damit können die produzierten Inhalte in mehreren Medien und auf unterschiedlichen Vertriebswegen wiederverwertet werden, um einerseits die Vorteile der Digitalisierung des Produktionsprozesses zu nutzen und andererseits den Aufwand für den Produktion durch den Einsatz eines einzigen Systems für die Redaktion zu minimieren.

### 2.2.2 Redaktionelle Prozesse

Es gibt eine große Bandbreite von redaktionellen Prozessen. Immer, wenn die Erstellung von Dokumenten - hier im Sinne der allgemeinen Brockhaus-Definition von Abschnitt 2.1 zu verstehen - in einem Prozeß mit den Schritten Bearbeiten und Kontrollieren stattfindet, kann von einem redaktionellen Prozeß gesprochen werden. Ein bekannter redaktioneller Prozeß ist das *Vier-Augen-Prinzip*, das vor allem in Behörden eingesetzt wird.

Dieser Abschnitt stellt Beispiele für redaktionelle Prozesse vor und beschreibt, wie ein solcher Prozeß typischerweise bei einem Online-Redaktionssystem aussieht, das für die Erstellung und Pflege von Internet Websites eingesetzt wird.

#### Beispielprozesse

Redaktionelle Prozesse findet man beispielsweise bei:

- der Beschaffung von Büchern in einer Bibliothek,
- der Erstellung wissenschaftlicher Arbeiten,
- der Erarbeitung von Standardisierungsvorschlägen und
- der Verarbeitung von Nachrichten.

Bei der *Beschaffung von Büchern in einer Bibliothek* läuft unter den Rahmenbedingungen wie Etat und Platzangebot der Bibliothek ein redaktioneller Prozeß, in dem Leser und Mitarbeiter über die Werke der Autoren entscheiden. Die Leser kontrollieren die Werke der Autoren indirekt durch die Ausleihe von Büchern und durch die Nachfrage nach nicht vorhandenen Büchern. Die Mitarbeiter wiederum nehmen über die Sichtung und Auswahl von Veröffentlichungen sowie durch den Vergleich mit anderen Bibliotheken Einfluß. Eine Entscheidung über die Beschaffung eines Buches fällt so in sorgsamer Abwägung der Kosten und der zu erwartenden Ausleihe. Langfristig hängt von dieser Entscheidung der Zuspruch der Leser und der Ruf der Bibliothek ab.

Bei der *Erstellung von wissenschaftlichen Arbeiten* sowie der *Erarbeitung von Standardisierungsvorschlägen* haben die Autoren häufig eine Doppelfunktion inne. Sie erstellen Texte und gehören auch zu denen, die Texte beurteilen. Damit üben sie eine Kontrollfunktion aus. Bei der Erstellung der Texte versucht der Autor durch die Diskussion innerhalb von

Gremien oder Arbeitsgruppen, den Text zu verbessern, indem das Urteil anderer hinzugezogen wird. Was im Vorfeld von Konferenzen im kleinen Rahmen abläuft, kann später bei Konferenzen oder Arbeitstreffen auf einer höheren Ebene wiederholt werden. Im Endeffekt kann über diesen Abstimmungsprozeß, an dem viele Menschen beteiligt sind, die Qualität der Texte bestimmt und verbessert werden.

Die *Verarbeitung von Nachrichten* findet ebenfalls im Rahmen einer Arbeitsgruppe statt, die üblicherweise streng hierarchisch aufgebaut ist. Die Autoren sind eingeteilt in erfahrene und weniger erfahrene Redakteure. Ein Redaktionsleiter koordiniert die Arbeit der Redakteure und entscheidet über die Vorlage der Nachrichten beim Chefredakteur. Dieser wiederum entscheidet über die Veröffentlichung der Nachrichten und zeichnet sich verantwortlich für sie. Die Verarbeitung von Nachrichten ist ein anschauliches Beispiel für das Vier-Augen-Prinzip. Jede Entscheidung einer Hierarchie-Stufe muß von der nächsten bestätigt werden, bevor sie durchgeführt wird. Die Leistung der Redakteure ist die Auswahl einiger Informationen aus einem unüberschaubaren Angebot und die Aufbereitung der Informationen zu Nachrichten. Für den Leser müssen die Nachrichten glaubhaft sein. Da der Nachrichtenbereich für den Leser sehr transparent ist, sind Falschmeldungen oder tendenziöse Berichterstattung leicht aufzuspüren. Dies kann schnell zu einem Verlust an Vertrauen und Ansehen führen. Die Leser informieren sich an anderer Stelle, mit ihnen gehen die Anzeigenkunden und damit die Werbebudgets.

### Prozeß für Internet Websites

Online-Redaktionssysteme werden vorrangig zum Aufbau und zur Pflege einer Internet Website eingesetzt. Sie bieten ihre Leistungen für die Verarbeitung von Dokumenten an, aus denen die Website aufgebaut ist. Für Websites ist die Aktualität der präsentierten Dokumente ein wesentliches Kriterium, da Besuchern nur ständig aktualisierte und gepflegte Websites wieder besuchen.

Viele der Dokumente, die Websites anbieten, kommen aus externen Quellen. Die Redakteure wählen die Dokumente aus, bereiten sie für die Präsentation auf der Website auf und erstellen eigene Beiträge, die die Präsentation der Informationen abrunden. Es handelt sich bei dieser Art der Verarbeitung um ein sog. *Many of a kind*-Szenario. Wenn die Dokumente durch die Erzeugung einer *HyperText Markup Language* (HTML)-Seite visualisiert werden, haben sie ein weitgehend einheitliches Layout. Mit einheitlichem Layout ist hier nicht gemeint, daß ein Dokument in nur einem Layout wiedergeben werden kann, sondern, daß nicht jedes Dokument ein individuelles Layout hat. Dadurch sollen Dritte anhand einer HTML-Seite die Website erkennen und dem Betreiber der Website zuordnen können. Dies fördert die Ausbildung einer Organisationsidentität (*Corporate Identity*), der Idee, daß durch das gezielte Auftreten einer Organisation eine einheitliche Sicht auf die Organisation ausgebildet werden kann. Diese Sicht soll sowohl nach innen in die Organisation hinein als auch nach außen auf Kunden und die Öffentlichkeit wirken. Dazu werden eine kleine Menge von Symbolen einsetzt, die konsequent bei jedem Auftreten der Organisation verwendet werden. Von einer Corporate Identity spricht man dann, wenn imit dem Auftreten der Symbole die Organisation assoziiert wird.

Analog zu dem Layout der HTML-Seiten verhält es sich mit der Struktur der Dokumente. Eine Klasse von Dokumenten mit gleicher Struktur wird häufig als Dokumenttyp bezeichnet. Nicht jedes Dokument hat einen eigenen Dokumenttyp, sondern es wird angestrebt, nur wenige, einheitliche Dokumenttypen zu definieren. Indem man Dokumente von vorneherein einem Dokumenttyp zuordnet, stellt man eine Einheitlichkeit der Struktur der Dokumente sicher.

Die Personen und die Abschnitte eines typischen redaktionellen Prozesses in einem Online-Redaktionssystem sind in Abbildung 2.1 dargestellt. Zur Visualisierung wird ein *Anwendungsfalldiagramm* aus der *Unified Modeling Language* (UML) [Rumbaugh et al. 99] benutzt. Es identifiziert die *Aktoren* des Prozesses und ordnet ihnen *Anwendungsfälle* (*Use*

*Cases*) zu. Der zeitliche Ablauf des Prozesses mit den Zusammenhängen zwischen den Abschnitten ist in Abbildung 2.2 durch ein *Aktivitätsdiagramm* dargestellt. Die einzelnen *Aktivitäten* sind durch *Bedingungen* verbunden, die beim Übergang zur nächsten Aktivität erfüllt werden müssen. Üblicherweise sind die Schritte eines redaktionellen Prozesses wie in Abbildung 2.1 auf mehrere Benutzer mit unterschiedlichen Kompetenzen und Verantwortlichkeiten verteilt. Neben diesem redaktionellen Prozeß arbeiten Graphiker an der Gestaltung der Website. Zu ihrer Aufgabe gehört der Entwurf neuer Designs z.B. zur Erweiterung der Website um neue Themenbereiche oder zur kompletten Umgestaltung (*Relaunch*).

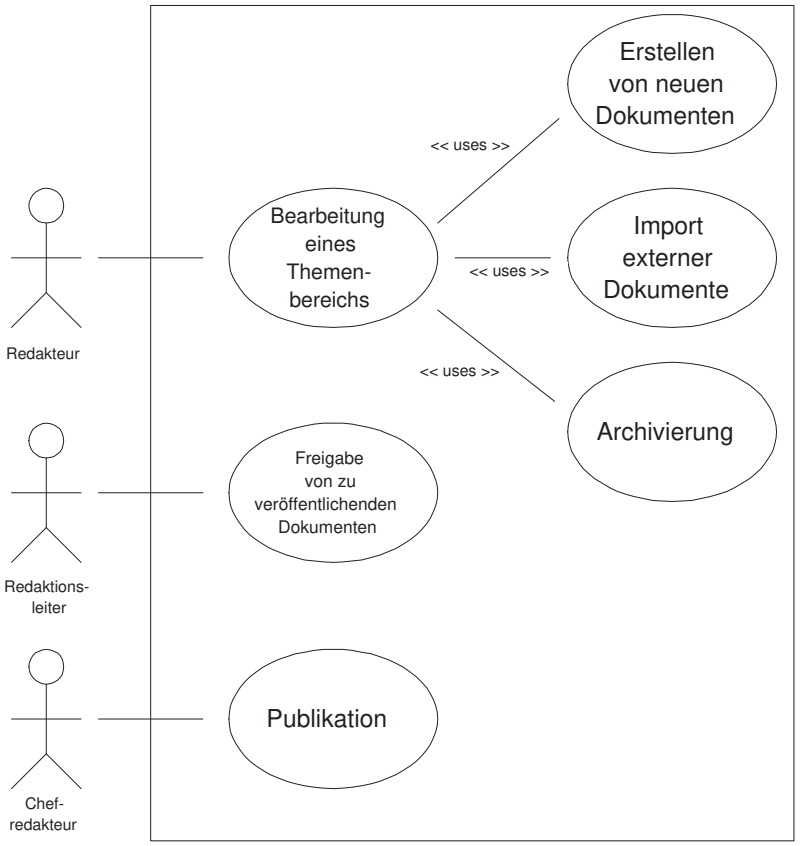


Abbildung 2.1: Aktoren und Anwendungsfälle eines redaktionellen Prozesses

Der *Import von externen Dokumenten* geschieht meist vollautomatisch aus den externen Quellen. Dazu müssen die Dokumente in das Dokumentenformat des Online-Redaktionssystems umgewandelt werden. Danach können die transformierten Dokumente in den Bestand eingegliedert werden. Dies kann manuell oder automatisiert auf der Basis von Regeln geschehen. Der Einsatz und die Formulierung der Regeln sind Teil der Konfiguration des Systems. Beim manuellen Ansatz ist die Arbeit des Systems beendet, sobald die Redakteure über den Import der externen Dokumente informiert werden. Im automatisierten Fall entscheiden die Regeln über die Weiterverarbeitung des Dokuments. Eine Regel enthält eine Menge von Bedingungen sowie die Beschreibung einer oder mehrerer Aktionen.

Für jedes importierte Dokument wird überprüft, ob die Bedingungen einer Regel auf das Dokument zutreffen. Ist dies der Fall, dann beschreiben die Aktionen der Regel, was mit dem Dokument geschehen soll. Beispielsweise kann das Dokument verworfen, an einer bestimmten Stelle im Dokumentenbestand eingegliedert oder gleich weiterverarbeitet werden.

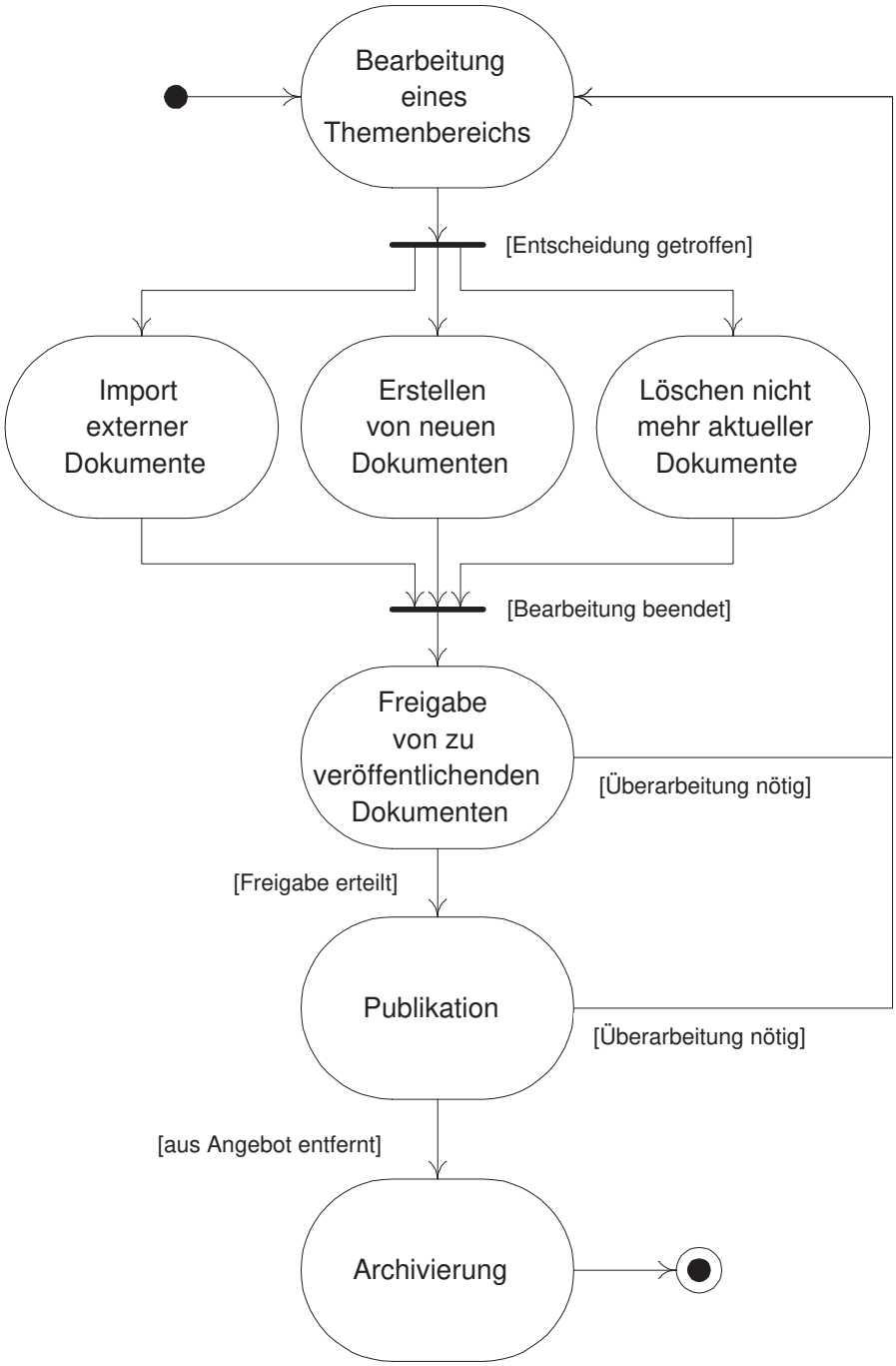


Abbildung 2.2: Aktivitäten eines redaktionellen Prozesses

Anschließend informiert das System die Redakteure über den Import und die Verarbeitung des neuen Dokuments.

Nachdem der Redaktionsleiter über die *Bearbeitung eines Themenbereichs* entschieden hat, beginnt ein Redakteur mit seiner Arbeit. Er entscheidet über das *Löschen nicht mehr aktueller Dokumente* aus dem Bestand. Diese werden während der Publikation aus dem aktuellen Bestand entfernt und stehen danach zur Recherche im Archiv bereit. Daneben sucht der Redakteur in den importierten Dokumenten nach neuen Informationen zum Themenbereich. Das *Erstellen von neuen Dokumenten* ist der nächste Schritt im redaktionellen Prozeß. Meist werden die Dokumente erstellt, um einen Themenbereich zu vervollständigen, da die externen Quellen nicht immer den ganzen Themenbereich abdecken. Jedoch wird selten ein Themenbereich komplett neu erstellt, da die importierten Dokumente die Basis für einen Themenbereich darstellen.

Die *Freigabe der zu veröffentlichenden Dokumente* geschieht nach der Durchsicht der Dokumente durch einen Redaktionsleiter, der die Arbeit der Kollegen kontrolliert. Hierzu gehört auch die Kontrolle der zu löschenden Dokumente. Für die *Publikation*, d.h. die Veröffentlichung der Dokumente, ist der Chefredakteur zuständig. Er ist im Sinne des Presserechts verantwortlich für die veröffentlichten Dokumente. Die *Archivierung* der aus den Themenbereich gelöschten, nicht mehr aktuellen Dokumente übernimmt das Online-Redaktionssystem selbständig. Es stellt diese Dokumente auf Anfrage wieder für die Bearbeitung zur Verfügung.

### 2.2.3 Eigenschaften

Die Anforderungen, die durch den Einsatzbereich und die Arbeitsweise an Online-Redaktionssystemene gestellt werden, spiegeln sich in den Eigenschaften der Systeme wieder. Zu ihren Leistungen gehören:

- die Definition und Implementierung redaktioneller Prozesse,
- die Einteilung der Benutzer in Gruppen,
- die Zuweisung von Rollen zu Benutzergruppen,
- die Definition von Zuständigkeiten für Benutzergruppen,
- die Verwaltung von Bearbeitungszuständen für Dokumente,
- die Versionierung der Dokumente,
- die Bildung von Varianten,
- die Archivierung aller Dokumente und
- die Unterstützung mehrerer Ausgabemedien.

Die Anforderung, Publikationen für die neuen Medien zu erlauben, klassifiziert die Online-Redaktionssysteme als eine Teilmenge der Systeme zur multimedialen Dokumentenverarbeitung. Die Eigenschaften dieser Systeme sind in Abschnitt 2.1.1 beschrieben. Darüber hinaus unterstützen Online-Redaktionssysteme die Produktion von Inhalten in den neuen Medien mit starkem Aktualitätsbezug über die *Definition und Implementierung redaktioneller Prozesse*.

Ein *redaktioneller Prozeß* nutzt Rollen, um die Kooperation der Benutzer [Berlin, O'Day 90] zu unterstützen. Ein bekannter redaktioneller Prozeß ist das Vier-Augen-Prinzip, das in Abschnitt 2.2.2 vorgestellt wurde. Mit dem Einsatz eines redaktionellen Prozesses kann

eine Arbeitsteilung zwischen den Benutzern erzielt werden, mit dem Ziel, die Geschwindigkeit der Dokumentenverarbeitung zu erhöhen. Durch die Einführung von Kontrollschritten wie beim Vier-Augen-Prinzip kann eine Qualitätskontrolle stattfinden, bevor Dokumente aus dem System herausgegeben werden. So wird die Qualität der Arbeit bzw. deren Ergebnis verbessert, indem die Schritte des Bearbeitens und Kontrollierens im redaktionellen Prozeß solange durchlaufen werden, bis die Arbeit freigegeben wird.

Bei der Unterstützung redaktioneller Prozesse durch ein Redaktionssystem ist die Einbettung des Systems in seine Umwelt nicht zu vernachlässigen: Beim Vier-Augen-Prinzip gibt es immer zwei Beteiligte. Ein Beteiligter erstellt die Dokumente. Die Kontrolle und Freigabe der Arbeit übernimmt ein zweiter Beteiligter, der die entsprechenden Befugnisse hat und die Verantwortung für die Freigabe übernimmt. Das Kongruenzprinzip [Wöhe 81] besagt, daß *Aufgabe*, *Kompetenz* und *Verantwortlichkeit* übereinstimmen müssen, damit ein Benutzer seine Rolle wahrnehmen kann. Sie müssen innerhalb und außerhalb des Systems definiert werden. Das System kann hier nur unterstützend wirken. Es kann dabei helfen, Aufgaben zu verteilen und Rechte zu vergeben. Für einen redaktionellen Prozeß ist diese Unterstützung durch das System wichtig, doch erreicht man eine Qualitätskontrolle nur, wenn Aufgabe, Kompetenz und Verantwortlichkeit außerhalb des Systems mit der Definition innerhalb überstimmen.

Die *Einteilung der Benutzer in Gruppen* erlaubt eine leichtere Administration des Systems, da bei Veränderungen nicht die Rollen oder Zuständigkeiten jedes einzelnen Benutzers, sondern lediglich die der Gruppe angepaßt werden müssen. Durch die *Zuweisung von Rollen zu Benutzergruppen* kann mehreren Benutzern aufgrund der Zugehörigkeit zu dieser Gruppe eine Rolle zugewiesen werden. Die Rollen sind die Basis für die *Definition von Zuständigkeiten* im redaktionellen Prozeß. Die Zuständigkeit meint hier die Aufgabe, die die Rollen im Prozeß zu erfüllen haben. Zur Erledigung dieser Aufgabe muß die Rolle die nötigen Kompetenzen definieren und die Verantwortlichkeit dafür feststellen. Die Definition von Kompetenzen kann über die Verteilung von Rechten vorgenommen werden. Durch die Protokollierung kann nachvollzogen werden, welche Änderunge erfolgt sind, um die Verantwortlichkeit für die Aktionen mit dem System festlegen zu können.

Die *Verwaltung von Bearbeitungszuständen für Dokumente* ermöglicht es, die Position des Dokuments im redaktionellen Prozeß festzulegen. Um diese Position bestimmen zu können, muß man die Historie der Zustandsänderungen des Dokuments kennen. Ein Bearbeitungszustand besteht u.a. aus dem Datum, an dem der Zustand erreicht wurde, dem Benutzer, der es in den Zustand gebracht hat, dem vorherigem Zustand und, sofern existent, dem nachfolgenden Zustand.

Die *Versionierung der Dokumente* erlaubt es einerseits, alte Zustände eines Dokuments zu rekonstruieren und andererseits einen neuen Bearbeitungszustand zu erzeugen. Die alten Zustände ergeben eine Versionshistorie, die die Entstehungsgeschichte des Dokuments aufzeigt. Der neue Bearbeitungszustand erlaubt es, ein Dokument zu modifizieren und gleichzeitig die älteren Zustände für andere Benutzer zur Verfügung zu stellen. Ohne einen Bearbeitungszustand existiert nur eine einzige Version des Dokuments und alle Benutzer sehen die kleinen, unfertigen Zwischenzustände während der Bearbeitung des Dokuments. Die Versionen können jeweils linear oder hierarchisch zueinander in Beziehung stehen. Bei linearer Versionshistorie besitzt jede Version nur eine Vorgänger- und eine Nachfolgeversion. Dadurch ist das gleichzeitige Bearbeiten einer Version durch mehrere Benutzer nicht möglich. Im Gegensatz hierzu steht eine hierarchische Versionshistorie. Dort tritt das Problem auf, daß gleichzeitig bearbeitete Versionen wieder zusammengeführt werden müssen. Dabei soll möglichst keine Änderung verloren gehen. Hierarchisch versionierte Dokumente findet man in der Software-Entwicklung in Form von Versionskontrollsystemen, in denen der Quelltext von Programmen verwaltet wird.

Die *Bildung von Varianten* eines Dokuments ermöglicht es, unterschiedliche Ausprägungen des Dokuments als zusammengehörig zu verwalten. Dies ist für Dokumente, die in

unterschiedlichen Sprachen oder für unterschiedliche Ausgabemedien vorliegen, sinnvoll. Ein anderes Beispiel ist die Berücksichtigung von rechtlichen Regelungen in Bedienungsanleitungen oder in der Dokumentation. Nur wenige Systeme bieten eine Unterstützung für Varianten, so daß ein Dokument in mehrere Dokumente aufgeteilt werden muß, von denen jedes eine Variante des Dokuments aufnimmt.

Die *Archivierung aller Dokumente* erlaubt, daß alle Dokumente, die einmal in das System aufgenommen wurden, jederzeit wieder abgerufen werden können. Dies ist möglich, auch wenn sie nicht mehr zum aktuellen Dokumentenbestand des Systems gehören, z.B. weil sie gelöscht und archiviert oder aktualisiert wurden.

Die *Unterstützung von mehrere Ausgabemedien*, das auch als *multimodales Publizieren* (*Cross Media Publishing*) bezeichnet wird, bietet die Möglichkeit, ein Dokument oder Teile eines Dokuments in mehreren, unterschiedlichen Medien auszugeben. Mögliche Ausgabemedien sind z.B.

**Print** Ausgabe auf Druckmaschinen für Zeitungen oder Magazine

**Copy On Laser Disk** (COLD) Kopie auf einem optischen Datenträger

**Videotext** Anzeige über den Fernseher

**Pager** Anzeige auf kleinen, mobilen Empfängern

**Short Message System** (SMS) Anzeige für Mobil-Telefone

**Wireless Application Protocol** (WAP) neue Variante für Mobil-Telefone

**World Wide Web** (WWW) Ausgabe im Internet

Da die Ausgabemedien sehr unterschiedlich sind, ist es nicht immer möglich, ein Dokument ohne inhaltliche Änderungen auf allen Medien auszugeben. Dies kann durch die Erstellung von mehreren Varianten desselben Dokumentes umgangen werden. Der Vorteil, daß ein Dokument die gesamte Information enthält und nicht aufgespalten oder verteilt werden muß, bleibt so erhalten.

### 2.2.4 Technologische Basis

Nachdem die Anforderungen an ein Online-Redaktionssystem feststehen und die Eigenschaften der Systeme charakterisiert sind, werden nun die Mittel zur Realisierung dieser Anforderungen vorgestellt. Dazu wird die technologische Basis beschrieben, die die Implementierung, Wartung und Weiterentwicklung maßgeblich beeinflußt und gleichzeitig auch begrenzt. Zum Vergleich der Online-Redaktionssysteme und im Hinblick auf die Implementierung der generischen Visualisierung sind zum einen die technischen Grundlagen und zum anderen die Arbeit des Benutzers mit dem System interessant. Zu den wesentlichen Eigenschaften der technologischen Basis zählen:

- die Programmiersprache,
- die Technik zur Erzielung der Persistenz,
- die Kommunikationstechnik zwischen Klient und Server,
- die Konfigurierbarkeit,
- die Skalierbarkeit,
- der Dokumentenbegriff,

- die Verknüpfungsverwaltung und
- die Bearbeitungswerkzeuge.

Die *Programmiersprache* bestimmt die Möglichkeiten bei der Implementierung, Wartung und Weiterentwicklung. Die Eigenschaften der Programmiersprache, d.h. ob sie interpretiert oder kompiliert, imperativ, funktional oder objektorientiert, statisch oder dynamisch typisiert ist, bestimmen wesentlich die Wartbarkeit, Erweiterbarkeit und Performanz des Systems. Mit interpretierten, imperativen, dynamisch typisierten Programmiersprachen, den sog. *Skriptsprachen* lassen sich schnell Ergebnisse erzeugen, da das Abstraktionsniveau der Skriptsprachen hoch ist, kein Übersetzungsschritt vor der Ausführung anfällt und die Veränderung des Programms zur Laufzeit möglich ist. Diese Vorteile werden mit einigen Nachteilen erkauft: Die dynamische Typisierung hat als Nachteil, daß zur Laufzeit des Programms *Typfehler* auftreten können. Ein Programm so zu testen, daß alle Typfehler gefunden und behoben werden können, ist jedoch nicht möglich [Cardelli, Wegner 85]. Dazu kommt die geringe Performanz des Interpreteransatzes, der auf den Aufwand zurückzuführen ist, der für die Interpretation des Codes nötig ist. Für den Aufbau komplexer, verteilter Systeme sind kompilierte Programmiersprachen mit statischer *Typüberprüfung* deutlich besser geeignet. Die Kompilation in direkt für den Prozessor ausführbare Befehle beschleunigt die Verarbeitung von Programmen erheblich. Durch ein statisch überprüfbares Typsystem können Typfehler zur Laufzeit vermieden werden. Die Notation der Typen beim Programmieren erfordert jedoch mehr Arbeit. Das *objektorientierte Paradigma* hat seine Überlegenheit gegenüber den anderen Ansätzen in den letzten Jahren bewiesen. Heutzutage werden komplexe, neue Systeme fast ausschließlich mit objektorientierten Programmiersprachen wie C++ [Ellis, Stroustrup 90] oder Java [Sun Microsystems Inc. 96b] realisiert.

Die *Technik zur Erzielung der Persistenz* ist der zweite Einflußfaktor für das Potential eines Online-Redaktionssystems. Die Verwendung von Dateien zur Speicherung der Dokumente und Verwaltungsinformationen bedeutet, daß das System die Leistung eines *Datenbankmanagementsystems* (DBMS) selbst erbringen muß. Dazu sind auf der Basis von Dateien Transaktionen, Sperren (Locking), Protokollierung (Logging) und Wiederherstellung (Recovery) zu implementieren [Lockemann, Schmidt 87]. Die Verwendung eines Standardproduktes ermöglicht es, bei der Implementierung des Online-Redaktionssystems auf einem höheren Abstraktionsniveau zu arbeiten. Ein objektorientiertes DBMS [Heuer 92] bietet den Vorteil, daß es das objektorientierte Datenmodell aus der Programmiersprache ohne Paradigmenbruch [Manthey 91] unterstützt. Im Gegensatz dazu spricht für ein relationales DBMS die Reife der Systeme sowie eine standardisierte Abfragesprache, die *Structured Query Language* (SQL) [ISO9075 87]. Die Reife der relationalen Datenbanken äußert sich in großer Stabilität und hoher Performanz. Dies sind beides Eigenschaften, die objektorientierten DBMS (noch) nicht zugesprochen werden. Der Nachteil der relationalen DBMS ist, daß eine aufwendige objektrelationale Abbildung des Datenmodells aus der Programmiersprache auf das Relationenmodell nötig ist. Dies wirkt sich negativ auf die Performanz aus. Nachteilig ist außerdem, daß in SQL nur eine geringe Menge von Basisoperationen standardisiert sind. Die Hersteller von relationalen DBMS bieten deshalb proprietäre Erweiterungen an, um weitere Funktionalität zur Verfügung zu stellen. Diese Erweiterungen sind jedoch nicht standardisiert. Sofern sie verwendet werden, müssen sie für jedes einzelne relationale DBMS angepaßt werden. So ist z.B. die für multimediale Systeme typische Speicherung von großen Binärdaten, den *Binary Large Objects* (BLOB), nicht genormt und in den gängigen DBMS gänzlich unterschiedlich gelöst.

Bei einer *Client-Server-Architektur* [Orfali et al. 96] ist die *Kommunikationstechnik* (*Middleware*) der dritte Einflußfaktor für das Potential eines Online-Redaktionssystems. Proprietäre Lösungen bieten hier den Vorteil, daß sie individuell auf die Problemlösung zugeschnitten werden können. Doch ebenso wie bei den Datenbanken spricht der erhebliche

Entwicklungsaufwand gegen proprietäre Lösungen und für den Einsatz von Standardprodukten. Die bekanntesten Middleware-Techniken sind die von der *Objekt Management Group* (OMG) genormte Common Object Request Broker Architecture (CORBA) [Pope 98], das Distributed Computing Environment (DCE) der *Open Software Foundation* und das Distributed Component Object Model (DCOM) von Microsoft [Orfali et al. 96]. Ein Austausch der Middleware ist sehr aufwendig, da sich die Middleware-Techniken zu stark unterscheiden. Systeme, die ausschließlich auf der Windows-Plattform laufen, benutzen DCOM als Middleware. Für CORBA als Middleware-Technik spricht die Plattformunabhängigkeit und die Standardisierung.

Eine Alternative bei der Erstellung eines Online-Redaktionssystem bietet die Nutzung eines Systems zur multimedialen Dokumentenverarbeitung. Damit lassen sich die Probleme umgehen, die bei der Wahl einer Persistenz- und Kommunikationstechnik entstehen können. Das Basissystem übernimmt diese Funktionen und erlaubt es, von der Persistenz- und Kommunikationstechnik zu abstrahieren. Auf dieser Basis wird dann das Online-Redaktionssystem entwickelt, wobei der Vorteil darin besteht, daß man sich auf die Kernfunktionalität konzentrieren kann. Das Problem dieses Ansatzes ist, ein geeignetes Basissystem zu finden, das aktiv weiterentwickelt wird, das eine gute technische Basis darstellt und das die Implementierung der Funktionalität, die für ein Online-Redaktionssystem nötig ist, nicht einschränkt.

Die *Konfigurierbarkeit* eines Online-Redaktionssystems ist ein Maß für die Eigenschaft des Systems, Anpassungen an die Bedürfnisse des Benutzers zu ermöglichen. Eine größere Flexibilität bei der Konfiguration sichert, daß die Bedürfnisse der Benutzer berücksichtigt werden können. Dies gilt sowohl bei der Installation als auch im laufenden Betrieb, wenn neue Anforderungen an das System gestellt werden. Beispielsweise sollte der Benutzer Aufbau, Layout und Präsentation der Dokumente festlegen können. Hier erweist es sich als vorteilhaft, wenn das System Layout und Inhalt der Dokumente trennt. Das Datenbankmanagementsystem, in dem die Dokumente abgelegt werden, muß wachsende Anforderungen erfüllen können. Dazu müssen alle Parameter der Datenspeicherung konfigurierbar sein. Dies zahlt sich beim Wachstum der Datenbasis, der Optimierung der Geschwindigkeit und der Verteilung des Systems aus. Die Bearbeitungszustände von Dokumenten können vordefiniert, in geringem Umfang anpaßbar oder frei definierbar sein. Dasselbe gilt für die Definition von Rollen. Insgesamt ist die Konfiguration eines eigenen redaktionellen Prozesses mit weiteren Rollen und anderen Abläufen sehr komplex. Da die Implementierung einer solchen Konfigurationsmöglichkeit dementsprechend aufwendig ist, kann ein eigener redaktioneller Prozeß nur in wenigen Systemen realisiert werden.

Ein Online-Redaktionssystem sollte mit steigenden Anforderungen mitwachsen können. Diese Eigenschaft der *Skalierbarkeit* wird üblicherweise über Maßzahlen wie die Anzahl

- der gleichzeitig arbeitenden Benutzer,
- der zu verwaltenden Dokumente,
- der abrufbaren Dokumente pro Sekunde und
- der generierbaren Dokumente pro Sekunde

gemessen. Darüber hinaus beeinflussen nichtmeßbare Eigenschaften wie Qualität, Erweiterbarkeit und Konfigurierbarkeit die Skalierbarkeit. Sie bestimmen, wie gut ein System zukünftige Anforderungen bewältigen kann.

Der *Dokumentenbegriff* bestimmt den Aufbau der Dokumente im System. Ein Dokument kann beispielsweise aus einer Menge von Name-Wert-Paaren besteht, die flach oder in einer hierarchischen Struktur abgelegt sind. Eine andere Möglichkeit ist, daß Dokumente aus einem mit Auszeichnungen strukturierten Text bestehen. Der Aufbau von Dokumenten, Dokumentenmodellen und Typsystemen wird in Abschnitt 3.1 ausführlich diskutiert.

Die *Verknüpfungsverwaltung* (*Linkmanagement*) sorgt dafür, daß logische Verknüpfungen (*Links*) zwischen Dokumenten im System definiert werden können und stets verfolgbar bleiben. Dadurch trägt die Verknüpfungsverwaltung zur Qualität der erstellen Inhalte bei. Die Verknüpfungen können uni- oder bidirektional ausgelegt sein. Unidirektionale Verknüpfungen zeigen nur vom referenzierenden Dokument, in dem sie definiert werden, auf das referenzierte Dokument. Bei bidirektionalen Verknüpfungen läßt sich auch der umgekehrte Weg verfolgen, d.h. man kann vom referenzierten Dokument zu den referenzierenden Dokumenten gelangen.

Die *Bearbeitungswerkzeuge* dienen zur Erstellung und Bearbeitung von Dokumenten, zur Ablage der Dokumente in einer Ordnerhierarchie und zur Visualisierung des redaktionellen Prozesses. Zusätzlich sollte die Administration des Systems mit der Unterstützung des Bearbeitungswerkzeugs möglich sein. Die Eigenschaften dieser Schnittstelle des Systems zum Benutzer werden in Abschnitt 2.3 näher erläutert.

### 2.2.5 Beispiele für Systeme

Nach den Anforderungen, den Eigenschaften und der technologischen Basis von Online-Redaktionssystemen folgt nun eine Aufzählung von Lösungsmöglichkeiten im Bereich der Online-Redaktionssysteme. Sie basiert auf einer Marktübersicht, die im Rahmen einer Diplomarbeit erstellt wurde [Zschau 00]. Die dort erstellte Marktübersicht hat als Ziel, einen möglichst breiten Überblick über den Bereich der Online-Redaktionssysteme zu erzielen. Insgesamt sind dort etwa 35 bis 40 Systeme aufgeführt, die dem Bereich der Online-Redaktionsssysteme zuzuordnen sind. Für diese Arbeit sind daraus vier Systeme ausgewählt worden, die eine Standardlösung zur generischen Visualisierung ermöglichen könnten, da sie sich als kommerziell erfolgreich erwiesen haben.

Die Beispiele für Online-Redaktionssysteme werden zunächst kurz beschrieben und dann ihre Eigenschaften und ihre technologische Basis tabellarisch verglichen (siehe Tabelle 2.1). Darauf baut schließlich die Entscheidung für das System, das zur Implementierung der generischen Visualisierung unter den erläuterten Randbedingungen als am besten geeignet erscheint.

#### StoryServer

Das StoryServer 4.0 [Vignette Inc. 99] System der Vignette Inc. ist das in den USA kommerziell erfolgreichste Produkt im Bereich der Online-Redaktionssysteme. Der Schwerpunkt des Systems liegt im *Internet Relationship Management*, d.h. dem Aufbau, der Verwaltung und Pflege von Kundenbeziehungen über das Internet. Die Mittel dazu sind eine personalisierte Internet Website, die Erstellung von Profilen über Kunden und die angebotene Inhalte sowie die Integration mit Warenwirtschaftssystemen. Ein sog. *Syndication Server* erlaubt den Austausch von Inhalten und Profilinformationen mit externen Systemen.

Der Arbeitsablauf des Systems umfaßt die Rollen Redakteur, Graphiker, Entwickler und Unternehmensentscheider (business manager). StoryServer verwaltet keine Inhalte, sondern lediglich Vorlagen für die Website, die mit externen Daten gefüllt werden können. Das System trennt die externen Inhalt und das über Vorlagen generierte Layout. Die Vorlagen sind versioniert und werden zusammen mit Metadaten gespeichert. Ein *Staging* genannter Prozeß schaltet Vorlagen für Besucher der Internet Website frei. Dabei erfolgt die Produktion der Vorlagen auf demselben System wie die Publikation der Vorlagen mitsamt den Inhalten. StoryServer unterstützt nur dynamisch generierte Web-Seiten. Diese Seiten werden in zwischenspeicherbare (*cacheable*), statische und nicht-zwischenspeicherbare (*non-cacheable*), dynamische Anteile untergliedert.

Das StoryServer System ist auf zwei Systemplattformen verfügbar und nutzt die gängigen relationalen Datenbanken zur Speicherung der Vorlagen sowie als Quelle für die externen

Inhalte. Die Datenbanken werden zur Verbesserung der Performanz über einen *Connection Pool* angesprochen. Zur Kommunikation zwischen einzelnen Prozessen benutzt das System ein *Remote Procedure Call* (RPC) Protokoll [Nelson 81]. Zur Auslieferung der Web-Seiten werden Netscapes Fasttrack, Apache, Microsoft Internet Information Server und IBMs WebSphere unterstützt. Das System ist in der Skriptsprache TCL geschrieben. Die graphische Benutzungsoberfläche zur Verwaltung der Vorlagen wurde in Java erstellt. Die Vorlagen basieren auf HTML, das durchsetzt ist mit TCL- und SQL-Befehlen. Die Inhalte liegen außerhalb des Systems und müssen mit externen Bearbeitungswerkzeugen erstellt und gepflegt werden. Die Bearbeitungsmöglichkeit für die Inhalte ist für jede Installation individuell zu erstellen. Üblich ist der Einsatz von HTML-Formularen, die als Mittel zur Gestaltung von Benutzungsoberflächen in Abschnitt 2.3 vorgestellt werden.

#### Online Redaktionssystem

Das Online Redaktionssystem 3.0 [Imperia 99] von Imperia besitzt in Deutschland die größte Installationsbasis. Der Schwerpunkt des Systems liegt auf der Realisierung einer preiswerten Publikation im Internet.

Das System unterstützt keinen Arbeitsablauf. Es existiert ein einfaches Rechtesystem für die Rollen Redakteur und Graphiker, um die Publikation zu steuern. Die Produktion der Inhalte und Vorlagen erfolgt auf demselben System wie die Publikation. Das System trennt Inhalt und Layout nicht. Es können Vorlagen hinterlegt werden, die ausschließlich bei der Erstellung neuer Inhalte zum Einsatz kommen. Diese Vorlagen sind versioniert. Das System verwaltet Metadaten und untypisierte Binärdaten, die die Web-Seiten enthalten. Die Verwaltung von Verknüpfungen zwischen diesen Daten ist nur rudimentär vorhanden. Eine Validitätsprüfung der produzierten Inhalte ist nicht möglich.

Das Online Redaktionssystem basiert auf Perl Skripten und ist auf meisten Systemplattformen verfügbar, für die eine Perl Umgebung bereit steht. Zur Kommunikation zwischen einzelnen Prozessen benutzt das System das *HyperText Transfer Protocol* (HTTP) Protokoll [Stevens 96]. Die Binärdaten mit den Web-Seiten werden in Dateien verwaltet. Zur Auslieferung der Web-Seiten werden die Web-Server von Netscape, Microsoft und Apache unterstützt. Das System erstellt Web-Seiten ausschließlich dynamisch. Die Web-Seiten basieren auf HTML, das durchsetzt ist mit Perl-, SQL- und JavaScript-Befehlen. Die Inhalte werden über HTML-Formulare gepflegt, die für jede Installation individuell zu erstellen sind.

#### VIP

Die Versatile Internet Plattform (VIP) 3.0 [Gauss Interprise AG 99] ist laut der Gauss Interprise AG das innovativste Online-Redaktionssystem am Markt. Der Schwerpunkt des Systems liegt in der Wartung der Inhalte durch externe Werkzeuge. Die Koordination der externen Werkzeuge mit dem System und die Ansicht der produzierten Web-Seiten im Browser sind zentral für die Arbeit mit dem System.

Der Arbeitsablauf des Systems sowie die Rollen der Benutzer sind frei definierbar. Standardmäßig existieren die Rollen Redakteur und Graphiker und ein Arbeitsablauf, der ein ISO 9000-konformes Qualitätsmanagement [Lumbrecht 93] ermöglicht. VIP trennt Inhalt und Layout nicht. Es können Vorlagen hinterlegt werden, die ausschließlich bei der Erstellung neuer Inhalte zum Einsatz kommen. Eine Versionierung ist nicht vorgesehen. Das System verwaltet nur Metadaten zu den sog. Objekten. Dies sind untypisierte Binärdaten, aus denen die Website aufgebaut wird. Ein Staging genannter Prozeß kopiert die Objekte vom Pflege- aufs Produktionssystem. Die Verwaltung von Verknüpfungen zwischen den Objekten ist nur rudimentär vorhanden. Eine Validitätsprüfung der produzierten Inhalte ist nicht möglich.

VIP basiert auf Java und ist auf den meisten Systemplattformen verfügbar, für die eine Java Laufzeitumgebung bereit steht. Zur Kommunikation zwischen den einzelnen Systemen benutzt das System *Sockets* [Stevens, Wright 95] und ein proprietäres Protokoll. Die Objekte werden in Dateien verwaltet, für die Metadaten wird ein eigenes DBMS benutzt. Zur Auslieferung der Web-Seiten werden die Web-Server von Netscape und Microsoft unterstützt. Das System liefert nur statisch generierte Seiten aus. Sog. dynamische Seiten werden durch die Verteilung auf vorhandene statische Seiten realisiert. Dazu dienen Vorlagen mit einer einfachen Syntax, die das Einfügen der Metadaten der Objekte oder der Objekte selbst erlauben.

### CoreMedia

Mit dem CoreMedia Content Publisher 3.0 [Higher-Order Informations- und Kommunikationssysteme GmbH 99] bezeichnet sich die Higher-Order GmbH als Technologieführer im Bereich der Online-Redaktionssysteme. Der Schwerpunkt des Systems liegt auf einem offenen, auf Standards basierten, multimedialen Kernsystem, das einfach anzupassen und flexibel erweiterbar ist.

Der Arbeitsablauf des Systems ist den redaktionellen Prozessen einer Nachrichtenagentur angelehnt und fest vorgegeben. Die Rollen der Benutzer sind frei definierbar. Standardmäßig existieren die Rollen Redakteur, Graphiker, Ressortleiter sowie Chef vom Dienst. CoreMedia trennt Inhalt von Layout in Dokumenten und Vorlagen. Das System erlaubt *Cross Media Publishing* durch die Trennung in medienneutrale Inhalte in Dokumenten auf der einen Seite und Vorlagen für die medienspezifische Aufbereitung auf der anderen Seite. Es existiert ein Typsystem, um die Strukturierung der Daten in Dokumente zu beschreiben. Alle Daten in den Dokumenten inklusive ihrer Metadaten werden versioniert. Die Vorlagen liegen außerhalb des Systems und müssen mit externen Werkzeugen erstellt und bearbeitet werden. Ein Staging genannter Prozeß gleicht die Dokumente zwischen Produktions- und das Publikationssystem ab und erhält so die Konsistenz der publizierten Inhalte.

CoreMedia basiert auf Java und ist auf allen Systemplattformen verfügbar, für die eine Java Laufzeitumgebung bereit steht. Zur Kommunikation zwischen den Systemteilen wird CORBA als Kommunikations-Middleware verwendet. Die Dokumente und Metadaten des Systems werden in einer relationalen oder in einer objektorientierten Datenbank abgelegt. Die Datenbank wird zur Verbesserung der Performanz über einen *Connection Pool* und eine eigenes *Job Scheduling* angesprochen Zur Auslieferung der Web-Seiten können beliebige Web-Server benutzt werden. CoreMedia unterstützt dynamisch und statisch generierte Web-Seiten. Die Vorlagen für die Web-Seiten werden in zwischenspeicherbare (cacheable), statische und nicht-zwischenspeicherbare (non-cacheable), dynamische Anteile untergliedert. Die auch als *Templates* bezeichneten Vorlagen basieren auf den *Java Server Pages* (JSP) [Sun Microsystems Inc. 99b] und müssen durch externe Werkzeuge bearbeitet werden. Für die Bearbeitung der Dokumente und die Steuerung des redaktionellen Prozesses existiert eine graphische Benutzungsoberfläche.

### Fazit

Auf Basis der kurz vorgestellten Eigenschaften einiger ausgewählter Online-Redaktionssysteme (siehe Tabelle 2.1) ist ein System zu wählen, das aufgrund seiner technologischen Basis eine erfolgreiche Implementierung einer generischen Visualisierung verspricht und gleichzeitig durch eine weite Verbreitung den Grad einer Standardlösung erreichen könnte.

Aus der Beschreibung der technologischen Basis ist zu ersehen, daß von den vier ausgewählten Systemen nur das CoreMedia System mit einem Dokumentenbegriff arbeitet,

| Kriterium | Vignette StoryServer 4.0 | Gauss Interprise VIP 3.0 | Imperia Online Redaktionssystem 3.0 | Higher-Order CoreMedia 3.0 |
|---|---|---|---|---|
| Programmiersprache | TCL | Java | Perl | Java |
| Persistenz | SQL-DB | Dateien | Dateien | SQL/OO-DB |
| Kommunikationstechnik | RPC | Sockets | HTTP | CORBA |
| Vorlagenbearbeitung | GUI | externe Werkzeuge | externe Werkzeuge | externe Werkzeuge |
| Inhaltsbearbeitung | HTML-Formulare | externe Werkzeuge | HTML-Formulare | GUI |
| Schnittstelle zu redaktion. Prozeß | - | GUI | - | GUI |
| Konfigurierbarkeit | sehr umfangreich | eingeschränkt | gering | umfangreich |
| Konfigurationsaufwand | sehr hoch | hoch | sehr hoch | mittel |
| Skalierbarkeit | mittel | hoch | niedrig | sehr hoch |
| redaktion. Prozeß | - | beliebig | - | fest |
| Rollen | fest | beliebig | fest | beliebig |
| Dokumentenbegriff | - | untypisierte Binärdaten | Name-Wert-Paare | strukturierte Information |
| Vorlagen | HTML mit TCL und SQL | sehr rudimentär | HTML mit Perl, SQL und JavaScript | JSP |
| Seitengenerierung | nur dynamisch; Caching möglich | nur statisch | nur dynamisch | statisch oder dynamisch; Caching möglich |
| Verknüpfungsverwaltung | nur zwischen Vorlagen | eingeschränkt und fehlerträchtig | externer Prozeß überprüft Verknüpfungen | konsistenzerhaltend |

Tabelle 2.1: Vergleich der Online-Redaktionssysteme

der Inhalt und Layout trennt sowie über Dokumenttypen eine weitergehende Strukturierung der Inhalte bietet. Das StoryServer-System befaßt sich nicht mit der Verwaltung von Dokumenten. Das VIP-System und das Online Redaktionssystem von Imperia sehen Dokumente als unstrukturierte Daten an. Da die Möglichkeit, Dokumente generisch zu visualisieren, wesentlich von der Information über die Strukturierung der Inhalte abhängt, wären bei diesen Systeme grundlegende Eingriffe in die Systemarchitektur nötig. Für solche Maßnahme ist ein vollständiger Einblick in die Quelltexte der Systeme notwendig, um entsprechende Modifikationen vorzunehmen. Dieser Punkt bedeutet einen weiteren Vorteil für das CoreMedia Online-Redaktionssystem: Der Hersteller des Systems kooperiert mit Hochschuleinrichtungen und erlaubt den Einblick in die Interna des Systems. Eingriffe am Kernsystem, die für die generische Visualisierung multimedialer Dokumente nötig sein könnten, sind so möglich.

Aufgrund dieser Argumente wird die Implementierung in dieser Arbeit auf Basis des CoreMedia Online-Redaktionssystems durchgeführt. Die Grundlagen zur generischen Visualisierung, die Architektur und die Implementierung werden in Hinblick auf diese Entscheidung für CoreMedia vorgestellt.

## 2.3 Gestaltung der Schnittstelle zum Benutzer

Die Ausrichtung auf eine Benutzergruppe erlaubt bzw. erfordert eine Einschränkung der Lösungsmöglichkeiten für die Gestaltung der Schnittstelle zum Benutzer. Die Benutzer eines Online-Redaktionssystems sind Redakteure, die für die Erstellung der Inhalte zuständig sind und Graphiker, die die Präsentation der von den Redakteuren erstellten Inhalte übernehmen. Die Hauptkompetenz der Redakteure liegt im Bereich des Journalismus, die der Graphiker beim Entwurf des Layouts. Es handelt sich bei beiden Benutzergruppen um technische Laien.

Eine intuitiv bedienbare *Benutzungsschnittstelle* vergrößert den Kreis der Benutzer eines Systems um die Benutzer, die wenig Erfahrung im Umgang mit dem System haben. Für die Darstellung und Bearbeitung von multimedialen Inhalten werden Schnittstellen zum Benutzer benötigt, mit denen die Inhalte effizient bearbeitet werden können. An die Benutzungsschnittstelle eines Online-Redaktionssystems werden diese beiden Anforderungen gleichzeitig gestellt. Zum einen soll mit den Redakteuren eine Benutzergruppe mit dem System arbeiten, die Kenntnisse aus anderen Bereichen einbringt. Zum anderen soll das System diesen Benutzern die Erstellung und Verarbeitung multimedialer Inhalte ermöglichen.

Bei der Gestaltung von Benutzungsschnittstellen hat sich in den letzten 15 Jahren ein Paradigmenwechsel vollzogen. Die Einflußfaktoren dabei sind

- die wachsende Informationstechnik-Verbreitung,
- das steigende Ressourcenangebot und
- der verbesserte Bedienkomfort der Rechner.

Das steigende Ressourcenangebot der Rechner ermöglichte neue Anwendungsbereiche für den Einsatz von Informationstechnik und erforderte eine verbesserte Bedienung. Dies wiederum führte zu einer weiteren Ausweitung der Einsatzmöglichkeiten. Die drei Faktoren haben sich so wechselseitig beeinflußt und verstärkt. Der verbesserte Bedienkomfort wird vor allem durch einen Paradigmenwechsel bei den Benutzungsoberflächen erreicht. Man ging von einer *Terminal*-basierten Bedienung vom Programmen zu einer *graphischen Benutzungsoberfläche* über. Mit dem Aufkommen des Internets entstanden daneben Benutzungsoberflächen auf der Basis von *HTML-Formular*en. Diese drei Bedienmöglichkeiten werden im Folgenden vorgestellt und auf ihre Tauglichkeit für die Einsatz in einem Online-Redaktionssystem geprüft.

### 2.3.1 Terminal

Ein Terminal stellt die Bedienschnittstelle für Großrechner dar. In einer solchen *Host*-Architektur bedient ein leistungsfähiger, zentraler Rechner eine große Anzahl von einfachen Terminals. Die Programme laufen auf dem Host ab und geben ihre Ausgaben an ein Terminal ab. Das Terminal kann zeichenbasierte *Bildschirmmasken* monochrom darstellen und bietet eine Tastatur zur Eingabe.

Das Vorgehen bei der Benutzung eines Terminals ist ablauforientiert: Der Benutzer wählt *Programmfunktionen* aus einer Hierarchie von Menüpunkten aus. Benötigt das Programm Daten vom Benutzer, um fortfahren zu können, so präsentiert das Terminal dem Benutzer eine Bildschirmmaske. Eine solche Bildschirmmaske enthält eine Menge von Beschriftungen und *Eingabefeldern*, die vom Benutzer vollständig ausgefüllt werden müssen. Danach überträgt der Druck einer ausgezeichneten Taste den Inhalt der Eingabefelder vom Terminal zum Großrechner. Die eingegebenen Daten werden dort vom Programm ausgewertet. Bei Fehlern erhält der Benutzer die Bildschirmmaske mit einem Fehlerbeschreibung zum erneuten Ausfüllen zurück.

In einer solchen Host-Architektur bildet der Host als zentrale Komponente den Flaschenhals des Systems. Deshalb ist er darauf optimiert, möglichst viele Ein-/Ausgabevorgänge bearbeiten zu können: Jede Eingabe muß innerhalb einer für den Benutzer akzeptablen Zeit zu einer Ausgabe führen. Die Ausgaben dürfen das System jedoch nicht überlasten. Zur Entlastung wird die Anzahl der Ein- und Ausgabezyklen zwischen dem Host und Terminals niedrig gehalten. Die Interaktivität zwischen dem Host und dem Benutzer vor dem Terminal ist damit niedrig.

Die Einflußmöglichkeit des Benutzers beschränkt sich auf die Reihenfolge, in der die Eingabefelder der Bildschirmmasken ausgefüllt werden, sowie auf die Auswahl der Menüpunkte, die die Funktionen des Programms zur Verfügung stellen.

Das Programm auf dem Host verbringt die meiste Zeit mit Warten. Es wird nur aktiv, wenn der Benutzer eine Programmfunktion auswählt und daraufhin eine neue Bildschirmmaske benötigt. Während der Benutzer die Bildschirmmaske ausfüllt, kann das Programm keinerlei Aktionen unternehmen, um ihn dabei interaktiv zu unterstützen. Deshalb sind Hilfen oder Auswahlmenüs, die das Ausfüllen der Bildschirmmasken durch erklärende Worte und vordefinierte Antworten erleichtern, nicht realisierbar.

Aufgrund der Beschränkung der allermeisten Terminals auf die oben beschriebene monochrome, zeichenorientierte Darstellung können mit diesen Terminals weder Grafiken dargestellt noch Audio- oder Videosequenzen abgespielt werden. Die Verarbeitung multimedialer Inhalte setzt dies jedoch voraus. Außerdem erfordert das technisch bedingte Bedienkonzept eines Terminals einen geschulten Benutzer. Es ist erhebliches Vorwissen für die wenig intuitive Bedienung notwendig. Beide Kritikpunkte zusammen lassen ein Terminal als Benutzungsschnittstelle für ein Online-Redaktionssystem als nicht geeignet erscheinen.

### 2.3.2 Graphische Benutzungsoberfläche

Eine graphische Benutzungsoberfläche ist heutzutage die am häufigsten eingesetzte Bedienschnittstelle für Arbeitsplatzrechner. Diese *Personal Computer* (PC) können unabhängig von anderen Rechnern arbeiten und Programme ausführen. In einer Client-Server-Architektur werden sie als Klient in lokale Netzwerke eingebunden. Die Programme einer Client-Server-Architektur laufen verteilt auf den Rechnern von Klienten und Servern ab. Die Klienten übernehmen die Präsentation der Daten des Programms und die Interaktion mit dem Benutzer. Die Server beeinhalten die Programmlogik, koordinieren die Klienten und speichern die Daten.

Die wichtigsten Eigenschaften eines Personal Computers als Arbeitsplatzrechner sind:

- Mehrprogrammbetrieb (*Multitasking*),
- Fenstertechnik (*Windowing*),
- graphische Präsentation (GUI),
- Bedienung per Maus,
- hohe Interaktivität und
- verbesserte Hilfefunktionen.

Der *Mehrprogrammbetrieb* erlaubt es, mehrere Programmme scheinbar parallel auszuführen. Um die Ausgaben der Programme besser verfolgen zu können, setzt man die *Fenstertechnik* ein. Sie erlaubt es, zur selben Zeit, die Ausgaben mehrerer Programme in verschiedenen Fenstern anzuzeigen. Alternativ können die Daten eines Programms in mehreren Fenster präsentiert und manipuliert werden.

Die *graphische Präsentation* erlaubt durch eine höhere Auflösung und umfangreichere Farbpaletten die Verwendung von Symbolen (*Icons*) und Grafiken. Bei der weitverbreiteten Schreibtischmetapher soll die Benutzungsoberfläche ein Abbild der Realität darstellen. Die Bedienung des Rechners soll dem Benutzer durch diesen Analogieschluß aus der Realität erleichtert werden. Eine derartige Benutzungsoberfläche nach der *Windows, Icons, Menus, Pointer*-Metapher [Wessel 98] gilt als intuitiv zu bedienen und ist heutzutage Standard bei den Personal Computern.

Die Programmfunktionen lassen sich über *Menüs* und mit Hilfe einer *Maus* als Eingabegerät anwählen. Dies ist im Vergleich zur Tastatur einfacher, da die Maus als Eingabegerät nicht die Kenntnis aller Tasten und Tastenkombinaten für den Aufruf der Programmfunktionen erfordert. Somit erweitert die alternative Benutzung von Maus und Tastatur die Aufrufmöglichkeiten für Programmfunktionen, da die Bedienung der Programme über die Tastatur für erfahrene Benutzer weiterhin möglich sein soll. Normalerweise werden mit der Maus überwiegend Programmfunktionen aus der Menüzeile, aus den Symbolleisten und durch die direkte Manipulation vom Programmobjekten (*Drag and drop*) ausgewählt, während die Tastatur zur Eingabe von Text dient.

Mit dem Mehrprogrammbetrieb, der Fenstertechnik, der graphischen Präsentation und der Maus ist eine *hohe Interaktivität* der Programme mit dem Benutzer möglich. Auf jede Aktion eines Benutzers kann sofort eine Reaktion des Programms folgen, die dem Benutzer entsprechend visualisiert wird. Die Wirkung einer Aktion kann so unmittelbar sichtbar sein, ebenso wie im Fehlerfall die Fehlermeldung. Die optische Hervorhebung, die sog. *Selektion*, der Daten in einem Fenster zeigt dem Benutzer, worauf sich seine nächste Aktion bezieht. Mit graphischen Mitteln wird der Fortgang längerer Programmfunktionen für den Benutzer angezeigt. Ein Abbruch laufender Programmfunktionen wird, sofern sinnvoll, immer angeboten.

In den Richtlinien für graphische Benutzungsoberflächen ist es üblich festzulegen, daß der Benutzer durch das Programm möglichst wenig in seinen Aktionen eingeschränkt sein sollte. Ein Mittel, um dies zu erreichen ist, die Daten des Programmes soweit wie möglich durch den Benutzer direkt manipulierbar zu gestalten. Jede Funktion des Programms sollte, sofern dies sinnvoll und möglich ist, jederzeit aufrufbar sein. Eine Ausnahme von dieser Regel sind *modale Dialoge*. Dies sind Fenster, die den Zugriff auf die anderen Fenster unterbinden und eine Benutzerinteraktion erzwingen.

Um die Funktionen des Programms möglichst jederzeit aufrufen zu können, stehen zum ihrem Aufruf mehrere Möglichkeiten bereit.

Dies sind:

- eine Menüleiste (*Menubar*), deren Menüeinträge eine Referenz aller Programmfunktionen darstellt,
- eine Symbolleiste (*Toolbar*) mit kleinen Symbolen (*Icons*) für die wichtigsten Funktionen,
- Tastaturkürzel (*Shortcuts*) für Funktionen, die ohne den Einsatz der Maus aufgerufen werden können,
- die direkte Manipulation von Programmobjekten,
- der Einsatz von *Assistenten*, die den Benutzer auf Wunsch durch umfangreichere Programmfunktionen führen.

In moderen Programmen sind diese Aufrufmöglichkeiten für die Programmfunktionen in der Regeln vom Benutzer anpaßbar. Damit der Benutzer die kurz gefaßten Menütexte und die vielen Symbole besser versteht, ist eine *Tooltip* genannte Hilfsfunktion üblich. Beim Verweilen des Mauszeigers auf einer beliebigen Komponente der Oberfläche wird einen kurzer Hilfetext eingeblendet. Für eine ausführlichere Hilfestellung existiert eine *kontextsensitive Hilfe*, die der Benutzer jederzeit abrufen kann. Sie zeigt Hilfetexte abhängig vom aktuellen Zustand der Oberfläche an. Zusätzlich gibt es eine Volltextsuche über den Hilfetext und ein Stichwortverzeichnis.

Soll der Benutzer ein Element aus einer Menge auswählen, stellt eine Auswahlliste (*Combobox*) die Elemente der Menge zur Auswahl bereit. Dies vereinfacht die Bedienung für den Benutzer, da er sich die verschiedenen Alternativen nicht merken oder in Handbüchern nachschlagen muß. Außerdem kann gewährleistet werden, daß der Benutzer nur Elemente aus einer vorgegebenen Menge auswählen kann. Für Eingabefelder gibt es einen vergleichbaren Mechanismus, der eine Historie der letzten Eingaben zur Verfügung stellt. Über eine Eingabeliste (*Listbox*) ist die Auswahl und Übernahme bereits gemachter Eingaben möglich.

Durch diese Vielzahl von Eigenschaften erhöht die Einführung graphischer Benutzungsoberflächen den Aufwand für die Kodierung von Programmen gegenüber dem Terminalbetrieb deutlich. In der Regeln wird für die Benutzungsoberfläche etwa die Hälfte der Ressourcen benötigt, die zur Erstellung eines Programmes bereit stehen [Grønbæk 88]. Der Grund dafür liegt in der hohen Interaktivität, die moderne graphische Benutzungsoberflächen dem Benutzer durch die genannten Eigenschaften bieten. Gegenüber einfacheren Ansätze wie dem Terminalbetrieb aus der Zeit der zentralisierten Großrechner bedeutet dies eine deutliche Steigerung der Komplexität.

Für die Verarbeitung multimedialer Inhalte ist ein Personal Computer mit einer graphischen Benutzungsoberfläche gut geeignet. Zum einen ermöglicht er die Bearbeitung von Grafiken, Audio- und Videosequenzen. Zum anderen bietet er eine intutive, vertraute graphische Benutzungsoberfläche, mit der ein weiter Personenkreis multimediale Inhalte erstellen und pflegen kann.

### 2.3.3 HTML-Formulare

Mit dem Aufkommen des Internets wurde mit den HTML-Formularen (FORM) eine der graphischen Benutzungsoberfläche verwandte Benutzungsschnittstelle möglich. Dabei dient ein Web-Browser als generischer Klient für Programme, die über einen Web-Server HTML-Formulare zur Bearbeitung ausgeben. Der Benutzer erhält so eine graphische Benutzungsoberfläche mit Grafiken, Symbolen und Bedienung mit der Maus. Funktionen

werden durch das Anklicken von Menüflächen, hinter denen Hypertext-Verknüpfungen stehen oder über spezielle Aktionsknöpfe aufgerufen. Daraufhin fordert der Browser eine neue HTML-Seite vom Web-Server an, die er dem Benutzer präsentiert. Zur Eingabe von Daten stehen dem Benutzer aufgrund der Einschränkungen der HTML-Formulare nur zwei Möglichkeiten zur Verfügung:

- Eingabefelder für Text und
- Listen, Auswahlboxen (*Checkbox*) und -knöpfe (*Radiobutton*).

Der Komfort einer solchen Lösung ist relativ niedrig [Rice et al. 95]. Die Eingabefelder bieten kaum Unterstützung bei der Bearbeitung der Texte. Die Auswahlboxen und -knöpfe sowie die Listen sind nur für die Auswahl aus einer fest vorgegebenen Menge sinnvoll.

Die Interaktivität bei einer solchen Lösung ist sehr gering. In diesem Punkt sind HTML-Formulare mit einem Terminal vergleichbar. Der Grund für die niedrige Interaktivität liegt darin, daß nur die Auswahl einer Hypertext-Verknüpfung durch den Benutzer zu einem Kontakt mit dem Web-Server führt. Dieser stellt ein neues HTML-Formular zusammen und wartet auf die nächste Anforderung durch den Benutzer. Eine Benachrichtigung des Benutzers bei Änderungen ist nicht möglich, da alle Aktionen vom Web-Browser des Benutzers ausgehen. So können Menüpunkte nicht deaktiviert und Programmobjekte nicht direkt manipuliert werden, da eine sofortige Rückmeldung auf die Aktion nicht möglich ist. Interaktive Hilfen können mangels geeigneter Mechanismen nicht zur Verfügung gestellt werden. Erläuternde Texte können jedoch direkt in den Formularen oder über Hypertext-Verknüpfungen, die auf weitere Hilfetexte verweisen, realisiert werden.

Die Gestaltung der Schnittstelle zum Benutzer mit Hilfe von HTML-Formularen vermag mittels des Web-Browsers Grafiken darzustellen sowie Audio- und Videosequenzen abzuspielen. Der umgekehrte Weg ist mit neueren Web-Browsern über den Einsatz von Techniken wie *PUT* und *POST* möglich. Eine solchen Lösung ist jedoch wenig intuitiv für den Benutzer, unkomfortabel zu bedienen und anfällig für Fehler.

Der Komfort bei der Bedienung ist aufgrund der eingeschränkten Möglichkeiten gering. Ungeübte Benutzer sehen sich vor allem bei multimedialen Inhalten mit zu vielen Einschränkungen und Behelfslösungen konfrontiert. So ist das Erscheinungsbild im Web-Browser kaum zu beeinflussen, da die Entscheidung über das Layout dem Web-Browser obliegt. HTML-Formulare sind für das Eingeben von Suchanfragen entwickelt worden. Als Benutzungsoberfläche für ein Online-Redaktionssystems sind sie nicht geeignet.

# Kapitel 3

# Grundlagen der generischen Visualisierung

Die Grundlagen der generischen Visualisierung bilden den konzeptuellen Hintergrund für diese Arbeit. Aufbauend auf den bereits vorgestellten Begriffen Dokument und Multimedia sind die Eigenschaften elektronischer Dokumente zu diskutieren. Da elektronische Dokumente nicht physisch repräsentiert werden, ergeben sich eine Reihe von Vorteilen gegenüber Papierdokumenten. So ist die logische Struktur eines elektronischen Dokuments nicht durch die graphische Struktur inhärent vorgegeben. Vielmehr kann die graphische Struktur beliebig aus der logischen Struktur des Dokuments hergeleitet werden.

Mit Dokumentenmodellen kann die Menge der Ausprägungen von Dokumenten beschrieben und dadurch die Anzahl ihrer logischen Strukturen begrenzt werden. Dadurch ist es möglich, Algorithmen zu erstellen, die Dokumente mit beliebigen logischen Strukturen bearbeiten und visualisieren können. Die Visualisierung eines solchen elektronischen Dokuments kann auf verschiedenen Ebenen stattfinden. Es gibt nicht nur *eine* Visualisierung, vielmehr stehen verschiedene Möglichkeiten zur Auswahl, von denen einige in diesem Kapitel beschrieben werden. Der Kontext der Arbeit spannt einen Lösungsraum auf und begrenzt die Menge der Lösungsmöglichkeiten zur Visualisierung.

Der Anwendungskontext erfordert eine graphische Benutzungsoberfläche, die in der Lage ist, alle Dimensionen von Dokumenten in einem Online-Redaktionssysteme zu visualisieren. Diese Dimensionen sind das Dokumentenmodell, das die Menge der möglichen Ausprägungen von Dokumenten beschreibt, die Versionshistorie, die für alle Dokumente zu verwalten ist und die Zustände, die die Dokumente während des Arbeitsablaufs im System einnehmen können. Damit grenzen diese Dimensionen die realisierbaren Möglichkeiten ein und formulieren gleichzeitig weitere Anforderungen an eine generische Visualisierung im Rahmen einer graphischen Benutzungsoberfläche.

Dieses Kapitel stellt den konzeptuellen Hintergrund der generischen Visualisierung von multimedialen Dokumenten vor. Es beschreibt die Eigenschaften elektronischer Dokumente und führt in einige Dokumentenmodelle ein. Das Dokumentenmodell des CoreMedia Online-Redaktionssystems, die Versionierung und die Zustände der Dokumente während des Arbeitsablaufs werden erläutert sowie ihre Auswirkungen auf die generische Visualisierung beschrieben.

## 3.1 Dokumentenmodelle

Dokumente stellen die Basis für die Arbeit mit Online-Redaktionssystemen dar. Die Dokumentenmodelle der Systeme grenzen die Menge der möglichen Dokumente und Visuali-

sierungen ein. Dieser Abschnitt präzisiert den Begriff des elektronischen Dokuments und definiert Begriffe für seine Strukturinformationen. Dokumentenmodelle können als eine Beschreibung der möglichen Ausprägungen von Dokumenten gesehen werden. Also legen Dokumentenmodelle die Menge der möglichen Dokumente fest. Einige Beispiele zeigen diese Eigenschaften der Dokumentenmodelle. Schließlich werden Möglichkeiten vorgestellt, wie Dokumentenmodelle selbst definiert werden können.

### 3.1.1 Elektronische Dokumente

In Abschnitt 2.1 wurden allgemeine Definitionen des Begriffs Dokument vorgestellt. Das wesentliche Merkmal von Dokumenten ist danach, daß sie aufgezeichnete Information darstellen. Für die Betrachtung von Dokumentenmodellen sind die vorgestellten Definitionen zu unscharf. Es gilt, den Begriff Dokument für die technische Betrachtung weiter einzuengen.

In der Büroautomatisierung [Khoshafian et al. 92] wird ein Dokument definiert als

> „zusammengehörende Ausprägung von Text, Ton oder Bild."

Vergleichbar ist die Definition aus der SGML-Norm [ISO8879 86]:

> „Ein Dokument ist eine Sammlung von Informationen, die als eine Einheit verarbeitet werden."

Beide Definitionen beziehen sich auf Dokumente, die per Rechner verarbeitet werden. Sie stellen die Einheit der Informationen, die das Dokument ausmacht, in den Vordergrund. Die Definition aus der SGML-Norm legt besonderes Gewicht darauf, daß die Sammlung von Informationen gemeinsam verarbeitet wird.

Beim Austausch von Dokumenten [Krönert 88] unterscheidet man direkte und indirekte Kommunikation. Bei direkter Kommunikation ist eine räumliche Distanz zu überbrücken, aber keine zeitliche. Die für die Kommunikation anfallenden Daten können als transient betrachtet werden, weil sie, wenn überhaupt, nur für den Zweck der Übertragung gespeichert werden müssen. Indirekte Kommunikation muß dagegen sowohl räumliche wie auch zeitliche Distanz überbrücken und bedarf deshalb eines dauerhaften Datenträgers. Für solch einen dauerhaften Datenträger wird verallgemeinernd die Bezeichnung Dokument verwendet.

Der Kontext der vorliegenden Arbeit gibt vor, daß solche Dokumente immer *elektronische Dokumente* sind, die per Rechner verarbeitet werden. Dabei steht die indirekte Kommunikation mit Dokumenten als dauerhaften Datenträgern für die Daten gleichberechtigt neben der direkten Kommunikation. Beides sind Kernfunktionalitäten eines Online-Redaktionssystems, das sowohl die persistente Speicherung der Dokumente, als auch die Auslieferung der Dokumente über ein digitales Medium leisten muß.

### 3.1.2 Bestandteile von Dokumenten

Dokumente werden aus den drei Bestandteilen Daten, Strukturen und Formatierung [Goldfarb 95] gebildet. Die Bestandteile und ihr Verhältnis zum Dokument sind in der Abbildung 3.1 graphisch dargestellt. Die verschiedenen Anwendungsgebiete bei der Verarbeitung von Dokumenten decken das in der Abbildung gezeigte Dreieck unterschiedlich ab.

Der Schwerpunkt bei der Erstellung von Dokumenten liegt auf der *Strukturierung* der Informationen, die das Dokument enthält. Für eine effiziente maschinelle Verarbeitung ist der Aufbau einer internen Struktur beim Lesen eines Dokumentes erforderlich. Eine

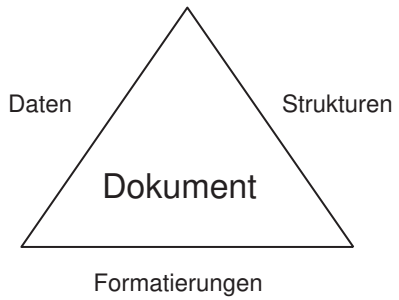


Abbildung 3.1: Dokumente und ihre Bestandteile

geeignete Strukturierung kann die Verarbeitung des Dokumentes daher deutlich vereinfachen. Vor allem Verarbeitungsprozesse mit mehreren Schritten profitieren davon. Die Aufbereitung des Dokuments zur Präsentation liegt meist in den Händen eines Experten für die graphische Gestaltung. Dieser beschäftigt sich besonders mit der *Formatierung* des Dokuments. Die *Daten* stellen den Informationsgehalt des Dokumentes da. Hierzu gehört neben dem Text auch die multimedialen Bestandteile wie Bilder, Grafiken, Audio- und Videosequenzen. Die Daten sind die vom Dokument über zeitliche oder räumliche Distanzen transportierten Informationen. Sie werden auch als *Inhalt* des Dokumentes bezeichnet.

Die *Strukturen* eines Dokuments werden auch als *logische Strukturen* bezeichnet [Wilhelm, Heckmann 96]. Sie gliedern das Dokument in Strukturelemente. So besteht z.B. diese Arbeit aus dem Titel, dem Inhalts- und Abbildungsverzeichnis, den Kapiteln, den Anhängen und dem Literaturverzeichnis. Die Kapitel bestehen aus einer Überschrift und einer Menge von Abschnitten. Man kann weiter fortfahren und diese Strukturierung bis zu atomaren Elementen wie Buchstaben und Linien herunterbrechen. Die vorgestellte logische Struktur ist hierarchisch organisiert. Daneben gibt es mit den Querverweisen eine weitere logische Struktur, die beliebige Strukturelemente des Dokuments zueinander in Beziehung setzt. Sie bringt damit eine zusätzliche, nichtlineare Struktur in das Dokument, so daß die logische Struktur ein Netz darstellt. Dokumente, deren Merkmal ist, daß die logische Struktur hauptsächlich mit Querverweisen gebildet wird, bezeichnet man auch als *Hypertext*. Die Bedeutung von Hypertext für Online-Redaktionssystemen wurde in Kapitel 2 erläutert. Das Dokumentenmodell für Hypertext-Dokumente wird in Abschnitt 3.1.3 vorgestellt.

Wenn sich in einem Dokument gewisse Strukturelemente wiederholen, spricht man von der *generischen Struktur*. Zu ihr gehören Aussagen wie z.B. daß es Kapitel und Abschnitte gibt und daß ein Kapitel aus einer nichtleeren Folge von Abschnitten besteht. Eine generische Struktur läßt sich z.B. mit Hilfe von *Typen* definieren.

Im Gegensatz zur generischen Struktur gibt die *spezifische Struktur* an, wie die logische Struktur in jedem einzelnen Fall realisiert ist. Wenn zur generischen Struktur die Aussage gehört, daß ein Kapitel aus einer nichtleeren Folge von Abschnitten besteht, dann kann eine Aussage zur spezifischen Struktur besagen, daß das dritte Kapitel aus genau sechs Abschnitten besteht. Die konkrete spezifische Struktur kann als ein *Exemplar* bzw. eine *Instanz* eines Typen angesehen werden.

Es ist durchaus möglich, daß verschiedene Exemplare von Dokumenten dieselbe generische Struktur besitzen. Dokumente übereinstimmender generischer Struktur gehören zu derselben *Dokumentklasse*. Man sagt auch, daß sie von demselben *Dokumenttyp* sind. Diese Begriffe werden in Abschnitt 3.2 näher erläutert.

Die Formatierungen eines Dokuments werden auch als die *graphische Struktur* oder *Layout* bezeichnet. Zur Formatierung gehören alle Aspekte, die die optische Wahrnehmung

eines Dokuments betreffen. Darunter fallen sowohl Eigenschaften der Zeichen wie Höhe, Breite oder Schriftart, Eigenschaften von Absätzen wie Ausrichtung oder Zeilenabstand als auch Eigenschaften weiterer Medien wie die Höhe, Breite, Farbe oder Rasterung von Bildern und Grafiken. Die logische Struktur kann auf einem Ausgabemedium nur durch passende Formatierungen wahrgenommen werden. Es ist also eine weitere Aufgabe von Formatierungen, die Aufnahme der Daten des Dokuments zu erleichtern.

Ein *Dokumentenmodell* legt die Menge der möglichen Ausprägungen der generischen Strukturen für Dokumente fest. Es begrenzt per Definition die Anzahl der Dokumentklassen. Es gibt unterschiedliche Methoden, mit denen ein Dokumentenmodell beschrieben werden kann. Dies kann beispielsweise über eine Konfigurationssprache geschehen, die die Elemente eines Dokuments definiert. Eine andere Möglichkeit ist ein Typsystem, das die möglichen Ausprägungen eines Dokuments durch Typen bestimmt.

In dieser Arbeit wird ein Dokumentenmodell als ein *Typsystem* für Dokumente aufgefaßt. Es liegt nahe, es mit den Mitteln eines Typsystems zu beschreiben. Die Dokumententypen, die sich in dem Dokumentenmodell definieren lassen, sind dann die Typen des Typsystems. Sie legen die generische Struktur für die Exemplare ihres Typen fest. Ein konkretes Dokument mit einer spezifischen Struktur ist solch ein Exemplar eines Dokumenttypen. Die Typsysteme von Programmiersprachen, Datenbanken und von CoreMedia werden in Abschnitt 3.2 vorgestellt und verglichen.

Zusammenfassend beschreibt ein Dokumentenmodell mit Hilfe von Dokumenttypen abstrakt, wie konkrete Ausprägungen von Dokumenten aussehen. Khoshafian et al. [Khoshafian et al. 92] definieren daher ein Dokumentenmodell vereinfachend als

> „Definition der möglichen Ausprägungen von Dokumenten“.

Ein Dokumentenmodell stellt also für ein konkretes System den Rahmen da, innerhalb dessen Dokumentklassen existieren können. Dies bietet die Möglichkeit, Aussagen über die Mächtigkeit des Dokumentenmodells zu machen und die möglichen Einsatzbereiche zu bestimmen.

### 3.1.3 Beispiele für Dokumentenmodelle

Dieser Abschnitt stellt zur Verdeutlichung einige Beispiele für Dokumentenmodelle vor. Für jedes Beispiel wird zunächst das Dokumentenmodell beschrieben und danach, sofern dies sinnvoll ist, ein Beispieldokument vorgestellt. Beispiele für Dokumentenmodelle sind:

#### SGML

Ein Dokument der *Standardized General Markup Language* (SGML) [ISO8879 86] ist definiert als eine Sammlung von Informationen, die als Einheit verarbeitet werden. Ein Dokument wird dadurch klassifiziert, daß es von einem bestimmten *Dokumenttyp* ist. Die Kennzeichnung der logischen Struktur geschieht durch die *Markierung* (*Markup*) des Dokumenteninhalts. Eine Markierung wird begrenzt durch *Marker* (*Tags*) zu Beginn und am Ende der Markierung. Die Marker definieren ein *Element*, das über *Attribute* weitere Informationen tragen kann. Elemente können beliebig tief verschachtelt werden. Diese Strukturierung des Dokuments ist das wesentliche Anliegen von SGML.

Anstatt die logische Struktur eines Dokuments durch entsprechende Formatierungen hervorzuheben, wird diese Struktur in SGML explizit gemacht. Dazu wird vorab der Dokumenttyp und damit die generische Struktur in einer *Document Type Definition* (DTD) definiert. Der SGML-Standard selbst definiert keine DTD, denn SGML ist eine Metasprache: Sie bietet eine standardisierte Methode, eine logische Struktur mit Hilfe einer DTD zu definieren.

Für die Formatierung des Dokuments, d.h. die graphische Struktur, stellt SGML keine Mechanismen bereit. Die mit SGML definierte logische Struktur stellt jedoch die Informationen bereit, die für die Visualisierung genutzt werden können. Mit der *Document Style Semantics and Specification Language* (DSSSL) [ISO10179 96] gibt es eine standardisierte Möglichkeit, die logische Struktur eines SGML-Dokumentes in eine graphische Struktur umzusetzen. Ein *Stylesheet* in der DSSSL-Sprache beschreibt dazu für die Elemente von SGML-Dokumenten einen Satz von Verarbeitungsregeln. Diese können benutzt werden, um eine spezifische graphische Struktur zur Präsentation des Dokuments zu erstellen. Diese Trennung von Inhalt und Layout hat dazu geführt, daß man sich bei SGML-Dokumenten auf den Inhalt beschränkt. Es existieren kaum Werkzeuge [Mekelburg 99] zur Präsentation von SGML-Dokumenten über DSSSL-Stylesheets.

Das wohl bekannteste Beispiel einer SGML-Dokumenttypdefinition ist die für *HyperText Markup Language* (HTML) Dokumente. Es ist die Sprache für die Hypertext-Dokumente im *World Wide Web* (WWW) [Tolksdorf 96]. Man spricht von HTML als einer *Anwendung* von SGML. Für die Formatierung des HTML-Dokuments, d.h. seine graphische Struktur, gibt es zwei unterschiedliche Ansätze, die in der Entwicklung des WWW begründet sind. Anfangs zeichneten Autoren lediglich die Struktur ihrer Dokumente durch Markierungen aus. Die Idee war, den Inhalt des Dokuments vom Layout zu trennen. Der Autor kümmert sich dabei ausschließlich um den Inhalt des Dokuments. Die Verantwortung für das Layout liegt vollständig bei den Programm, das zur Darstellung der HTML-Dokumente benutzt wird, dem *Browser*. Diese enthalten dazu einen Satz von Regeln, die bestimmen, wie jedes einzelne Element eines HTML-Dokumentes angezeigt wird. Manche Browser erlauben dem Benutzer, grobe Vorgaben für den Layout-Prozeß zu machen. Da es keine festen Vorgaben für die Visualisierung der Elemente gibt, ein einheitliches Layout jedoch häufig gewünscht ist, müssen die HTML-Dokumente für jeden Browser angepaßt werden. Dadurch haben die Autoren im Laufe der Zeit neben der Struktur- auch die Layout-Markierung der HTML-Dokumente übernommen.

Mit den *Cascading Style Sheets* (CSS) [Tolksdorf 96] wurde eine Möglichkeit geschaffen, analog zu den DSSSL-Stylesheets Verarbeitungsregeln für die Umsetzung der logischen Struktur in eine graphische zu definieren. Dadurch wird die Anpassung von HTML-Dokumenten an die Browser deutlich einfacher. Außerdem erreicht man durch die Aufteilung in HTML-Dokument und CSS-Stylesheet wieder eine Trennung von Inhalt und Layout, die zu Beginn des WWW angestrebt wurde und dann verloren ging.

Durch die Verbreitung des WWW, die Standardisierung und das Softwareangebot wurde HTML auch als Dokumentenformat für Anwendungen interessant, die über den ursprünglich anvisierten Einsatzzweck hinausreichen. Dazu ist es nötig, weitere Metainformationen bereitzustellen und die logische Struktur selbst zu definieren, um die Dokumente besser strukturieren zu können. Die Möglichkeiten, die durch die HTML-DTD vorgegeben sind, beschränken sich jedoch auf ein Element für Metainformationen sowie sechs Elemente für Überschriften und zwei verschiedene Aufzählungsarten zur Strukturierung.

Für die flexible Definition komplexer Strukturen bietet sich eine Metasprache wie SGML an. Mit den standardisierten Dokumenttypdefinitionen bietet SGML einen mächtigen Mechanismus an. Doch diese Mächtigkeit macht SGML sehr komplex. Programme, die SGML-Dokumente verarbeiten können, sind aufwendig zu erstellen und damit teuer. Deshalb entwickelte das *World Wide Web Consortium* (W3C) eine Lösung, die einfacher als SGML, aber flexibler als HTML ist. Die *eXtensible Markup Language* (XML) [DuCharme 99] stellte eine echte Teilmenge von SGML dar.

XML wurde mit dem Ziel entworfen, möglichst einfach und interoperabel mit HTML und SGML zu sein. Für XML-Dokumente ist die Definition einer DTD optional. Ein Dokument, dessen spezifische Struktur den Gültigkeitsregeln der XML-Spezifikation [DuCharme 99] gehorcht, nennt man *wohlgeformt* (*well formed*). Wenn eine DTD existiert und die spezifische Struktur des Dokuments der generischen Struktur gehorcht, die die DTD vorgibt,

nennt man das XML-Dokument *gültig* (*valid*). Durch diese Begrifflichkeiten und weitere Vereinfachungen, die in der XML-Spezifikation aufgeführt sind, lassen sich sehr einfach Programme zur Verarbeitung von XML-Dokumenten erstellen.

Die Formatierung eines XML-Dokumentes kann über zwei verschiedene Wege erfolgen. Eine Möglichkeit wird mit der *eXtensible Stylesheet Language* (XSL) gerade entwickelt. Damit soll analog zu DSSSL die Umsetzung der logischen Struktur eines XML-Dokuments in eine graphische Struktur über eine Menge von Verarbeitungsregeln möglich werden. Mit den Verarbeitungsregeln kann zur Präsentation des Dokuments z.B. in einem XSL-fähigen Browser eine spezifische graphische Struktur erstellt werden. Die Normung von XSL ist noch nicht abgeschlossen und es existieren im Moment noch keine XSL-fähigen Browser. Deshalb werden über *Transformatoren* XML-Dokumente in HTML-Dokumente übersetzt. Diese können dann mit den beschriebenen Möglichkeiten zur Formatierung von HTML-Dokumenten zur Präsentation aufbereitet werden.

Die Beziehungen zwischen den Markierungs- und den dazugehörigen Layout-Sprachen rund um XML und XSL sind in Abbildung 3.2 graphisch verdeutlicht.

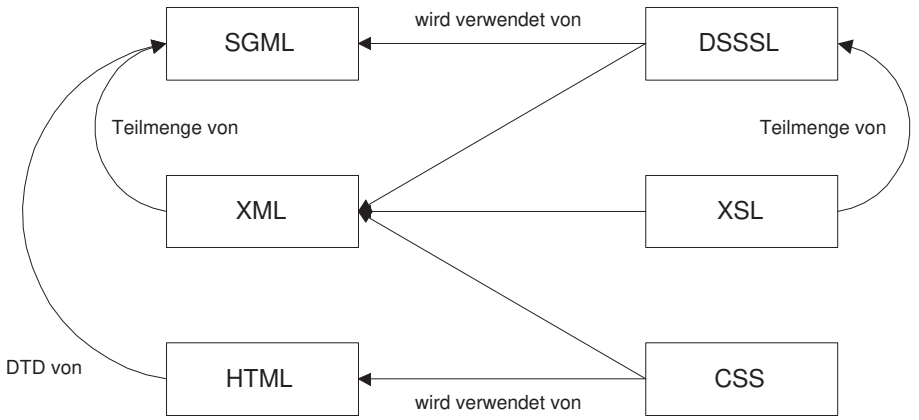


Abbildung 3.2: Beziehungen zwischen SGML, XML und HTML

### HTML

Die *HyperText Markup Language* (HTML) ist die Auszeichnungs- und Layoutsprache für Hypertext-Dokumente im World Wide Web. Um einen Eindruck von der charakteristischen Markierung durch die Marker zu vermitteln, wird ein Beispieldokument vorgestellt. Dazu zeigt Abbildung 3.3 ein Gedicht als Beispiel für ein HTML-Dokument. Die Regeln für den Aufbau einer DTD [Goldfarb 95] oder die aktuelle DTD für HTML [HTML-4.0 98] werden hier aus Platzgründen nicht beschrieben. Hierzu sei auf die Literatur verwiesen.

Im Beispieldokument findet man die Referenzierung der zu verwendenden HTML-DTD in der ersten Zeile. Danach sind einige Metainformationen im Kopf des Dokuments eingebettet. Der Rumpf besteht aus Paragraphen und Überschriften, die zur Strukturierung eingesetzt werden. Die Syntax für die Markierung des Inhalts läßt bei HTML große Spielräume: Bei den Namen von Elementen und Attributen wird nicht zwischen Groß- und Kleinschreibung unterschieden. Das Schließen der Beschreibungs- (META) oder Paragraphenmarkierung (P) mit Endmarkern ist nicht erforderlich.

Die logische Struktur des HTML-Dokuments wird durch die mit den Markern ausgezeichneten Elemente definiert. In Abbildung 3.4 ist die Baumstruktur der Elemente des HTML-Dokuments graphisch dargestellt. Jeder Kreis steht dabei für ein Element des Dokuments,

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 3.2//EN">
<HTML>
<HEAD>
  <META HTTP-EQUIV="Content-Type"
        CONTENT="text/html; charset=iso-8859-1">
  <TITLE>Weihnachten</TITLE>
</HeaD>
<body>
  <H1>Weihnachten</h1>
  <H2>von Joseph von Eichendorff</H2>
  <P>Markt und Straßen steh´n verlassen,<BR>
     still erleuchtet jedes Haus.<Br>
     Sinnend geh´ ich durch die Gassen,<BR>
     alles sieht so festlich aus.
  <p>An den Fenstern haben Frauen<br>
     buntes Spielzeug fromm geschmückt;<br>
     tausend Kindlein steh´n und schauen, <Br>
     sind so wunderstill beglückt. </p>
</body>
</HtMl>
```

Abbildung 3.3: Beispiel für ein HTML-Dokument

dessen Name im Kreis geschrieben steht. Die Kanten zwischen den Elementen stehen für die Enthalten-Sein-Beziehung der Elemente. Weiter unten stehende Elemente, die durch weiter oben stehende Elemente mit einer Kante verbunden sind, sind in den oben stehenden Elementen enthalten.

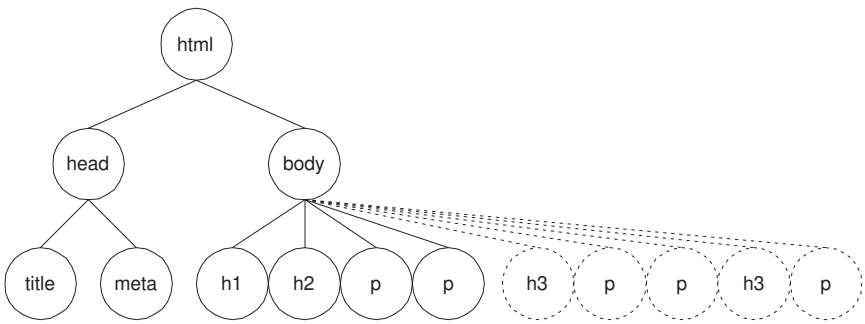


Abbildung 3.4: Baumstruktur des HTML-Dokuments

Wie die Abbildung zeigt, kann ein HTML-Dokument nur auf drei Ebenen strukturiert werden, da die Elemente im Rumpf, die zur Strukturierung des Textes in Paragraphen und Überschriften dienen, keine weiteren Elemente zur Strukturierung enthalten dürfen. Zudem ist mit den Elementen keine anwendungsspezifische Semantik verbunden. So nimmt das Kopfelement (H1) den Titel des Gedichtes und die Paragraphenelemente (P) die Strophen auf, weil dies bei der Darstellung im Web-Browser eine ansprechende graphische Struktur ergibt.

### XML

Die *eXtensible Markup Language* (XML) als eine vereinfachte Variante von SGML wird im Abschnitt 3.1.3 ausführlich erläutert. An dieser Stelle ist ein Beispieldokument eingefügt, um die Unterschiede zu HTML und die Möglichkeiten von XML anzudeuten. Für die Regeln zum Aufbau von XML-Dokumenten und XML-DTDs [DuCharme 99] sei hier aus Platzgründen auf die entsprechende Literatur verwiesen.

Als Beispiel ist das als HTML-Dokument (siehe Abbildung 3.3) vorgestellte Gedicht in Abbildung 3.5 nochmal als XML-Variante dargestellt. Dabei ist zu erkennen, daß XML-Dokumente mit einem Prolog beginnen, der aus der XML- und der DTD-Deklaration besteht. Danach folgen die Elemente zur Strukturierung der Information. Um zu entscheiden, ob das Dokument gültig ist, muß die DTD herangezogen werden. Doch auch ohne DTD kann entschieden werden, ob das Dokument wohlgeformt ist, da es den Gültigkeitsregeln der XML-Spezifikation gehorcht. Dazu gehört z.B. daß alle öffnenden Marker auch durch dazu passende schließende Marker geschlossen werden. Und im Unterschied zu HTML werden Groß- und Kleinschreibung unterschieden.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE poem SYSTEM "poem.dtd">
<poem>
  <title>Weihnachten</title>
  <author>Joseph von Eichendorff</author>
  <verse>
    <line>Markt und Straßen steh´n verlassen,</line>
    <line>still erleuchtet jedes Haus.</line>
    <line>Sinnend geh´ ich durch die Gassen,</line>
    <line>alles sieht so festlich aus.</line>
  </verse>
  <verse>
    <line>An den Fenstern haben Frauen</line>
    <line>buntes Spielzeug fromm geschmückt;</line>
    <line>tausend Kindlein steh´n und schauen,</line>
    <line>sind so wunderstill beglückt.</line>
  </verse>
</poem>
```

Abbildung 3.5: Beispiel für ein XML-Dokument

Die logische Struktur des Textes kommt im Vergleich zum HTML-Dokument deutlich hervor, da sie durch eine eigene DTD ausgedrückt werden kann. Ein HTML-Dokument muß dagegen mit den in der HTML-DTD vordefinierten Elementen auskommen. Dafür findet sich in der XML-Variante kein Hinweis zur graphischen Struktur des Dokuments, während die Formatierung der HTML-Variante durch die impliziten Formatierungsregeln für HTML-Elemente beschrieben wird.

Wie beim HTML-Dokument wird die logische Struktur des XML-Dokuments durch die Auszeichnung von Elementen mit Markern definiert. In Abbildung 3.6 ist die Baumstruktur der Elemente des XML-Dokuments dargestellt. Für die graphische Darstellung der Baumstruktur gelten dabei dieselben Hinweise wie für die HTML-Struktur in Abbildung 3.4. Die XML-Variante kann die Struktur des Gedichts ausdrücken, indem selbstdefinierte Elemente benutzt werden, ohne dabei eine graphische Struktur definieren zu müssen. Dies hebt z.B. hervor, daß die Verse des Gedichtes in Zeilen aufgeteilt sind, was in der HTML-Variante nicht möglich ist.

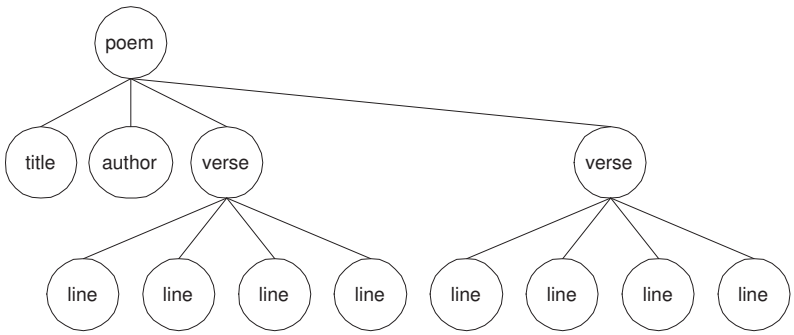


Abbildung 3.6: Baumstruktur des XML-Dokuments

### Hypertext

Als *Hypertext* bezeichnet man ein Dokumentenmodell für Texte, bei dem einzelne Textabschnitte als Knoten eines Graphen aufgefaßt und durch beliebige gerichtete oder ungerichtete Kanten miteinander verbunden werden können [Conklin 87]. Diese Kanten heißen auch *Verknüpfungen* (*Links*). Neben der linearen Aufreihung der Abschnitte können beliebige Querverweise auf andere Knoten eingerichtet werden. Man spricht deshalb auch von „nichtlinearem“ Text, in dem schnell navigiert werden kann. Die Verknüpfung der Textabschnitte ist das wesentliche Anliegen bei Hypertext-Dokumenten.

Beim Hypertext-Dokumentenmodell ist der gesamte Datenbestand als ein *Netz* von Textfragmenten, den Knoten, organisiert. Ein zusammenhängendes Netz wird als Hypertext-Dokument bezeichnet. Aufgrund dieser Netzstruktur spricht man auch von einem Informationsgeflecht. So kann Hypertext auch als universelle Organisationsform für große Mengen von Textdokumenten betrachtet werden.

Ein Beispiel für eine Hypertext-Anwendung ist das World Wide Web, das die multimedialen Hypertext-Dokumente des Internets enthält. Die Dokumente des WWW sind HTML-Dokumente, die Abschnitt 3.1.3 beschrieben wurden. Andere Beispiele für Hypertext-Systeme sind Hypercard [Apple Computer Inc. 88], Hoopertext [Berlin, O'Day 90], SEPIA [Lucarella et al. 92] und das Knowledge Management System [Meyer-Wegener 91]. Einfache Systeme wie das Knowledge Management System bilden bei der Erstellung der graphischen Struktur eines Dokuments ein Hypertext-Dokument auf eine Bildschirmseite ab. Außer den Verknüpfungen mit anderen Dokumenten existiert keine weitere Strukturierungsmöglichkeit. Die Systeme Hoopertext und SEPIA bieten darüber hinaus verbesserte Aufsuch- und Navigationsmöglichkeiten. Die Strukturierung und Visualisierung der Dokumente ist mit frühen Versionen von HTML vergleichbar, da ein Dokument lediglich logisch strukturiert wird und der Hypertext-Browser für die graphische Strukturierung sorgen muß.

### CoreMedia

Die in den vorherigen Abschnitten beschriebenen Dokumentenmodelle der SGML-ähnlichen Markierungssprachen und der Hypertext-Systeme unterscheiden sich prinzipiell von dem Dokumentenmodell, das hinter dem CoreMedia Online-Redaktionssystem steht. SGML und XML sind Metasprachen: Sie definieren eine Notation, mit der festgelegt werden kann, wie konkrete Dokumentstrukturen aussehen. Im Gegensatz dazu bietet CoreMedia keine Metasprache für die graphische Darstellung der Dokumentstrukturen an. Die Leistung von CoreMedia liegt darin, strukturierte Dokumente zu verwalten. Nur ausgehend von konkreten Dokumentstrukturen gelangt man zum abstrakten Dokumentenmodell, auf

dem CoreMedia basiert. Daraus folgt, daß es viele Möglichkeiten einer graphischen Visualisierung eines CoreMedia-Dokuments gibt. In Abbildung 3.7 ist die logische Struktur eines Beispieldokuments dargestellt. Der Dokumenttyp des hier dargestellten Beispieldokuments ist im Anhang A.2 definiert. Zur Visualisierung der Objekte der logischen Struktur wird ein *Objektdiagramm* aus der Unified Modeling Language benutzt.

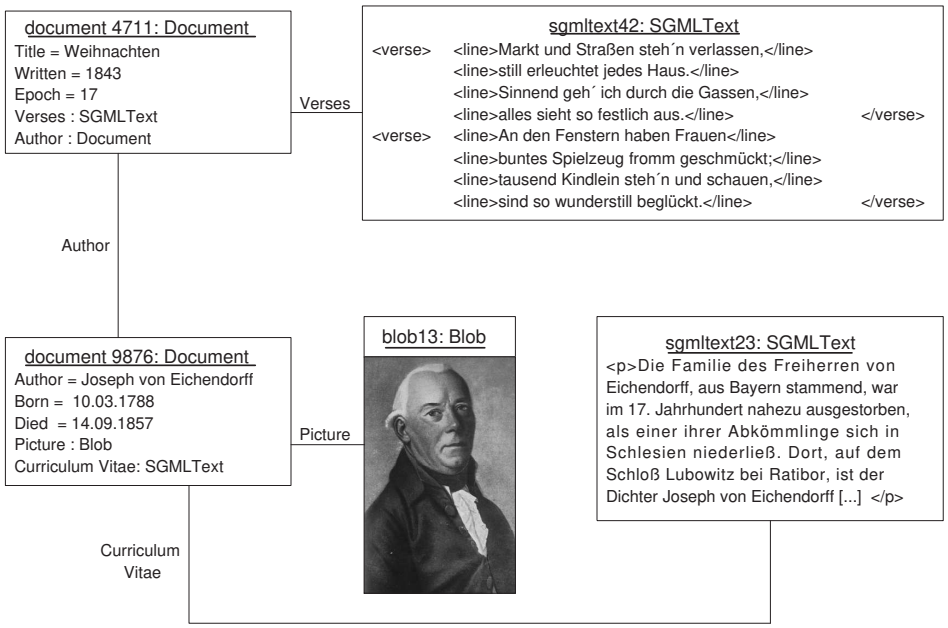


Abbildung 3.7: Objektdiagramm eines CoreMedia-Dokuments

Ein CoreMedia-Dokument wie das obige „document4711“ besteht aus einer Reihe von Objekten, die unterschiedliche Eigenschaften besitzen. Dies können Zeichenketten fester Länge („Title“) und Datumsangaben („Written“) sein, wie man sie aus relationalen Datenbanken kennt. Außerdem können CoreMedia-Dokumente komplexere Datenstrukturen enthalten. Dazu gehören mit Markierungen versehene Texte („sgmltext42“), die an SGML-Dokumente erinnern. Weiter kann ein CoreMedia-Dokument binäre Daten, d.h. Folgen von Bytes, enthalten. Beim Beispieldokument „document9876“ ergibt diese Bytefolge („blob13“) ein Bild, sofern die Kodierung des Bildes bekannt ist. Schließlich kann ein CoreMedia-Dokument Verweise auf andere CoreMedia-Dokumente enthalten (s.o. „Author“). Dies ist vergleichbar mit den Querverweisen eines Hypertext-Dokuments.

Der Aufbau von CoreMedia-Dokumenten mitsamt den verschiedenen Typen von Objekten, die im Dokument enthalten sein können, wird im Abschnitt 3.2.1 detaillierter beschrieben.

## 3.2 Typsysteme

In Abschnitt 3.1.2 wurde ein Dokumentenmodell als ein Typsystem für Dokumente bezeichnet. Umgekehrt erhält man durch die Definition von Dokumenttypen die Typen des Typsystems, das als Dokumentenmodell bezeichnet wird. Ein konkretes Dokument ist demzufolge ein Exemplar oder Wert eines Dokumenttyps. Diese Terminologie ist ganz analog zu der in Programmiersprachen [Cardelli, Wegner 85]. Genauso wie Typen in Programmiersprachen dazu dienen können, um Objekte nach ihrer Benutzung zu kategorisieren, können Dokumenttypen zur Klassifikation von Dokumenten dienen.

In diesem Abschnitt wird das Typsystem des CoreMedia Online-Redaktionssystems (siehe Abschnitt 2.2.5) mit dem von Programmiersprachen und Datenbanken verglichen. Damit kann die Mächtigkeit des Dokumentenmodells von CoreMedia leichter eingeschätzt werden.

### 3.2.1 CoreMedia

Im CoreMedia Online-Redaktionssystem besitzen Dokumente einen *Dokumenttyp*, der die generische Struktur der Dokumente definiert. Ein solcher Dokumenttyp hat einen eindeutigen Namen und beschreibt eine Menge von Attributen für die Dokumente. Die Attribute besitzen einen Namen, der sie eindeutig identifiziert, und einen *Attributtypen*, der die Werte des Attributs einschränkt. So wie Objekte in objektorientierten Programmiersprachen einen assoziierten Objekttyp haben, gilt dies für den Dokumenttyp der Dokumente und den Attributtyp der Attribute. Auf den CoreMedia-Dokumenttypen kann eine Subtypbeziehung definiert werden, die einen Inklusions-Polymorphismus [Cardelli, Wegner 85] auf den Subtypen erlaubt. Dabei ist die generische Struktur der CoreMedia-Dokumente auf der Ebene der Dokumenttypen immer identisch mit der spezifischen Struktur.

Im Gegensatz zu den Dokumenttypen findet man in CoreMedia nur vordefinierte Attributtypen. Die Definition weiterer Attributtypen ist nicht vorgesehen. Drei Attributtypen können lediglich eine fest vorgegebene Menge an Daten aufnehmen.

Dies sind:

- die längenbegrenzte Zeichenkette (BoundedString),
- die vorzeichenbehaftete Zahl (Integer) und
- das Datum (Date).

Daneben existieren Attributtypen, die Daten mit variabler Kardinalität enthalten können. Damit kann auf dieser Ebene die generische Strukur von der spezifischen abweichen. Es handelt sich um:

- eine Liste von Verweisen auf andere Dokumente (LinkList),
- ein typisiertes Binärobjekt (Blob) und
- einen SGML-Text (SgmlText).

Die Dokumenttypen sind in CoreMedia nicht orthogonal. Während es in SGML-Dokumenten möglich ist, beliebig tiefe Schachtelungen der Elemente ineinander zu haben, bietet CoreMedia diese Flexibilität nur bei Attributen, die SGML-Dokumente als Werte besitzen. Ein Binärobjekt kann beliebige Daten enthalten. Um diese Binärdaten bearbeiten zu können, besitzt ein Binärobjekt eine Typinformation, die Aussagen über die zulässigen Verarbeitungsmodi der Binärdaten zuläßt. Eine Liste von Verweisen in CoreMedia ist eine Mengenstruktur, die auf Verweise zu anderen Dokumenten beschränkt ist.

### 3.2.2 Programmiersprachen

In Programmiersprachen werden Werte von Typen unterschieden. Ein *Wert* kann definiert werden als [Watt 90]:

> „... alles, was ausgewertet, gespeichert, eingebettet in eine Datenstruktur, als Argument an eine Prozedur oder Funktion übergeben, als Funktionsergebnis zurückgegeben werden kann und so fort. In anderen Worten definieren wir, daß ein Wert alles sein kann, was während einer Berechnung existiert."

Üblicherweise gruppiert man Werte in Typen. Beispielsweise unterscheidet man Zahlen und Zeichenketten. Multiplikation und Division sind passende Operationen auf Zahlen, aber nicht auf Zeichenketten. Dementsprechend charakterisiert man einen *Typ* als eine Menge von Werten bzw. über die Operationen, die auf den Werten möglich sind. Unter den Typen unterscheidet man die primitiven und die zusammengesetzen Typen. Ein *primitiver Typ* ist atomar und kann nicht weiter in einfachere Typen zerlegt werden. *Zusammengesetzte Typen* bestehen aus primitiven und/oder weiteren zusammengesetzten Typen.

Bis zu diesem Punkt stimmen die Begriffe von Typ und Wert in Programmiersprachen mit denen von CoreMedia überein. Die *Typgleichheit* in CoreMedia beruht auf Namensgleichheit, während manche Programmiersprachen auch die strukturelle Gleichheit kennen. Das Prinzip der *Typvollständigkeit* in Programmiersprachen besagt, daß keine Operationen willkürlich in den Typen der Werte, auf denen sie arbeitet, eingeschränkt sein sollte. Im Vergleich zu Programmiersprachen ist das Typsystem von CoreMedia sehr klein und für einen speziellen Anwendungsfall konstruiert (siehe Abbildungen 4.8 und 4.9). Damit läßt es sich kaum mit den universell angelegten und deutlich mächtigeren Typsystemen von Programmiersprachen vergleichen. Das Typsystem in CoreMedia ist nicht orthogonal, da lediglich eine Zusammensetzung von Attributtypen zu Dokumenttypen unterstützt wird. Auch das Prinzip der *Typvollständigkeit* wird nicht erfüllt, da jeder Typ nur eine eingeschränkte, feste Anzahl von Operationen auf seine Werten erlaubt. Diese Einschränkungen des Typsystems rühren daher, daß die Erfüllung der Prinzipien eine für die das Anwendungsfeld der Online-Redaktionssysteme unnötige Komplexität mit sich brächte. Eine Programmiersprache dagegen ist darauf hin konzipiert, ein großes Feld von Anwendungen abdecken zu können.

### 3.2.3 Datenbanken

Die Typsysteme von Datenbanken hängen von den verwendeten Datenbankmodellen [Lockemann, Schmidt 87] ab. Das Netzwerk- und das hierarchische Datenbankmodell sind nicht mehr aktuell und werden daher nicht weiter beschrieben.

Das Typsystem einer relationalen Datenbank definiert eine Menge von *Basistypen*. Diese können bei der Definition der Mengenstruktur, den *Relationentypen*, benutzt werden. Es ist nicht möglich, die Basis- und die Relationentypen in einer anderen Art und Weise zu kombinieren. Das Typsystem ist also nicht orthogonal. Jedoch kann die Menge der Basistypen, die einen Relationentyp definieren, nach der Definition erweitert werden. Mit Programmiersprachen ist diese Erweiterung nicht möglich. Im Anwendungsfeld von Datenbanken, der langfristigen Speicherung von Daten, ist dies eine entscheidende Fähigkeit. Damit können Relationentypen erweitert werden, ohne daß Daten, die mit der alten Typdefinition gespeichert sind, ungültig werden. Die *objektrelationale Erweiterung* aktueller relationaler Datenbanken erlaubt die Definition von *Mengentypen*. Mit dieser zusätzlichen Struktur können Mengen von Basistypen gespeichert werden. Das so erweiterte Typsystem ist damit an dieser Stelle orthogonal, während es das ursprüngliche relationale Datenbankmodell nicht ist.

Im Vergleich zu CoreMedia zeigen sich kaum Unterschiede zwischen den Typsystemen. Das Typsystem relationaler Datenbanken ist zwar universeller angelegt und erlaubt die Definition eigener Relationentypen, um beliebige Anwendungsdomänen zu unterstützen. Davon abgesehen ist es nicht mächtiger als das Typsystem von CoreMedia. Dies erklärt sich dadurch, daß CoreMedia daraufhin konzipiert wurde, seine Daten auch in relationen Datenbanken speichern zu können. In diesem Fall muß CoreMedia eine Abbildung von den objektorientierten Strukturen der Programmiersprache auf die relationalen Strukturen der Datenbank realisieren. Bei der Benutzung einer objektorientierten Datenbank ist diese Abbildung nicht nötig. Sie vermeidet damit den *Impedance Mismatch* [Manthey 91] zwischen Objektorientierung in der Programmiersprache und Relationen in der re-

lationalen Datenbank. Ihr Typsystem ähnelt dabei so weit dem von objektorientierten Programmiersprachen, daß sie hier nicht weiter betrachtet werden.

## 3.3 Verarbeitungsweise

Dadurch, daß elektronische Dokumente nicht physisch repräsentiert werden, ergeben sich eine Reihe von Vorteilen. So ist es möglich, sie ohne Aufwand beliebig oft zu kopieren, sie maschinell zu bearbeiten und jede Repräsentation des Dokuments, die zur Präsentation aufbereitet wird, anders zu gestalten. Die graphische Struktur bei elektronischen Dokumenten ist im Gegensatz zu Papierdokumenten nicht inhärent im Dokument und ergibt sich somit nicht zwangsläufig aus der logischen Struktur. Die Strukturierung macht die Dokumente über die Syntaxanalyse durch sog. *Parser* [Aho et al. 87] maschinenles- und interpretierbar. Die Visualisierung der Dokumente ist dabei eine Möglichkeit, Dokumente zur Verarbeitung durch Benutzer zur Verfügung zu stellen.

Im Folgenden wird erläutert, welche Aufgabe die Visualisierung von Dokumenten hat und auf welchen Ebenen Dokumente visualisiert werden können. Eine geeignete Visualisierungsebene sowie Möglichkeiten zu ihrer Visualisierung werden beschrieben. Das Ziel ist eine graphische Repräsentation der Dokumente, so daß sie durch die Benutzer in der Zielgruppe (siehe Abschnitt 2.3) betrachtet und manipuliert werden können.

### 3.3.1 Ebenen der Visualisierung

Die unterschiedlichen Ebenen, auf denen CoreMedia-Dokumente visualisiert werden, um ihre Struktur darzustellen, ergeben sich aus den Teilen des Systems, in denen die Dokumente verarbeitet werden. Visualisierungen sind möglich:

- auf der Ebene der Speicherungsstrukturen,
- nach der Transformation in ein externes Format und
- für die Bearbeitung mit Hilfe einer graphischen Benutzungsoberfläche.

Die Visualisierung der *Speicherungsstrukturen* von CoreMedia-Dokumenten hängt von der Technik zur Erzielung der Persistenz (siehe Abschnitt 2.2.4) ab. Bei CoreMedia können hierzu relationale und objektorientierte Datenbanken [Heuer 92] benutzt werden. Diese unterscheiden sich in den Speicherungsstrukturen deutlich. Auf abstrakteren Ebenen läßt sich jedoch eine Abbildung [Manthey 91] von objektorientierten auf relationale Strukturen definieren. Zur Visualisierung der Speicherungsstrukturen können Beispiele für Tabellen oder ein *Entity Relationship*-Diagramm [Chen 76] für die Datenbeziehungen herangezogen werden. Zusätzlich können die flüchtigen Speicherungsstrukturen, die zur Laufzeit des Programms vorliegen, abgebildet werden. Ein Ausschnitt der graphischen Strukturen für die Verbindung von Dokumenten und SGML-Texten wird in Abbildung 3.8 dargestellt.

Das oberste UML-*Klassendiagramm* zeigt auf der Modellierungs-Ebene die Assoziation zwischen Dokumenten und SGML-Texten. Es besagt, daß ein Dokument beliebig viele SGML-Texte besitzen kann, wobei zu jedem SGML-Text das zugehörige Dokument eindeutig zugeordnet ist. Das mittlere Diagramm zeigt eine konkrete Implementierung dieser Modellierung in Java in Form eines UML-Objektdiagramms, die entsteht, wenn eine Abbildung der Objektstruktur auf eine relationale Struktur intendiert ist. Ein Java-Objekt für Dokumente unterhält dazu eine Assoziation zu einem zwischengeschalteten Textobjekt, das über eine Menge von SGML-Textobjekte aggregiert. Mit dieser Implementierung können sich mehrere Dokumente dieselben SGML-Texte teilen und gleichzeitig kann zu jedem SGML-Text das Dokument bestimmt werden, zu dem der SGML-Text gehört. Im

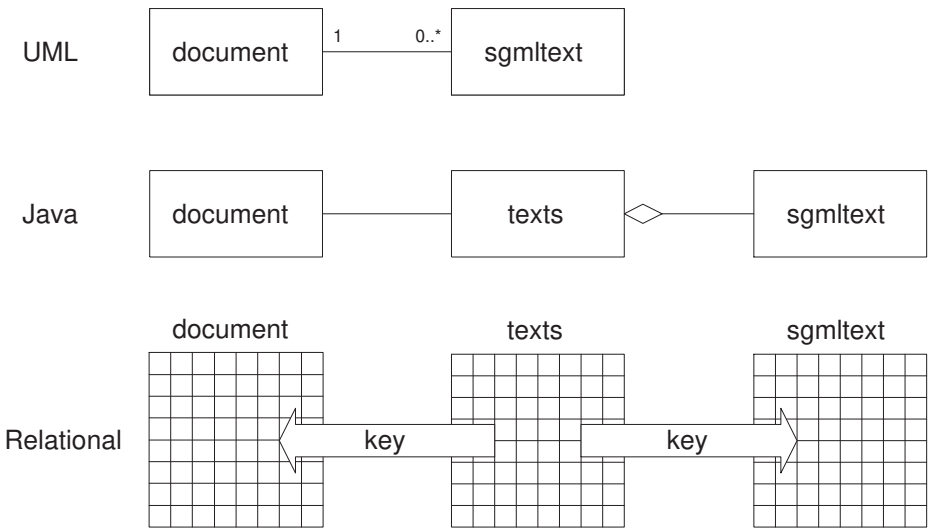


Abbildung 3.8: Speicherungsstrukturen eines CoreMedia-Dokuments

untersten Diagramm ist in einer freien Notation eine Realisierung dieser Beziehung in Form von Relationen im relationalen Datenbankmodell dargestellt. Sie ist eine Abbildung der Objektstrukturen des mittleren Diagramms auf Relationen.

Ein andere Möglichkeit, CoreMedia-Dokumente zu visualisieren, besteht darin, ein Objektdiagramm für die Objektbeziehungen heranzuziehen. Dabei geht man von einer objektorientieren Datenbank aus und abstrahiert von der objektrelationalen Abbildung. Ein Beispiel für ein Objektdiagramm eines CoreMedia-Dokuments wurde bei den Dokumentenmodellen in Abbildung 3.7 dargestellt. Alle diese Visualisierungen finden auf einer niedrigen Ebene des CoreMedia-Systems statt. In der Menge von Daten wird die logische Struktur der Dokumente nicht deutlich. Am adäquatesten ist die Darstellung der logischen Struktur durch die graphische Struktur eines Objektdiagramms möglich.

Das Ergebnis der *Transformation* eines CoreMedia-Dokuments in ein externes Format (siehe Abschnitt 2.1.1) ist eine weitere Möglichkeit, Dokumentstrukturen zu visualisieren. Durch die Transformation können jedoch Strukturinformationen verloren gehen, wenn das externe Format eine zu geringe Ausdrucksmächtigkeit besitzt, um ein CoreMedia-Dokument zu repräsentieren. Es ist nicht sinnvoll, ein Beispiel für ein transformiertes CoreMedia-Dokument zu geben, da das Ausgabemedium und der Ausgabezweck die Umsetzung der logischen auf eine graphischen Struktur einschränken. Die ursprüngliche logische Struktur bleibt nicht erhalten.

Die Visualisierung für die *Bearbeitung mit Hilfe einer graphischen Benutzungsoberfläche* dient zur Anzeige eines CoreMedia-Dokuments, während der Benutzer es bearbeitet (siehe Abschnitt 2.2.4). Da es sich hierbei genaugenommen auch um eine Transformation in ein externes Format handelt, ist von entscheidender Bedeutung, daß keine Strukturinformation verloren gehen. Ein Bearbeitungswerkzeug muß die konkrete logische Struktur ohne Einschränkungen innerhalb des durch den Dokumenttypen vorgegebenen Rahmen bearbeiten können.

### 3.3.2 Auswahl der Visualisierungsebene

Nachdem einige Beispiele für Visualisierungsebenen vorgestellt wurden, ist eine für die Aufgabenstellung geeignete Visualisierungsebene zu wählen. Der Kontext, aus dem sich die

Aufgabenstellung ergibt, spielt eine besondere Rolle. Er bestimmt die Visualisierungsebene aus den folgenden Gründen:

Die *Darstellung der Speicherungsstrukturen* ist nur interessant für Benutzer, die die Interna des Systems kennen oder kennenlernen möchten. Zur Bearbeitung von Inhalten innerhalb eines redaktionellen Prozesses ist diese Visualisierungsebene nicht geeignet.

Das Ergebnis der *Transformation* eines Dokuments in ein externes Format kann eine HTML-Seite auf einer Website, eine WAP-Seite für die Anzeige auf einem kleinen, mobilen Empfänger, eine SMS-Nachricht usw. sein. Ein Überblick über die Unterstützung mehrerer Ausgabemedien wurde in Abschnitt 2.2.1 gegeben. Diese Ebene der Visualisierung enthält eine externe Sicht auf den redaktionell erarbeiteten Inhalt des Dokuments. Dieser Inhalt wird durch die Transformation in eine dem Ausgabemedium angepaßte Formatierung verpackt. Eine solche Visualisierungsebene ist sinnvoll für die externen Benutzer des Systems, die das Ergebnis der redaktionellen Arbeit betrachten möchten. Es geht ausschließlich um das passive Konsumieren der erstellten Inhalte, nicht um die Bearbeitung innerhalb eines redaktionellen Prozesses.

Die Visualisierung für die *Bearbeitung mit Hilfe einer graphischen Benutzungsoberfläche* dient dazu, das Dokument dem Redakteur zur uneingeschränkten Bearbeitung innerhalb eines redaktionellen Prozesses zur Verfügung zu stellen. Die Rahmenbedingungen dafür legen die Anforderungen an Online-Redaktionssysteme fest, welche in Abschnitt 2.2.1 beschrieben wurden.

Eine Anforderung an eine geeignete Visualisierung ist, daß sie die logische Struktur des Dokuments in eine graphische Struktur transformiert, ohne dabei Strukturinformationen wegzulassen. Gleichzeitig dürfen die Grundprinzipien für den Aufbau von graphischen Benutzungsoberflächen (siehe Abschnitt 2.3.2) nicht verletzt werden. Es ist wichtig, daß sich die Elemente der Benutzungsoberfläche in das Benutzungsparadigma des Systems einpassen, ohne dabei Einbußen in der Funktionalität hinzunehmen. Dies kann z.B. der Fall sein, wenn Standardelemente verwendet werden, die zwar dem Benutzer vertraut sind, jedoch nicht genug Manipulationsmöglichkeiten bieten. Weiter ist entscheidend, daß die Elemente der Benutzungsoberfläche den gesamten Inhalt des Dokuments anzeigen können, sofern der Platz dies zuläßt. Außerdem gehört zu den Anforderungen, daß die Strukturelemente eines Dokuments in einer intuitiven Art und Weise plazierbar sind. Die bei lateinischen Schriften typischen Leseorientierung von links nach rechts und von oben nach unten ist zu beachten. Das bedeutet, daß Informationen, die sich links oben befinden, besser wahrgenommen werden, als Informationen, die rechts unten stehen [Döbele-Martin 97]. Neben der Plazierung ist Farbe ein weiteres Mittel, um die Aufmerksamkeit des Benutzers zu lenken. Dem setzen die Richtlinien für die Erstellung von graphischen Benutzungsoberflächen jedoch enge Grenzen.

Das Ziel dieser Arbeit ist es, eine graphische Struktur in einem Bearbeitungswerkzeug zur Verfügung zu stellen, das die generische Visualisierung und Bearbeitung der Dokumente ermöglicht. Eine Software-Architektur, die dies zum Ziel hat, wird in Kapitel 4 entworfen. Die Implementierung dazu wird in Kapitel 5 beschrieben. Die weiteren in diesem Abschnitt vorgestellten Ansätze zur Visualisierung von Dokumenten werden nicht weiter verfolgt.

## 3.4 Dokumentzustände

Bei der generischen Visualisierung der Dokumente sind neben dem Dokumentmodell und der Versionierung auch die Bearbeitungszustände der Dokumente während des Arbeitsablaufes zu berücksichtigen. Die Verwaltung von Bearbeitungszuständen gehört zu den Anforderungen, die in Abschnitt 2.2.1 an Online-Redaktionssysteme gestellt wurden. Der jeweilige Bearbeitungzustand soll für den Benutzer verständlich dargestellt werden und beeinflußbar sein. Zur Veranschaulichung sind die Zustände von Dokumenten in CoreMedia als *Zustandsdiagramm* in Abbildung 3.9 dargestellt.

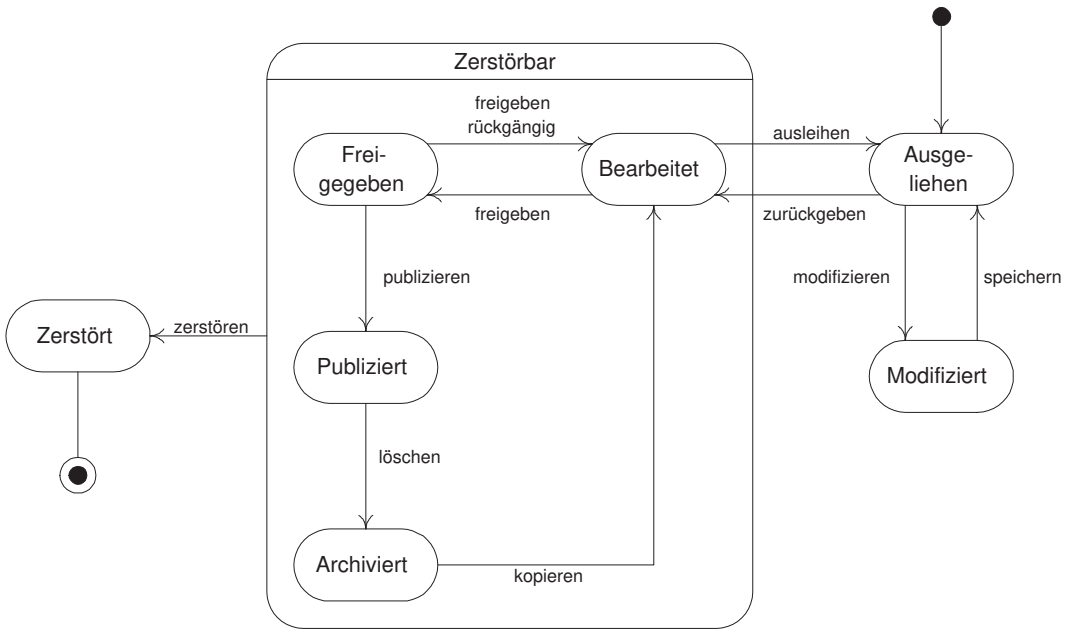


Abbildung 3.9: Zustände von Dokumenten

Aufgrund der Anordnung der Übergänge zwischen den Zuständen spricht man auch vom Lebenszyklus der Dokumente. Dieser Lebenszyklus stellt eine Zusammenfassung der Bearbeitungszustände für Dokumente in CoreMedia dar: Ein Dokument kommt von einer externen Quelle ins System oder wird neu erstellt und befindet sich im Zustand *Ausgeliehen*. Die Arbeit der Redakteure führt den Zustand des Dokuments von *Bearbeitet* beliebig oft in den Zustand *Ausgeliehen* und erzeugt dabei neue Versionen des Dokuments. Sobald eine Version des Dokuments entsteht, die publiziert werden kann, geht das Dokument in den Zustand *Freigegeben* über. Normalerweise folgt auf die Freigabe des Dokuments seine Publikation und der Zustand *Publiziert*. Wird die Freigabe rückgängig gemacht, befindet sich das Dokument wieder im Bearbeitungszyklus mit den Zuständen *Unbearbeitet* und *Ausgeliehen*. Das Löschen des Dokuments aus dem Angebot führt zur Archivierung des Dokuments im Zustand *Archiviert*. Aus dem Archiv kann das Dokument wieder in den Bearbeitungszyklus kopiert werden. Ein Sonderfall stellt das Zerstören eines Dokuments dar. Dies ist eine seltene Aktion, die das Dokument vollständig vernichtet, wenn rechtliche oder geschäftspolitische Gründe dies erfordern.

Für die Visualisierung von CoreMedia-Dokumenten ist dieser Lebenszyklus von Bedeutung. Dem Benutzer soll der jeweilige Bearbeitungszustand des Dokumentes trotz der Überlagerung des Bearbeitungszustandes durch die Versionierung klar dargestellt werden. Die Benutzungsoberfläche darf nur Aktionen für Zustandswechsel zulassen, die gemäß der Kanten in Abbildung 3.9 möglich sind.

# Kapitel 4

# Software-Architektur und Alternativen

Der Lösungsraum zur generischen Visualisierung multimedialer Dokumente innerhalb von Online-Redaktionssystemen wird durch eine Reihe von Faktoren eingeschränkt. Die Software-Architektur des CoreMedia Online-Redaktionssystems stellt einen wesentlichen Faktor dar. Ihre Eigenschaften begrenzen die Entwürfe für Alternativen zur generischen Visualisierung multimedialer Dokumente. Die Entwürfe sollen Dokumente mit multimedialen Inhalten innerhalb von CoreMedia mittels eines graphischen Bearbeitungswerkzeugs visualisieren und zur Bearbeitung durch Redakteure zur Verfügung stellen. Die Basis dazu bildet die Software-Architektur des CoreMedia Online-Redaktionssystems. Auf ihr setzen die alternativen Ansätze zur generischen Visualisierung auf. Diese Menge von Entwurfsalternativen decken das Feld zwischen den per Hand kodierten Lösungen auf der einen und den vollständig generierten Lösungen auf der anderen Seite gleichmäßig ab. Zwischen diesen beiden Polen werden eine Reihe von Entwürfen für Lösungen beschrieben. Durch die Variation der Entwürfe am Rand des Lösungsraumes können die beiden Pole einander so angenähert werden, daß sie sich überschneiden.

So bieten kodierte Lösungen eine maximale Flexibilität, dem ein hoher Aufwand bei der Erstellung der Benutzungsoberfläche gegenübersteht. Durch Abstriche bei der Flexibilität läßt sich der Aufwand verringern und die Benutzerfreundlichkeit erhöhen. Mit vollständig generierten Lösungen können ohne weiteren Aufwand Benutzungsoberflächen erzeugt werden. Diese generischen Lösungen stellen jedoch nur den kleinsten gemeinsamen Nenner, den Kern aller möglichen Bearbeitungsmöglichkeiten für Dokumente dar. Die Flexiblität einer solchen Lösung ist gering, so daß neue Anforderungen sofort den Rahmen der Lösung sprengen. Um den Preis eines höheren Aufwandes läßt sich die Flexibilität verbessern, bis der Bereich der kodierten Lösungen erreicht ist.

In diesem Kapitel wird zunächst die Software-Architektur des CoreMedia Online-Redaktionssystems vorgestellt. Sie bestimmt das Umfeld, auf dem Lösungen zur generischen Visualisierung aufbauen können. Eine Architektur zur generischen Visualisierung wird im Anschluß beschrieben. Sie spannt den Rahmen auf für eine Menge von Lösungen zur generischen Visualisierung. Es wird ein Schema vorgestellt, nach dem die Alternativen für die generische Visualisierung in die Lösungsmenge eingeordnet werden können. Die Entwürfe für die Lösungsalternativen werden vorgestellt und bewertet. Am Schluß steht die Entscheidung für die zu realisierende Lösung.

## 4.1 Architektur von CoreMedia

Die Architektur des CoreMedia Online-Redaktionssystems ist der Ausgangspunkt für die Implementierung einer generischen Visualisierung von multimedialen Dokumenten. Sie ist Voraussetzung für eine technisch saubere Implementierung zur Visualisierung der Dokumente mit Mitteln der Software-Technik wie z.B. begrenzter Sichtbarkeit, klaren Schnittstellen, fehlertolerantem Code, Entwurfsmustern usw. [Floyd 95; Denert 91; Gamma et al. 94]. Deshalb wird zunächst die Systemarchitektur von CoreMedia beschrieben. Diese resultiert aus den Anforderungen an Online-Redaktionssysteme, die in Kapitel 2 erarbeitet wurden. Die Umsetzung der Anforderungen bestimmt den technischen Kontext der generischen Visualisierung. Dazu gehört beispielsweise, daß eine Verteilung der Anwendung auf mehrere Rechner notwendig ist. Ein Modell für die örtliche Verteilung von Programmobjekten erlaubt es, das Verteilungsproblem auf einem höheren Abstraktionsniveau zu betrachten. Es wird ein abstraktes Verteilungsmodell vorgestellt, das eine Lösung für das Verteilungsproblem unter den durch den Kontext bestimmten Veraussetzungen bietet. Die Voraussetzungen resultieren aus Entwurfsentscheidungen auf der Ebene der Systemarchitektur, die die Verwendung bestimmter Komponenten für die graphische Benutzungsoberfläche erfordern. Diese Entscheidungen, die Konsequenzen daraus und die Auswirkungen auf das abstrakte Verteilungsmodell werden diskutiert.

### 4.1.1 Überblick

Das CoreMedia Online-Redaktionssystem ist als Client-Server-Anwendung konzipiert. Die Systemarchitektur in Abbildung 4.1 zeigt die Komponenten des Systems. Sie setzen die in der Anforderungsanalyse herausgearbeiteten Eigenschaften eines Online-Redaktionssystems (siehe Abschnitt 2.2.3) um. Da ein Online-Redaktionssystem immer auch ein System zur multimedialen Dokumentenverarbeitung darstellt, gehören auch die Eigenschaften eines Systems zur multimedialen Dokumentenverarbeitung (siehe Abschnitt 2.1.1) dazu. Die Anforderungen resultieren beim CoreMedia Online-Redaktionssystem in der vorgestellten Systemarchitektur, die den technischen Kontext für die generische Visualisierung bildet.

Der Anwendungsserver vereint die zentralen Dienste des Systems, die den Klienten zur Verfügung stehen. Die Dienste des Servers sind:

- die Speicherung, Verwaltung und Recherche der Dokumente mit Hilfe einer relationalen Datenbank,
- die Verwaltung von Benutzern, Gruppen und Zuständigkeiten,
- die Synchronisation des parallelen Betriebs der Benutzer und
- die Koordination des redaktionellen Prozesses.

Der Anwendungsserver greift über die *Java Database Connectivity* (JDBC) [Sun Microsystems Inc. 96a] Schnittstelle auf eine relationale Datenbank zu. Ein Zwischenspeicher für Ressourcen, der sog. *Ressourcen-Cache*, minimiert die Anzahl der Zugriffe auf die Datenbank, indem häufig benötigte Dokumente im Speicher gehalten werden. Zusätzlich werden nur die Attribute eines Dokuments aus der Datenbank geladen, die von einem Klienten angefordert werden. Beide Maßnahmen erhöhen die Geschwindigkeit des Anwendungsservers um einige Größenordnungen auf ein der Problemklasse angemessenenes Niveau.

Alle Klienten greifen über eine CORBA Schnittstelle auf den Anwendungsserver zu. Für den Austausch von Dokumenten mit anderen Systemen sorgen *Importer* und *Exporter*. Im Vergleich zu den anderen Klienten fragen die Im- und Exporter Dokumente häufig nur ein

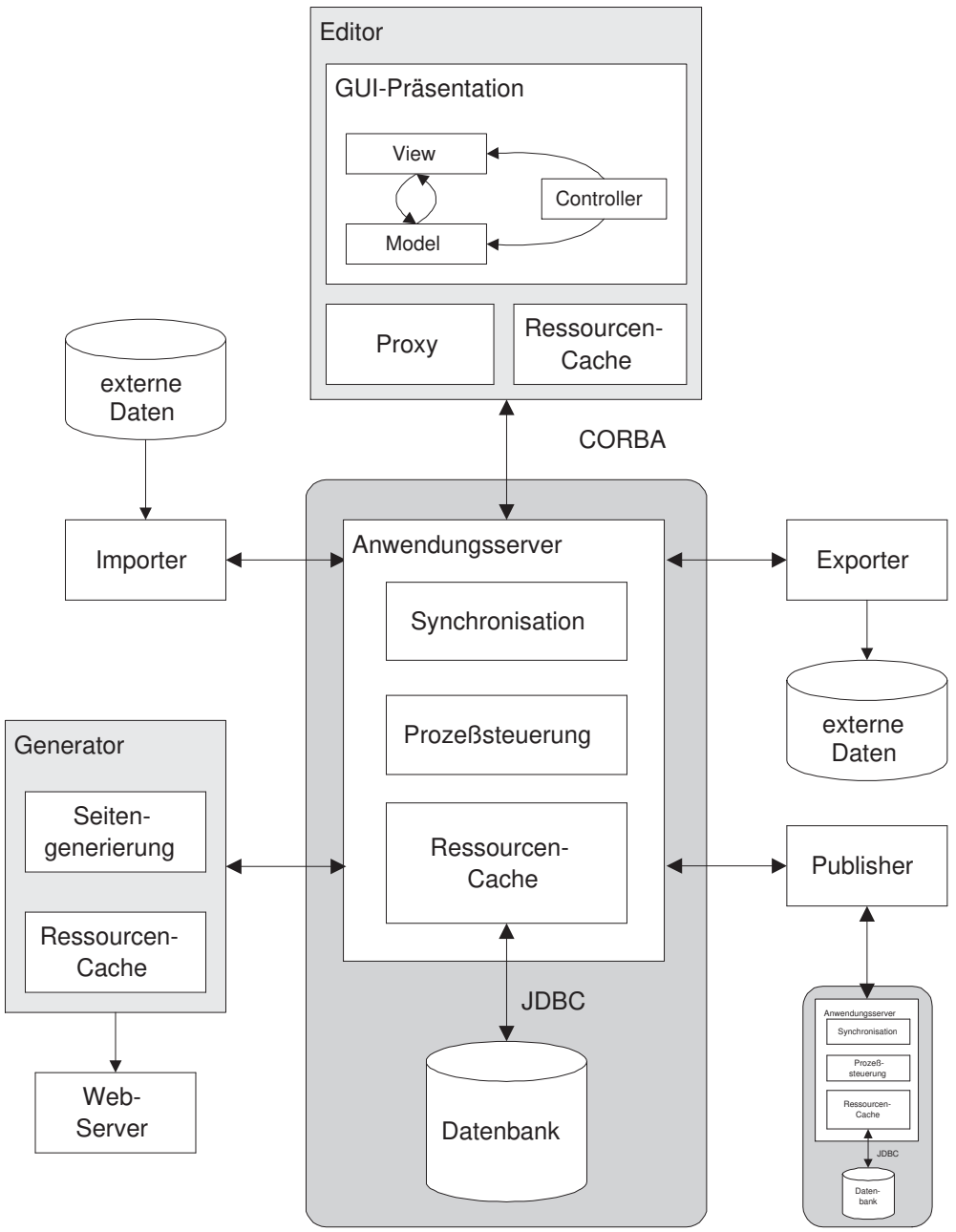


Abbildung 4.1: Systemarchitektur

einziges Mal ab, wenn sie es vom Anwendungsserver lesen oder bevor sie neue Dokumente in den Anwendungsserver schreiben, so daß sie von einem eigenen Ressourcen-Cache nicht profitieren. Die *Seitengenerierung* transformiert Dokumente in externe Visualisierungen (siehe Abschnitt 3.3.1). Zur Erhöhung der Performanz besitzt die Seitengenerierung einen eigenen Ressourcen-Cache, der die lesenden Zugriffe auf den Anwendungsserver reduzieren soll.

Ein komplettes CoreMedia System besteht aus zwei Anwendungsservern. Einer ist der sog. *Produktionsserver*, auf dem die Benutzer innerhalb des redaktionellen Arbeitsablaufs an der Verarbeitung des Dokumentenbestandes arbeiten. Der zweite Server, der sog. *Publikationsserver* stellt eine konsistente Version der Dokumentenbestände für externe Benutzer bereit. Ein *Publisher* genannter Klient gleicht die Dokumentenbestände von Produktions- und Publikationsserver ab und sorgt für einen jederzeit konsistenten Dokumentenbestand auf den Publikationsserver.

Der *Editor* stellt eine graphische Benutzungsoberfläche für die Bearbeitung und Recherche von Dokumenten zur Verfügung. Gemeinsam mit dem Anwendungsserver realisiert der Editor einen redaktionellen Prozeß, an dem alle Klienten des Servers, vor allem aber weitere Editoren beteiligt sind. Die für die Benutzungsoberfläche benutzten GUI-Komponenten des Editors arbeiten nach dem *Model-View-Controller*-Paradigma (siehe Abschnitt 4.1.2) für Benutzungsschnittstellen. Zur Entkopplung der graphischen Benutzungsoberfläche von den Dokumenten auf dem Anwendungsserver besitzt der Editor einen Ressourcen-Cache und setzt die Technik der Stellvertreterobjekte nach dem *Proxy*-Entwurfsmuster (siehe Abschnitt 4.1.3) ein. Der Ressourcen-Cache ist eine Komponente für Klienten, die die Anzahl der Zugriffe auf den Server minimiert und so die Performanz erhöht. Der *Proxy* macht den Ressourcen-Cache nutzbar für graphische Benutzungsschnittstellen, die nach dem Model-View-Controller-Paradigma arbeiten. Erst zusammen ermöglichen es beide Komponenten, zügig mit dem Editor zu arbeiten. Das Model-View-Controller-Paradigma und die Entwurfsentscheidungen, die zu der vorgestellten Architektur geführt haben, werden in den folgenden Abschnitten erläutert.

Eine der Eigenschaften eines Systems zur multimedialen Dokumentenverarbeitung ist der parallele Betrieb mehrerer Benutzer (siehe Abschnitt 2.1.1). Bei einem Online-Redaktionssystem wird dies zur Realisierung eines redaktionellen Prozesses benutzt. Während eines solchen redaktionellen Prozesses kooperieren mehrere Benutzer, die gleichzeitig mit dem System arbeiten. Da ein Online-Redaktionssystem eine graphische Benutzungsoberfläche benötigt (siehe Abschnitt 2.3) und die Benutzer gleichzeitig arbeiten sollen, kann die Bearbeitung der Dokumente nur verteilt an mehreren Rechnern erfolgen. Die Kooperation der Benutzer erfordert die Kommunikation zwischen den Programmteilen auf den Rechnern über ihren Zustand im verteilten Redaktionsprozeß. Das vorherrschende Paradigma für die Erstellung solcher verteilter Systeme ist die Client-Server-Architektur (siehe Abschnitt 2.2.4).

Eine Client-Server-Anwendung besteht aus einem oder mehreren Servern, auf die die über ein Netzwerk angeschlossenen Klienten zugreifen. Die Übertragung über das Netzwerk erzeugt notwendigerweise Latenzzeiten [Orfali et al. 96] für die Client-Server-Anwendung. Diese Latenzzeiten müssen für die Implementierung einer graphischen Benutzungsoberfläche vor dem Benutzer verborgen werden, um den Eindruck einer flüssigen Bedienung des Systems zu erreichen. Mit der Entstehung und Wirkung der Latenzen beschäftigt sich Abschnitt 4.1.2 näher.

Ein Möglichkeit, die Wirkung der durch das Netzwerk entstehenden Latenzzeiten zu reduzieren, besteht darin, eine *Replikation* der im Anwendungsserver vorliegenden Daten vorzunehmen. Die graphische Benutzungsoberfläche auf dem Klienten arbeitet dann auf dieser lokalen Kopie der Daten. Netzwerkzugriffe werden nur noch zum Aufbau dieser lokalen Kopie und zur Aktualisierung nach Änderungen benötigt. Die Technik hierfür muß die Replikation der auf dem Server vorliegenden Daten ermöglichen. Damit werden inkon-

sistente Zustände zwischen Klienten und Server vermieden und das Netzwerk möglichst gering belastet. Für eine detaillierte Diskussion der Probleme beim Aufbau verteilter Systeme wie z.B. Replikation und Partitionierung, verteilte Stati und Kausalitätsbeziehungen, Serialisierbarkeit und Nebenläufigkeit sei auf die Literatur verwiesen [Jessen, Valk 86; Waldo et al. 97; Vogel, Duddy 97; Orfali et al. 96].

### 4.1.2 Model-View-Controller

Das erste Rahmenwerk für graphische Benutzungsoberflächen stellt das *Model-View-Controller* (MVC) [Krasner, Pope 88] Anwendungsrahmenwerk von Smalltalk-80 dar. Es basiert auf der Annahme, daß Anwendungen mit graphischer Benutzungsoberfläche aus den drei Komponenten bestehen, die dem Rahmenwerk seinen Namen gaben:

- Model,
- View und
- Controller.

Diese drei Komponenten können zur Strukturierung solcher interaktiver Anwendungen genutzt werden. Sie sind ist in Abbildung 4.2 graphisch veranschaulicht. Das *Model* speichert die anwendungsspezifischen Daten, zum Beispiel die Zeichen des Textes bei einer Textverarbeitung. Eine *View*-Komponente präsentiert das Model bzw. seine Daten auf einem Anzeigegerät wie z.B. einem Bildschirm. Eine beliebige Anzahl von View-Komponenten kann das Model auf verschiedene Art und Weise präsentieren. Dabei greift jede Komponente auf dieselben Daten, die im Model gespeichert sind, zu. Der *Controller* behandelt die Eingabeereignisse wie Mausinteraktionen und Tastatureingaben.

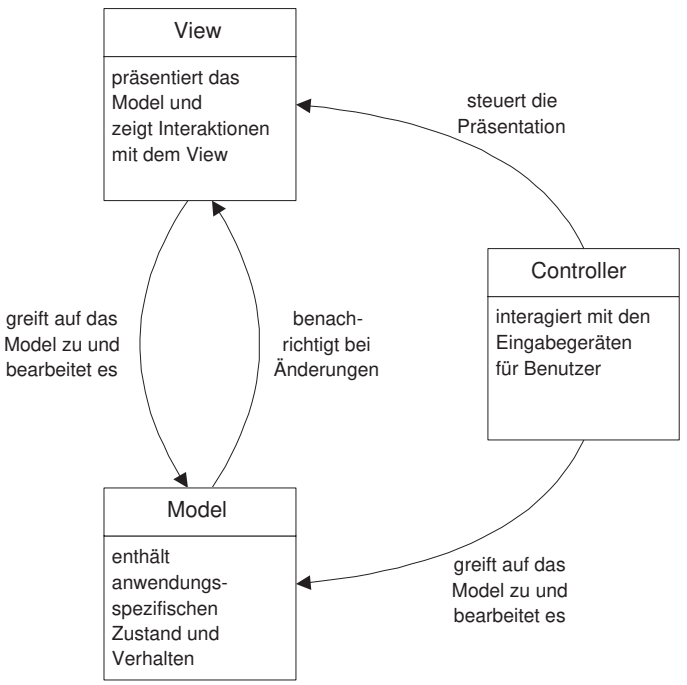


Abbildung 4.2: Model-View-Controller

Jedes View hat einen assoziierten Controller, der das View mit der Tastatur und der Maus verbindet. Daraus folgt, daß ein Model eine beliebige Anzahl von assoziierten View-Controller-Paaren haben kann, mindestens jedoch eine. View und Controller können auf das Model zugreifen. Das Model wiederum benachrichtigt das View von Änderungen.

Das Model-View-Controller-Paradigma hat einige Defizite. Dazu gehört, daß die strikte Trennung in drei Komponenten zu rigide ist und breite Schnittstellen zwischen den Komponenten erfordert. Zudem ist der einfache Benachrichtigungsmechanismus mit nur einer Nachricht zu ineffizient für viele Anwendungsfälle. Dennoch ist der Model-View-Controller-Ansatz in fast allen nachfolgenden Rahmenwerken für graphische Benutzungsoberflächen übernommen worden. Diese moderneren Implementierungen des Model-View-Controller-Ansatzes unterscheiden sich deutlich, so daß für die Realisierung der generischen Visualisierung von Dokumenten eine Entscheidung für eine konkrete Model-View-Controller-Implementierung notwendig ist. Für die vorliegende Arbeit wird dazu die Swing-Klassenbibliothek eingesetzt. Die Entscheidung für Swing wird in Abschnitt 5.1 erläutert.

Um den Defiziten des Model-View-Controller Ansatzes entgegenzutreten, sind bei Swing View und Controller verschmolzen. Die Views werden mittels fein abgestufter Nachrichten vom Model über Veränderungen benachrichtigt. Dies hat für die Implementierung des Editors weitreichende Implikationen. In Abbildung 4.3 sind mögliche Systemgrenzen für die Verteilung bei unterschiedlichen Model-View-Controller-Implementierungen abgebildet.

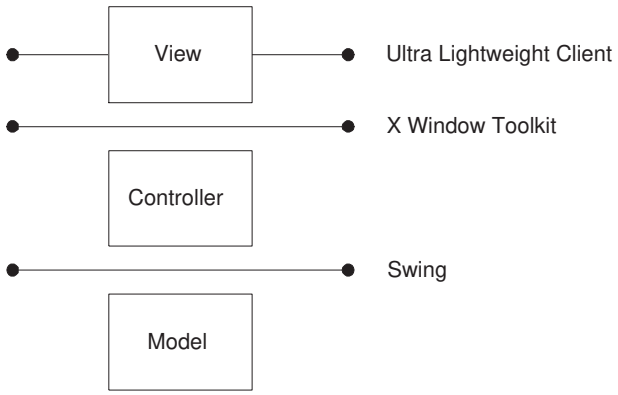


Abbildung 4.3: Systemgrenzen zur Verteilung bei Model-View-Controller

Die X Toolkit Widgets [Swick, Weissman 88] realisiert den klassischen Ansatz und legt die Systemgrenze zur Verteilung zwischen View und Controller. Der Ultra Lightweight Client [Gamma 99] geht einen neuen Weg und versucht, den Code auf der Klientenseite zu minimieren. Dazu teilt er das View in zwei Teile auf. Bei Swing ist eine Systemgrenze nur zwischen der View-Controller-Komponente auf der einen Seite und dem Model auf der anderen Seite möglich. Durch die enge Zusammenarbeit von View und Model ist die Schnittstelle zwischen diesen Komponenten sehr breit. Die strikte Trennung von Model und View macht sehr viele Methodenaufrufe nötig. Das bedeutet, daß die mögliche Trennung in ein Model auf dem Server und ein View auf dem Klienten aufgrund der Latenzzeiten des Netzwerks nicht möglich ist. Um dennoch die Swing-Komponenten benutzen zu können, arbeitet das View auf einem Stellvertreter-Model. Dieses Model wird vom Proxy zur Verfügung gestellt und repliziert den Zustand des Servers auf die Klienten. Dadurch kann das Stellvertreter-Model die Kosten für die Methodenaufrufe zum Server minimieren.

### 4.1.3 Abstraktes Verteilungsmodell

Zur Verdeutlichung der Architektur, die die Daten des Servers auf die Klienten repliziert, zeigt das Klassendiagramm in der Unified Modeling Language auf Abbildung 4.4 ein Beispiel. Die Klasse *ObjectImplementation* implementiert dazu ein Objekt auf dem Server. Dieses konkrete Objekt steht stellvertretend für *alle* Objekte, die in einer Client-Server-Anwendung zwischen Klienten und Server benutzt werden. Sollen in einer Anwendung beispielsweise Fünf Objekttypen unterstützt werden, dann ist das vorgestellte Schema fünfmal, d.h. für jeden Typ einmal, zu erstellen. Ein konkretes Objekt bietet dabei Methoden an, um seinen Zustand abzufragen und zu manipulieren. Diese Methoden stehen über die Schnittstelle *Object* den externen Klienten zur Verfügung. Daneben besitzt die Klasse *ObjectImplementation* weitere Methoden, die für die interne und vor Klienten verborgene Realisierung der durch die Schnittstelle angebotenen Leistung nötig sind.

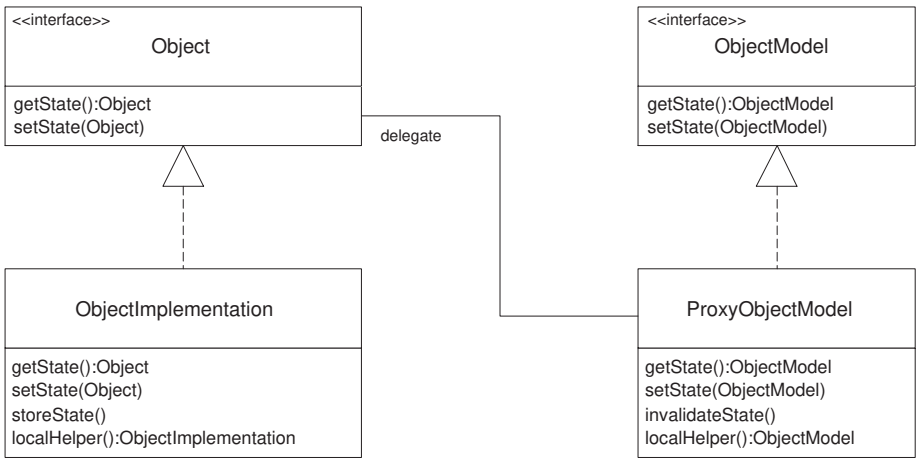


Abbildung 4.4: Klassendiagramm zur Verteilung von Objekten

Auf der Klientenseite implementiert die Klasse *ProxyObjectModel* einen Stellvertreter für das Server-*Object*. Der Präfix *Proxy* nach dem gleichnamigen Entwurfsmuster [Gamma et al. 94] deutet diese Funktionalität als Stellvertreter für das Server-Objekt an. Durch die Rolle als Stellvertreter kann ein *ProxyObjectModel* einen lokalen Zustand haben und auf diese Weise steuern, wann es sein Pendant auf der Server-Seite kontaktiert. Auf diese Weise entkoppelt das *ProxyObjectModel* die Zustände von Server und Klienten. Es hält seinen lokalen Zustand aktuell, so daß er als ein Abbild oder auch ein Modell des server-seitigen Zustandes betrachtet werden kann. Diese Rolle soll das Suffix *Model* andeuten.

Wie in Abschnitt 4.1.1 angedeutet, kann ein klientenseitiger Zustand dazu genutzt werden, die Latenzzeiten eines Netzwerks zu verbergen. Zur Aktualisierung des Klientenzustandes wird ein Mechanismus benötigt, der bei einer Zustandsänderung auf dem Server alle Kopien in den Klienten davon benachrichtigt. Dieses Entwurfsmuster heißt *Observer* oder *Publish-Subscribe* [Gamma et al. 94]. Das Klassendiagramm in Abbildung 4.5 zeigt den Benachrichtigungsmechanismus in dem vorgestellten Verteilungsmodell. Jede *ObjectImplementation* benachrichtigt alle angemeldeten *ObjectListener* bei Zustandsänderungen. Damit ein *ProxyObjectModel* seinen lokalen Zustand aktuell halten kann, registriert es einen *ProxyObjectListener* bei seinem Server-*Object*.

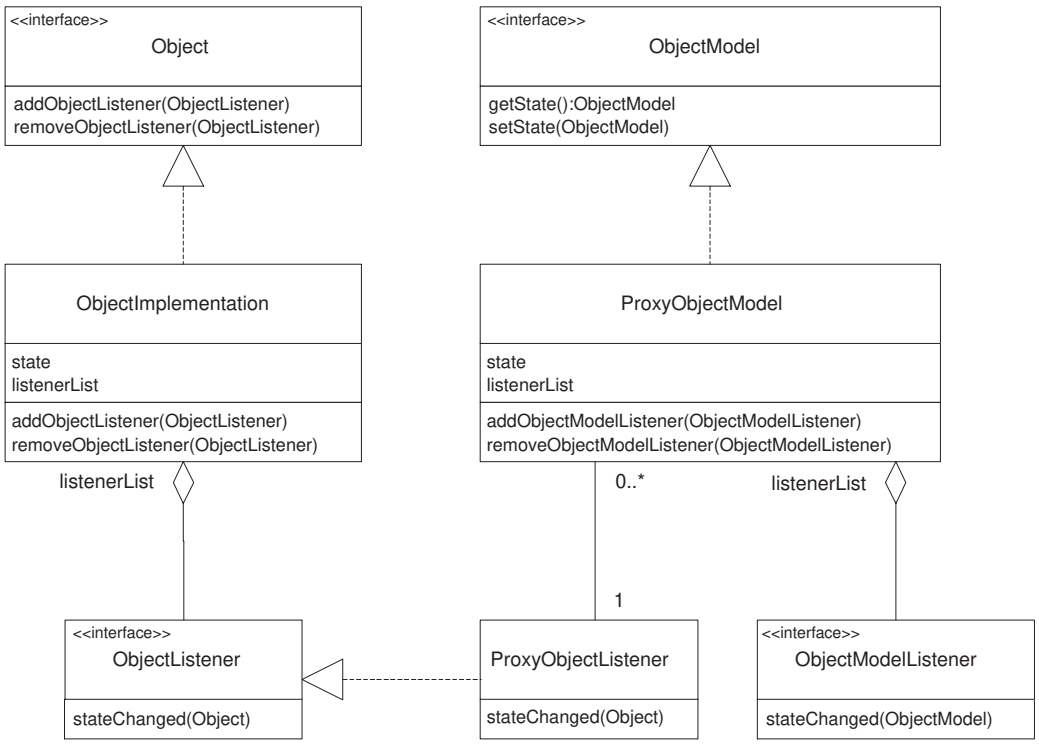


Abbildung 4.5: Klassendiagramm zur Aktualisierung der verteilten Objekte

Die Implementierung der Methoden, die über die Zustandsänderung benachrichtigen, erfüllen drei Bedingungen:

- Die von der Zustandsänderung betroffene *ObjectImplementation* geht eindeutig aus der Benachrichtigung hervor. Dann kann der *ProxyObjectListener* nach dem Entwurfsmuster eines *Flyweight* [Vogel, Duddy 97] ausgeführt werden. Es existiert dann nur ein Benachrichtigungsobjekt der Klasse *ProxyObjectListener*, das an alle *Objecte* auf dem Server angemeldet wird.
- Die Benachrichtigungen sind so fein granuliert, daß bei einer Benachrichtigung über eine Zustandsänderung auf dem Server der kleinstmögliche Teil des lokalen Zustandes eines *ProxyObjectModel* ungültig wird und bei Bedarf wieder neu von der *ObjectImplementation* angefordert wird. Dies steigert zwar den Aufwand bei der Implementierung, reduziert jedoch die Netzwerkbelastung deutlich.
- Die Daten, die bei einer Benachrichtigung mit verschickt werden, sind nur von geringem Umfang. Üblicherweise ermöglicht eine Benachrichtigung nur die eindeutige Identifikation des Objektes, das davon betroffen ist. Damit bleibt es dem *ProxyObjectModel* überlassen, wie es angemessen reagiert.

Um auch klientenseitig die Benachrichtigungen z.B. für Veränderungen der graphischen Benutzungsoberfläche nutzen zu können, bietet ein *ProxyObjectModel* denselben Mechanismus wie die *ObjectImplementation* an. So können sich *ObjectModelListener* bei ihm registrieren und von Veränderungen des lokalen Zustandes benachrichtigt werden. Die Initialisierungsphase eines *ProxyObjectModel*, das als Stellvertreter für die server-seitige *ObjectImplementation* erzeugt wird, ist im *Interaktionsdiagramm* in Abbildung 4.6 zu sehen.

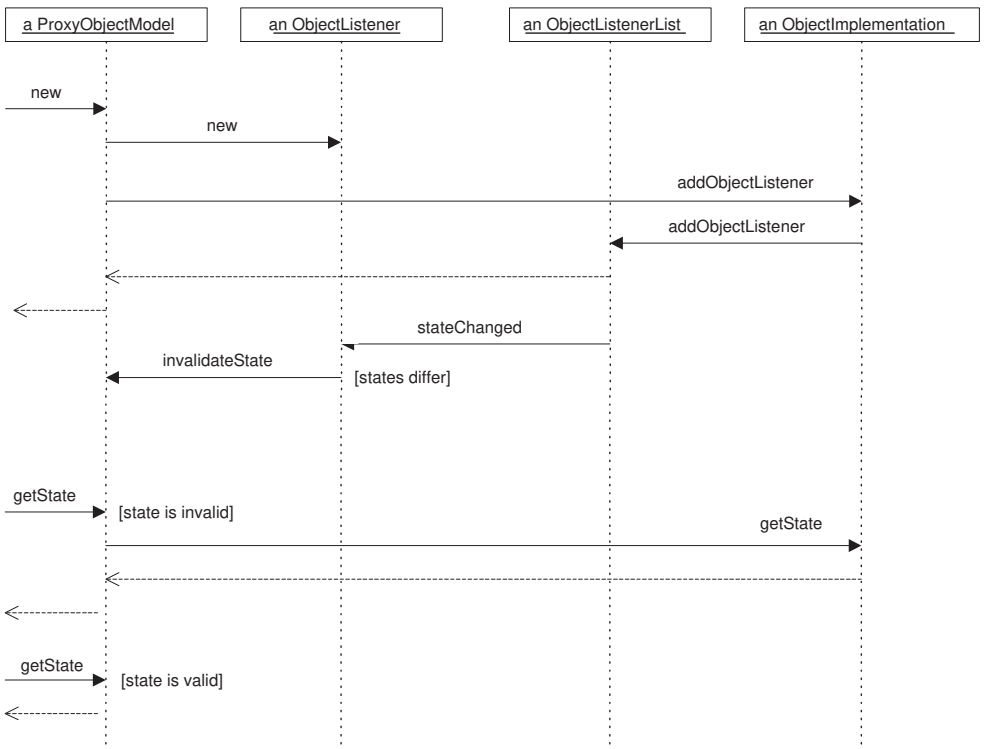


Abbildung 4.6: Interaktionsdiagramm zur Initialisierung der verteilten Zustände

Während der Initialisierung des *ProxyObjectModel* erzeugt es einen *ProxyObjectListener*, den es bei der *ObjectImplementation* auf der Server-Seite registriert. Dies ist das Verhalten, das man von einem sog. *Well behaved client*, d.h. ein Klient, der zur Kooperation mit dem Server bereit ist, erwartet. Ein ganze Reihe von Konventionen sind zum reibungslosen Zusammenspiel von Klienten und Server nötig: Beispielsweise muß ein Server-*Object* nach jeder Zustandsänderung eine Benachrichtigung schicken. Diese Benachrichtigung muß von der gerade erfolgten Zustandsänderung berichten, darf nicht vor oder nach anderen Benachrichtigungen ankommen etc.

Eine weitere Konvention dieses Verteilungsmodells besteht darin, daß die Server-Objekte nach der Registrierung eines Objekt-Observers diesem als erstes eine Benachrichtigung über eine komplette Zustandsänderung schicken. Der Vorteil liegt für den Klienten darin, daß er den kompletten Server-Zustand mit derselben Methode kopieren kann, die auch bei Veränderungen des Server-Objektes benutzt wird. Es ist kein spezieller Code für die Initialisierung nötig. Im vorliegenden Beispiel benutzt das *ProxyObjektModel* eine andere Strategie. Das Kopieren des Server-Zustandes verschiebt es auf den spätestmöglichen Zeitpunkt. Dies ist der Moment, in dem auf der Klientenseite die Daten benötigt werden. Ein Methodenaufruf der *ObjectImplementation* holt die Daten vom Server und speichert sie zwischen. Solange der lokale Zustand nicht zwischenzeitlich durch eine Benachrichtigung ungültig wird, können weitere Abfragen die lokal auf dem Klienten gespeicherten Daten benutzen.

Das Interaktionsdiagramm in Abbildung 4.7 stellt dar, wie der Abgleich der verteilten Zustände vor sich geht. Das *ProxyObjectModel* eines Klienten erhält den Methodenaufruf, den Zustand zu ändern. Dieser Aufruf wird an die *ObjectImplementation* auf der Server-Seite weitergereicht. Dort wird der neue Zustand gespeichert und eine Benachrichtigung an alle registrierten *ObjectListener* versandt. Zur weiteren Entkopplung geschieht dies asyn-

chron zum weiteren Programmablauf. Damit kann der Klient nach dem Aufruf des Servers weiterarbeiten und die Benachrichtigung über die Zustandsänderung trifft unabhängig davon ein. Der Server wiederum kann weitere Aufrufe von Klienten behandeln, während die Benachrichtigungen verschickt werden. So wirkt die Abarbeitung der Methodenaufrufe aus der Sicht des aufgerufenen Servers schneller. Genauso wirkt die Reaktion darauf aus der Sicht des aufrufenen Klienten beschleunigt.

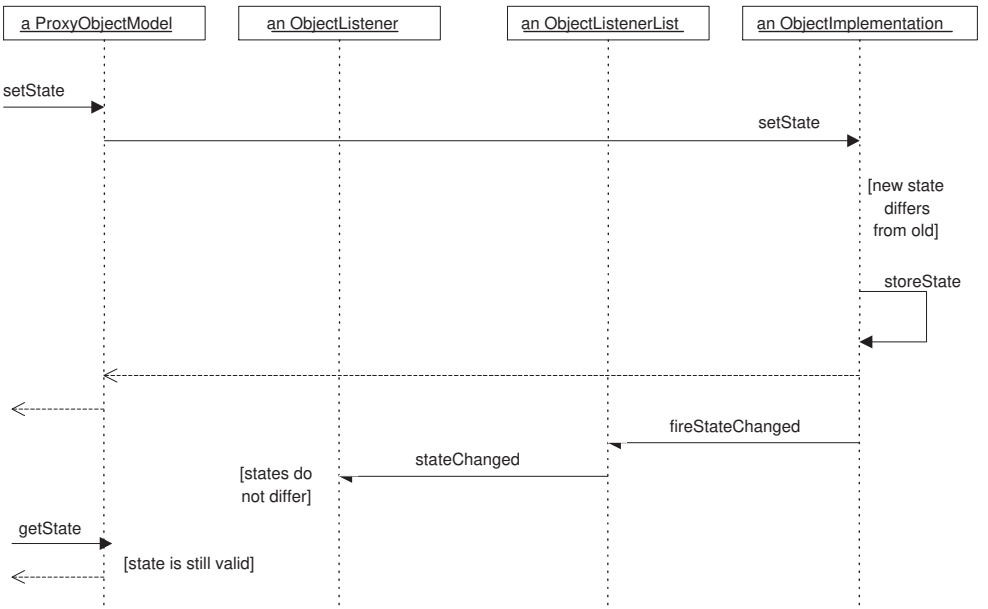


Abbildung 4.7: Interaktionsdiagramm zum Abgleich der verteilten Zustände

Der Abgleich der verteilten Zustände in Abbildung 4.7 zeigt die Interaktion für den Klienten, der die Server-*ObjectImplementation* modifiziert hat. Da damit der Zustand des Server-Objektes dem der Klientenkopie entspricht, wird der lokale Zustand des *ProxyObjectModel* durch die Benachrichtigung nicht ungültig. Jedoch müssen alle anderen Klienten ihren lokalen Zustand verwerfen und bei Bedarf wieder neu anfordern. Dies geschieht genau so wie nach der Initialisierungsphase in Abbildung 4.6.

Zusammengefaßt ist das vorgestellte *ProxyObjectModel* das Model nach dem Model-View-Controller-Paradigma (siehe Abschnitt 4.1.2) für Views der Swing-Klassenbibliothek. Die bei Swing verschmolzenen View- und Controller-Komponenten heißen dabei der Einfachheit halber wieder View. Das CoreMedia Bearbeitungswerkzeug benutzt diese verschmolzenenView-Komponenten auf Basis der Stellvertreter-Models für die Visualisierung und Bearbeitung der Dokumente.

## 4.2 Architektur eines Systems zur generischen Visualisierung

Die im folgenden vorgestellte Architektur für die generische Visualisierung multimedialer Dokumente erlaubt eine Menge von Lösungsansätzen. Den Kern dieser Lösungsansätze bilden die durch die Architektur vorgegebenen Eckpunkte. So spannt die Architektur als technischer Kontext einen Rahmen auf für die Lösungen zur generischen Visualisierung, die im Folgenden vorgestellt werden.

Zu den Eckpunkten gehört das Typsystem von CoreMedia mit seinen Dokument- und Attributtypen. Es zeigt sich, daß alle Lösungen auf die Benutzung dieser Typen zurückgreifen müssen, um die Dokumente des Systems zu visualisieren. Sie benutzen implizit auf ganz unterschiedliche Art und Weise festgelegte Verarbeitungsregeln, die versuchen, aus den Typen eine graphische Benutzungsoberfläche abzuleiten.

Die Ebene und der Zweck der Verarbeitungsregeln variiert in unterschiedlichem Maße. So nutzen alle Lösungen eine Abbildung von Typen auf Sichten, auch wenn dies nicht zwingend ist. Die Systemarchitektur von CoreMedia begünstigt solche Lösungen, die gesondert als Aggregationshierarchien des Systems (siehe Abschnitt 4.2.2) betrachtet werden.

Darauf aufbauend folgt eine Einordnung der Lösungsalternativen mittels einer Klassifikation von Techniken der Software-Erstellung und einem Schema für die Ablaufzeiten eines Programms. Daraus wird ein Schema für die Einteilung der Lösungen entwickelt. Am Ende steht die Eingrenzung der Lösungsalternativen auf eine zu realisierende Lösung.

### 4.2.1 Verarbeitungsregeln

Für die generische Visualisierung der Dokumente bietet es Vorteile, auf die Definition der Typen des CoreMedia Online-Redaktionssystems zurückzugreifen. So lassen sich Regeln aufstellen, die für eine Menge von Werten anwendbar sind. Da jedem Dokument ein Dokumenttyp zugeordnet ist (siehe Abbildung 4.8) und nur Dokumente von den definierten Dokumenttypen erzeugt werden können, steht die Anzahl der unterschiedlichen Dokumenttypen fest. Dieses Wissen kann für die Speicherung von Informationen über die graphische Struktur und damit auch für die Visualisierung genutzt werden. So läßt sich beispielsweise eine Abbildung von Dokumenttypen auf Komponenten der graphischen Benutzungsoberfläche realisieren.

Abbildung 4.8: Beziehung von Dokumenttyp und Dokument

Die logische Struktur der Dokumente in CoreMedia ist geprägt durch die Attributtypen. Ein Dokumenttyp wird durch eine feste Anzahl von Attributen beschrieben (siehe Abbildung 4.9).

Abbildung 4.9: Beziehung von Attributtyp und Dokumenttyp

Die Attribute besitzen einen eindeutigen Namen und einen Attributtypen. Der Name des Attributs kann sowohl für die Visualisierung als auch für eine Nachbearbeitung verwendet werden. Für die Aufstellung von Regeln, die bestimmen, welche graphischen Strukturen für ein Attribut zur Verfügung stehen, können die Attributtypen herangezogen werden. Die Regeln bestimmen, wie ein Element der logischen Struktur durch ein graphisches Element in Abhängigkeit von seinem Typ visualisiert wird.

Weiter lassen sich Eigenschaften des graphischen Elements wie

- Position,
- anfängliche Größe,
- Wachstumsverhalten bei größerem Platzangebot,
- Plazierung bei größerem Platzangebot und
- Schrumpfverhalten bei kleinerem Platzangebot

aus dem Attributtyp bestimmen. Da die graphischen Elemente unterschiedliche Eigenschaften haben, können weitere Regeln definiert werden, die bestimmen, wo ein graphisches Element plaziert wird.

Diese Eigenschaften der graphischen Elemente beschäftigen sich mit dem Bestandteil der Formatierung (siehe Abschnitt 3.1.2) eines Dokuments. Die dazu zu realisierende Abbildung der logischen Struktur des Dokuments auf eine graphische Struktur wurde in den Grundlagen zur generischen Visualisierung diskutiert. Der Kontext erfordert, daß die Prinzipien für die Gestaltung der Schnittstelle zum Benutzer, die in Abschnitt 2.3 erläutert wurden, beachtet werden.

### 4.2.2 Aggregationshierarchie durch Typen

Die Verarbeitungsregeln legen nahe, das Typsystem und die Werte, die es beschreibt, sowie seine Verbindungen zu den Komponenten, die die Werte in einer graphischen Benutzungsoberfläche visualisieren, aus einem anderen Blickwinkel zu betrachten. Abbildung 4.10 versucht dies mit Hilfe einer Aggregationshierarchie.

Jeweils grau unterlegt sind die drei Ebenen, auf denen Dokumente und ihre Attribute aggregiert werden. Horizontal ist die Menge der Werte eines Attributs zu sehen, die durch einen Typen, den Attributtypen, beschrieben werden. Diese Typdefinition erlaubt es, ein Model zu entwerfen, das diese Werte aufnehmen und mittels eines View darstellen kann. Auf dieser Basis lassen sich dann die Verarbeitungsregeln wie in Abschnitt 4.2.1 aufstellen.

Auf der Ebene der Dokumente aggregiert ein Dokument über eine Menge von Attributen. Wieder erlaubt der Dokumenttyp die Dokumentwerte, also die Menge der Dokumente, zu beschreiben. Und wieder lassen sich daraus ein Model und ein View erstellen. Auf der höchsten Ebene aggregiert der Dokumentenspeicher des Systems über die Menge aller Dokumente, die in ihm gespeichert sind.

### 4.2.3 Klassifikation der Alternativen

Die vorgestellte Architektur bildet den Rahmen für eine Menge von Lösungsansätzen zur generischen Visualisierung. Für die Einordnung wird ein Schema benutzt, um die Lösungsansätze zu klassifizieren. Das Schema stützt sich auf eine Klassifikation von Techniken der Software-Erstellung von Scheer und ein Schema für die Ablaufzeiten von Programmen nach Watt. Aus einer Kombination wird das Schema zur Einteilung der Lösungsansätze entwickelt. Die Lösungsansätze werden nach Kategorien eingeteilt und ins Schema eingeordnet. Eine Eingrenzung des Lösungsraumes auf eine zu realisierende Lösung bildet den Abschluß.

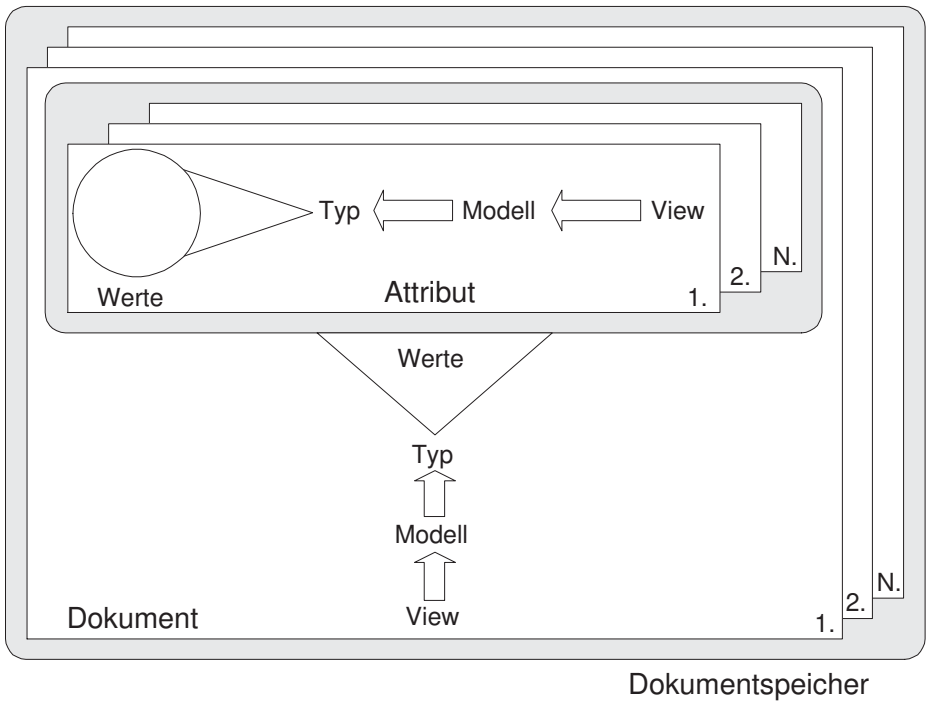


Abbildung 4.10: Aggregationshierarchie durch Typen

**Techniken der Software-Erstellung**

Nach Scheer [Scheer 97] gibt es vier Arten der Software-Erstellung, die auch auf das Teilgebiet der Erstellung einer graphischen Benutzungsoberfläche übertragen werden können. Dies sind:

- Programmierung,
- Konfiguration,
- Modularisierung und
- Generierung.

Unter *Programmierung* ist das Kodieren von Quelltext in einer Programmiersprache zu verstehen. Das Ergebnis ist ein Programm, das eine graphische Benutzungsoberfläche erzeugt.

Bei der *Konfiguration* wird die Benutzungsoberfläche eines Programmes durch die Festlegung von Konfigurationsdaten angepaßt. Diese müssen durch das Programm eingelesen und ausgewertet werden, um eine entsprechend angepaßte graphische Benutzungsoberfläche anzuzeigen.

Die *Modularisierung* ist eine Technik, die es erlaubt, Programmteile in Einheiten, sog. *Module*, zu zerlegen. Diese können bei der Programmierung als Bausteine oder Komponenten in ein Programm integriert werden. Alternativ können Module durch Konfiguration in ein Programme eingebunden werden. Die Modularisierung erlaubt es damit, eine komplexe graphische Benutzungsoberfläche aus kleinen, einfacheren Elementen zusammenzustellen.

Unter *Generierung* ist das maschinelle Erzeugen von Code zu verstehen. Dieser Code kann als Vorlage zum Programmieren dienen. Die Generierung unterscheidet sich dann konzeptionell nicht von der Konfiguration. Alternativ kann ein Programm direkt die Präsentation einer graphischen Benutzungsoberfläche generieren, ohne den Zwischenschritt über Quelltext zu nehmen.

Hybride Ansätze kombinieren zwei oder mehr Arten der Software-Erstellung. So ist z.B. der Ansatz weit verbreitet, eine generische Lösung zu programmieren und die Umsetzung konkreter Anforderungen über die Konfiguration zu erledigen. Zur Vereinfachung sind in dieser Arbeit die Lösungsansätze anhand der dominierenden Art der Software-Erstellung eingeordnet.

### Ausführungszeiten von Programmen

Bei der Ausführung der durch Programmierung, Konfiguration, Modularisierung oder Generierung erstellten Programme nach Watt [Watt 90] unterscheidet man

- die Übersetzungszeit (*compile time*),
- die Ladezeit (*load time*),
- die Bindezeit (*link time*) und
- die Laufzeit (*run time*).

Die *Übersetzungszeit* liegt vor dem Programmstart. Bei der Übersetzung wird Quelltext durch Übersetzer (*Compiler*) in einen Zielcode übersetzt. Bezogen auf die Erstellung graphischer Benutzungsoberflächen läuft die Übersetzung nach der Programmierung, um ausführbaren Maschinencode zu erhalten.

Zur *Ladezeit* eines Programms wird der Maschinencode des Programms geladen. In modernen Betriebssystemen wird die Ladezeit auf den spätest möglichen Zeitpunkt verschoben. Programmteile oder Module werden erst geladen, wenn das Programm sie benötigt. Der Zeitpunkt der Ausführung unterscheidet sich so je nach Programmteil. Während für einige schon die Laufzeit erreicht ist, befinden sich andere Programmteile in der Lade- und Bindezeit oder werden erst übersetzt, bevor sie geladen werden. Zur Ladezeit werden auch die Konfigurationsdaten für Benutzungsoberflächen geladen.

Während der *Bindezeit* werden Verweise von gerade geladenem Programmcode zu anderem Programmcode aufgelöst und der Programmcode so ablauffähig gemacht. Auf diese Weise können Module hinzugeladen werden, um eine Benutzungsoberfläche zu erweitern oder zu erstellen.

Von der *Laufzeit* eines Programmes spricht man, wenn dieses ausgeführt wird. Bei *Just in time*-Übersetzern wird der zur Übersetzungszeit aus dem Quelltext generierte Zielcode in Maschinencode übersetzt. Somit stellt der Zielcode nur einen Zwischencode bis zur Ausführung des Programmes da. Ebenso kann der Code ausgeführt oder interpretiert werden, um die Benutzungsoberfläche zu erstellen. Dabei werden die Komponenten der Benutzungsoberfläche erzeugt, mit Konfigurationsdaten initialisiert und sichtbar gemacht.

Die Lade- und die Bindezeit sind für die Erstellung von Benutzungsoberflächen nicht relevant. Die dafür notwendigen Funktionen reichen zwischen Übersetzungs- und Laufzeit soweit zusammen, daß die Aufgaben, die während der Lade- und Bindezeit erledigt werden könnten, zu Beginn der Laufzeit stattfinden können. Die Ladezeit bezogen auf einen Codeteil ist die Zeit, während das Codeteil geladen wird und bevor es zur Ausführung gebracht wird. Sobald das Codeteil ausgeführt wird, beginnt die Laufzeit. Zwischen den Ausführungszeiten verschiedener Codeteile muß unterschieden werden, da sich das Nachladen von Codeteilen durchaus mit der Laufzeit anderer Codeteile überschneiden kann.

Die Erstellung von Benutzungsoberflächen mit generischen Elementen ist nicht durch den einmaligen Aufruf zur Ladezeit möglich. Verlagert man dagegen diesen Abschnitt der Programme auf die Laufzeit, wird zur Ladezeit lediglich Code geladen und die Bindezeit ist nicht von Bedeutung. Zur Laufzeit dagegen kann die Benutzungsoberfläche dynamisch angepaßt werden. Während der Übersetzungs- und der Laufzeit wird das gesamte Spektrum der Funktionalitäten, die für Benutzungsoberflächen notwendig sind, abgedeckt.

### 4.2.4 Schema für die Einteilung

Das Schema mit den Arten der Software-Erstellung auf der einen Seite und den Ausführungszeitpunkten von Programmen auf der anderen Seite ist für eine Einteilung von Software-Architekturen hilfreich. Die vorzustellenden Entwürfe sind in Tabelle 4.1 in das vorgestellte Schema eingeordnet.

| | Programmierung | Konfiguration | Modularisierung | Generierung |
|---|---|---|---|---|
| Übersetzungszeit | direkte Kodierung | Nutzung von Standardwerkzeugen | Entwurf eines Rahmenwerkes | Generierung von Quelltext |
| Laufzeit | Übersetzung und Einbindung von Quelltext zur Laufzeit | Generierung mit Nachbearbeitung | | Layout-Generierung |

Tabelle 4.1: Einordung der Entwürfe

Wie sich zeigt, paßt das Schema für die vorgestellten Entwürfe fast. Die vorgestellten Lösungen überdecken die Übersetzungs- und die Laufzeit mit Ausnahme der Modularisierung zur Laufzeit. Ein Lösung für dieses Lösungsfeld ist nicht möglich, da die Modularisierung zur Laufzeit unmöglich ist, da sie zur Übersetzungszeit stattfinden muß. Zur Laufzeit ist nur eine Konfiguration von modularisiertem Code möglich (siehe Abschnitt 4.2.3).

Für die Beschreibung der Lösungsansätze dient eine Klassifikation der Übersicht. Die in Tabelle 4.1 in das Schema eingeordneten Entwürfe werden deshalb in den nächsten Abschnitten in den Kategorien

- Kodierung,
- Generierung und
- Generierung mit Nachbearbeitung

beschrieben. Diese Kategorien und ihre Namen sind durch die Techniken der Software-Erstellung motiviert. Der Aspekt der Ablaufzeit zeigt sich für die generische Visualisierung als wenig aussagekräftig, um zur Kategorisierung zu dienen.

## 4.3 Kodierung

Die unter dem Begriff Kodierung gefaßten Entwürfe zur Visualisierung von Dokumenten beruhen auf der Programmierung, Konfiguration und Modularisierung einer graphischen Benutzungsoberfläche. Der Zeitpunkt für die Kodierung von Quelltext liegt vor der Übersetzungszeit. Eine Benutzungsoberfläche wird durch Programmbefehle, Konfigurationsanweisungen oder mit Hilfe eines Werkzeuges erstellt. Die Kodierung kann aber auch per Hand stattfinden.

Die in diesem Abschnitt vorgestellten Entwürfe arbeiten auf der Ebene einer Programmiersprache und haben als Zielgruppe den Programmierer. Sie haben damit die Mächtigkeit und Komplexität der Programmierung, aber auch deren Fehleranfälligkeit und Aufwand gemeinsam.

### 4.3.1 Direkte Kodierung

Die direkte Kodierung ist der klassische Ansatz, nach dem Software-Projekte realisiert werden: Ausgehend von einer Aufgabenstellung wird eine Spezifikation für ein Programm erarbeitet, die implementiert, getestet und dann eingesetzt wird. In dieser Arbeit steht nicht die Vorgehensweise bei der Software-Entwicklung im Mittelpunkt. Es ist gleich, ob die Projektabwicklung von einem idealisierten, linearen Ansatz wie beim Wasserfallmodell [Denert 91] oder von einem evolutionären, zyklischen Ansatz wie beim Spiralmodell [Boehm 88] ausgeht. Davon wird in dieser Arbeit abstrahiert.

Die Analogie von der direkten Kodierung von graphischen Benutzungsoberflächen zu der Realisierung von Software-Projekten bezieht sich auf das recht konkrete, projektbezogene Herangehen. Es geht um eine vollständige Neuimplementierung. Fast alles muß von Grund auf neu erstellt und getestet werden, da keine Prototypen oder bestehender Code existieren, auf denen aufgesetzt werden kann. Der Code muß von Hand ausprogrammiert werden.

Für die Implementierung bedeutet dies, daß keine für die Aufgabenstellung unnötigen Abstraktionen durchgeführt werden. Der Zeitaufwand und die Fehlerträchtigkeit eines umfangreichen Projektes erzwingen dies. Die Arbeit wird mit eindeutigem Fokus auf die konkrete Aufgabenstellung hin durchgeführt. Sollte ein ähnliches Problem bei weiteren Aufgabenstellungen auftreten, ist die Implementierung weitestgehend neu zu realisieren.

Um den Aufwand bei der Realisierung zu reduzieren, ist die Einbindung von fremden, oftmals zugekauften *Funktionsbibliotheken* wie MacApp, die X Toolkit Widgets oder das Andrew Toolkit sinnvoll. Bei objektorientierter Programmierung bilden üblicherweise eine oder mehrere *Klassenbibliotheken* wie InterViews, ET++ oder Swing die Basis für die Kodierung (siehe Abschnitt 5.1). Diese Klassenbibliotheken bieten konfigurier- und erweiterbare Komponenten an, die die Erstellung einer graphischen Benutzungsoberfläche erleichtern. Dadurch wird das Abstraktionsniveau bei der Programmierung von der Realisierung einer komplett neu entwickelten Oberfläche auf das Konfigurieren und Zusammenstellen fremder Komponenten angehoben. Trotz dieser Erleichterungen bleibt die direkte Kodierung zeitaufwendig und fehlerträchtig.

### 4.3.2 Entwurf eines Rahmenwerks

Der Begriff des *Rahmenwerks* (*Framework*) hat in der objektorientierten Software-Entwicklung eine große Bedeutung gewonnen. Eine ausführliche Diskussionen über Rahmenwerke finden sich im Buch von Lewis [Lewis 95]. Dort wird ein Rahmenwerk definiert als:

> „Ein Rahmenwerk ist eine erweiterbare Bibliothek von kooperierenden Klassen, die eine wiederbenutzbare Entwurfslösung für eine gegebene Problemdomäne bilden."

Diese Beschreibung läßt sich auf die Erstellung graphischer Benutzungsoberflächen übertragen. Der Entwurf eines Rahmenwerks ist ein Ansatz, der die Programmierung mit der Modularisierung verbindet. Statt für jede Problemstellung eine neue Problemlösung zu erfinden, soll das Rahmenwerk eine Problemklasse einmal, d.h.generisch, lösen und danach flexibel zur Verfügung stellen. Hier liegt der Unterschied zu Klassenbibliotheken und

Toolkits, die lediglich Komponenten zur Verfügung stellen, aber keine bewährten Problemlösungen zu häufiger auftauchenden Problemstellungen bieten. Mit dem lösungsorientierten Ansatz versucht man beim Entwurf und ber der Implementierung eines Rahmenwerks den beschriebenen Nachteilen der direkten Kodierung entgegenzuwirken. Die Erstellung eines Rahmenwerks ist dementsprechend aufwendig und lohnt sich nur, wenn es wiederverwendet wird. Das Problem bei der Erstellung eines solchen Rahmenwerks ist es, das für die Aufgabenstellung angemessene Abstraktionsniveau zu finden. Im Allgemeinen sind dazu mehrere Iterationen eines Software-Entwicklungszyklus' wie dem *Autor-Kritiker-Zyklus* (siehe Abbildung 4.11) nötig.

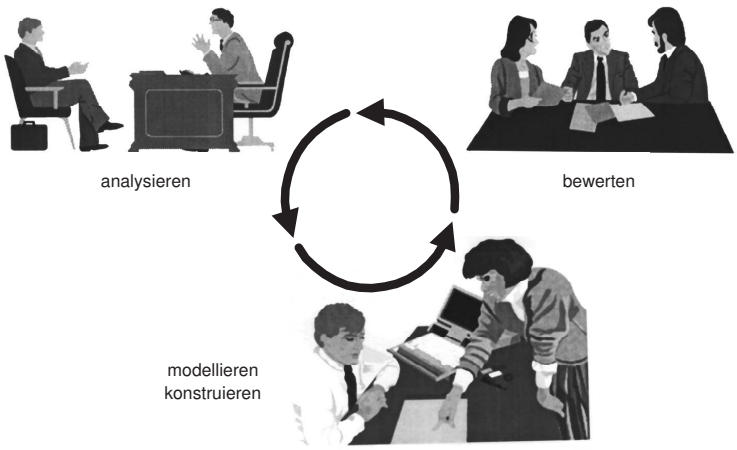


Abbildung 4.11: Autor-Kritiker-Zyklus

Ist das Rahmenwerk zu allgemein gefaßt, bedeutet dies eine größere Flexibilität im Einsatzbereich des Rahmenwerks. Ein solches Rahmenwerk erleichtert die Kodierung, doch der Aufwand bleibt aufgrund des für die Aufgabenstellung zu niedrigen Abstraktionsniveaus hoch. Dafür ist der Lösungsraum wenig eingeschränkt. Es herrscht nahezu völlige Freiheit bei den durch das Rahmenwerk unterstützbaren Anwendungsbereichen. Im umgekehrten Fall, wenn das Rahmenwerk zu speziell ist, kehren sich die Vorzeichen um. Eine Lösung, die bei der Erstellung eines speziellen Rahmenwerks mit hohem Abstraktionsniveau intendiert war, läßt sich sehr schnell kodieren. Dafür bedingt die geringe Flexibilität eines solchen Ansatzes einen großen Aufwand bei der Erstellung abweichender Lösungen. Es besteht dann leicht die Gefahr, dem Rahmenwerk zuwider zu laufen.

Das Ziel der Konstruktion eines Rahmenwerkes ist es, Konzepte und Entwürfe wiederzuverwenden. Es handelt sich von der Art der Software-Erstellung um eine Kombination der Techniken zur Modularisierung und Programmierung. Nach der Art der Wiederverwendung werden Rahmenwerke in *Blackbox*- und *Whitebox*-Rahmenwerke [Sean Cotter 95] unterschieden. Blackbox-Rahmenwerke zeichnen sich dadurch aus, daß ihre Klassen durch die Benutzung (*use*) wiederverwendet werden. Die Anpassung an die Bedürfnisse des Benutzers geschieht durch Parametrisierung. Whitebox-Rahmenwerke hingegen stellen Klassen zur Verfügung, die durch Vererbung (*inheritance*) und gezieltes Überschreiben von Methoden wiederverwendet werden. Die Verwendung eines Blackbox-Rahmenwerkes ist einfacher, da es kaum Verständnis über die innere Struktur erfordert. Ein Whitebox-Rahmenwerk dagegen kann flexibler an neue Anforderungen angepaßt werden, da es seine innere Struktur offen legt.

Bei der Erstellung eines Rahmenwerks gilt es, so flexibel, wie für die zukünftigen Aufgabenstellungen nötig zu bleiben. Gleichzeitig soll ein möglichst hohes Abstraktionsniveau erreicht werden, um den Aufwand für Projekte, die das Rahmenwerk benutzen, nied-

rig zu halten. Eine Möglichkeit, diese widersprüchlichen Ziele aufzulösen, besteht darin, ein Rahmenwerk zu konzipieren, das mehrere Abstraktionsstufen hat. Abhängig von der Aufgabenstellung kann dann die Abstraktionsstufe gewählt werden, mit der sich die Anforderungen noch erfüllen lassen, ohne daß die Grenzen der Rahmenwerkes überschritten werden. Für Aufgabenstellungen, die deutlich außerhalb des durch das Rahmenwerk vorgegebenen Lösungsraums liegen, ist eine Lösung selber zu erstellen. Alternativ kann das Rahmenwerk für die neue Aufgabenstellung erweitert werden.

Die Flexibilität bei der Anpassung eines selbst erstellten Rahmenwerks ist der entscheidende Vorteil, der für ein eigenes Rahmenwerk spricht. Dem gegenüber steht der Aufwand für die Erstellung und Pflege des Rahmenwerks. Eine Alternative besteht in der Nutzung eines von Dritten entwickelten Rahmenwerks für den anvisierten Anwendungsbereich. Das auf dem erwähnten InterViews aufbauende Unidraw [Vlissides, Linton 90] ist solch ein Rahmenwerk für den Bau von domänenspezifischen graphischen Bearbeitungswerkzeugen. Prinzipiell kann ein externes Rahmenwerk die Anforderungen nicht vollständig abdecken. Es muß ein breites Spektrum an Anforderungen abdecken, ohne die einzelnen Anforderungen im Detail zu kennen. Dagegen kann ein selbstentwickeltes Rahmenwerk die konkreten Anforderungen der eigenen Arbeit besser abbilden

Ein selbst erstelltes Rahmenwerk sollte auf einer Klassenbibliothek basieren, die erweiter- und konfigurierbare Komponenten für die Erstellung einer Benutzungsoberfläche bietet. Damit können die Anforderungen an eine moderne Benutzungsoberfläche abgedeckt und an die Klassenbibliothek delegiert werden. Das selbst erstellte Rahmenwerk befaßt sich mit den Objekten aus der eigenen Anwendungswelt und bildet deren Geschäftslogik ab. Es ist damit ein domänenspezifisches Rahmenwerk. Damit unterstützt es für eine Domäne die Entwicklung von Geschäftsanwendungen und die Präsentation der Geschäftsobjekte mit Hilfe einer Benutzungsoberfläche.

### 4.3.3 Nutzung von Standardwerkzeugen

Die Nutzung von Standardwerkzeugen ist ein Ansatz für die Visualisierung von Dokumenten, der auf der Konfiguration generischer Komponenten beruht. Dazu werden Bearbeitungswerkzeuge für die Erstellung graphischer Benutzungsoberflächen (*GUI-Editor*) eingebunden. Solche Bearbeitungswerkzeuge sind Bestandteil einer integrierten Entwicklungsumgebung (IDE) wie JBuilder [Inprise Corp. 99], Java Workshop [Sun Microsystems Inc. 99c] oder NetBeans Developer [NetBeans Inc. 00]. Ein solche Entwicklungsumgebung unterstützt den Software-Entwicklungsprozeß während der Phase der Implementierung durch eine graphische Benutzungsoberfläche. Hinter einer IDE steht die Idee einer beschleunigten Entwicklung von Anwendungen (*Rapid Application Development*).

Der Funktionsumfang einer IDE umfaßt typischerweise

- eine Projektverwaltung,
- einen Übersetzer zum Erzeugen von Maschinencode,
- eine Bibliothek mit häufig genutzten Algorithmen und Datenstrukturen,
- einen *Debugger* zur Fehleranalyse,
- einen Texteditor für den Quelltext,
- einen *GUI-Editor* für die Erstellung von Benutzungsoberflächen und
- eine Sammlung von Komponenten zur Unterstützung des GUI-Editors.

Für die Erstellung von Benutzungsoberflächen ist davon ausschließlich der GUI-Editor vom Bedeutung. Er bietet den Komfort, der bei der Erstellung von Benutzungsoberflächen per direkter Kodierung oder mit Hilfe eines eigenen Rahmenwerks fehlt. Ein weiterer Vorteil eines GUI-Editors liegt darin, daß die Benutzungsoberfläche bereits während der Erstellung visualisiert wird. Dieses *What You See Is What You Get* (WYSIWYG) Verfahren gewährleistet hier, daß die Benutzungsoberfläche während der Erstellung genauso aussieht, wie später zur Laufzeit des Programms. Der Zeitaufwand ist deutlich niedriger als beim direkten Kodieren, da das ständige Kontrollieren der Änderungen entfällt.

Ein GUI-Editor aus einem Standardwerkzeug hat einige wesentliche Nachteile: Die Nutzung einer bestimmten IDE *nur* um des GUI-Editors willen ist in den meisten Fällen nicht tragbar. Der Markt für IDEs ist unübersichtlich. Von der Vielzahl verfügbarer Entwicklungswerkzeuge sind die meisten ausschließlich für die schnelle Anwendungsentwicklung konzipiert. Die langfristige, evolutionäre Software-Entwicklung in Arbeitsgruppen wird selten unterstützt. Doch genau dort findet man eine heterogene Zusammenstellung von Entwicklungswerkzeugen. Diese können nur eingesetzt werden, wenn sie die Forderung erfüllen, daß sie keine Bindung an ein bestimmtes Entwicklungswerkzeug erfordern. Der kleinste gemeinsame Nenner ist hier der Quelltext.

Bei der Nutzung eines GUI-Editors für die Erstellung einer graphischen Benutzungsoberfläche gibt es keinen kleinsten gemeinsamen Nenner: Eine allgemeine, standardisierte Beschreibung einer graphischen Benutzungsoberfläche existiert ebensowenig wie Ansätze zur Lösung dieser Problematik [Lippert 97]. Die Hersteller der GUI-Editoren müssen diese Lücke schließen, wenn sie den Zustand der im GUI-Editor erstellten Benutzungsoberfläche persistent machen wollen. Das verwendete Format für die Speicherung kann immer nur von dem GUI-Editor gelesen werden, der es geschrieben hat.

Wenn die GUI-Editoren während der Konstruktion der Benutzungsoberfläche dieselben Strukturen wie zur Laufzeit aufbauen, kann eine spezielle GUI-Komponente versuchen, diese Strukturen zu serialisieren und abzuspeichern. Zur Laufzeit des selbst entwickelten Programms werden dann die serialisierten Strukturen wieder geladen und eine Benutzungsoberfläche daraus erstellt. Sofern eine solche Lösung realisiert werden kann, ist es sinnvoll, sie durch ein Rahmenwerk zu kapseln, um von konkreten Entwurfsentscheidungen und Implementierungen zu abstrahieren.

### 4.3.4 Übersetzung und Einbindung von Quelltext zur Laufzeit

Die Übersetzung und Einbindung von Quelltext zur Laufzeit versucht, die Dauer der langwierigen Zyklen, die für die direkte Kodierung typisch sind, zu verkürzen. Die Basis des Entwurfs ist die Programmierung. Obwohl die Übersetzung zur Laufzeit des gesamten Programms abläuft, findet hier nur eine Verschiebung der Zeitpunkte statt. Der Quelltext, der die Benutzungsoberfläche erzeugt, wird weiterhin vor der Übersetzungszeit erstellt und dann übersetzt.

Die Bearbeitung der Benutzungsoberfläche erfolgt in folgenden Schritten:

1. Modifikation des Quelltextes
2. Übersetzung des Quelltextes
3. bei Fehlern im Quelltext zurück zu Schritt 1.
4. Starten des Programms
5. Bedienung des Programms bis die modifizierte Stelle betrachtet werden kann
6. bei Fehlern in der Benutzungsoberfläche zurück zu Schritt 1.

Die Dauer dieses Zyklus läßt sich vor allem verkürzen, wenn man das recht langwierige Starten des Programms eliminiert. Die anderen Schritte sind im allgemeinen nicht sehr zeitintensiv, so daß mit diesem Entwurf eine Zeitersparnis erzielt werden kann.

Die Übersetzung von Quelltext zur Laufzeit ist ein Ansatz, der von der jeweiligen Programmiersprache und der Systemumgebung unterstützt werden muß. So ist ein Mechanismus nötig, der das dynamische Laden und Ausführen von Code zur Laufzeit ermöglicht. Vor allem die dynamischen und die interpretierten Programmiersprachen (siehe Abschnitt 2.2.4) bieten solche Mechanismen an. Um dann die Vorteile dieses Entwurfs ausnutzen zu können, muß eine Möglichkeit bestehen, den übersetzten und ausgeführten Code während der Laufzeit des Programms durch eine neue Version zu ersetzen. Unter dieser Voraussetzung läßt sich dann der Quelltext beliebig oft dynamisch übersetzen und ausführen.

Der Vorteil dieses Entwurfs besteht dann darin, daß die Dauer des Entwicklungszyklus' deutlich kleiner wird, was eine erhebliche Erleichterung für den Programmierer bedeutet. Gleichzeitig bleibt die maximale Flexibilität und Mächtigkeit einer Programmiersprache erhalten.

## 4.4 Generierung

Ein konzeptuell anderer Ansatz mit der Komplexität grapischer Benutzungsoberfläche umzugehen, besteht darin, einen *Software-Generator* einzusetzen. Die Voraussetzung dafür sind vom Rechner lesbare und formalisierbare externe Daten, die zur Generierung herangezogen werden können. Beispiele hier sind reguläre Ausdrücke oder kontextfreie Grammatiken [Hopcroft, Ullman 93]. In dieser Arbeit sind die externen Daten die Dokumenttypdefinitionen des CoreMedia-Systems (siehe Abschnitt 3.2.1).

Der Quelltext, der eine Benutzungsoberfläche erzeugt, besteht aus mehreren Bestandteilen. Diese sind

- Daten zur graphischen Struktur,
- Daten zu Eigenschaften der Komponenten,
- Programmcode für die Bearbeitung von Benutzerinteraktionen und
- Programmcode zur Aktualisierung der Komponenten.

Mit *Daten zur graphischen Struktur* der Benutzungsoberfläche wird die Positionierung der Komponenten und die Aufteilung des zur Verfügung stehenden Platzes festgelegt (siehe Abschnitt 4.2.1). Weitere Daten legen die *Eigenschaften der Komponenten* fest, aus denen die Benutzungsoberfläche besteht. Die Komponenten werden mit *Code für die Bearbeitung von Benutzerinteraktionen* verbunden. Damit stoßen die Komponenten die Ausführung der Algorithmen an, in denen die Funktionalität des Programmes enthalten ist. Weiterer Code sorgt für die *Aktualisierung der Komponenten* der Benutzungsoberfläche nach Änderungen an den Daten des Programms.

Die Kodierung dieser Bestandteile ist textuell sehr umfangreich und besteht aus vielen, wenig variierten Wiederholungen. Dies per Hand zu erledigen ist aufwendig, monoton und damit fehlerträchtig. Wenn ein Programm diese Konfigurationsdaten generiert, braucht nur das Ergebnis, sofern dies nötig sein sollte, manuell nachbearbeitet werden.

Der Zeitpunkt für eine Generierung kann zur Übersetzungszeit oder zur Laufzeit liegen. Die Möglichkeiten, die bei der Generierung offen stehen, unterscheiden sich je nach dem Gegenstand und dem Zeitpunkt der Generierung deutlich. So können z.B. zur Lauf- und aber auch zur Übersetzungszeit Quelltext, Maschinencode, Konfigurationsdaten oder ein graphisches Layout generiert werden.

### 4.4.1 Gegenstand der Generierung

Die Generierung graphischer Benutzungsoberflächen kann auf unterschiedlichen Ebenen stattfinden. Sie begrenzen die Funktion, das Einsatzgebiet, die Möglichkeiten und die Grenzen eines Software-Generators. Gegenstand der Generierung kann sein:

- Quelltext,
- Maschinencode oder
- das Layout der Benutzungsoberfläche.

*Generierter Quelltext* wird durch einen Software-Generator erzeugt. Der generierte Quelltext muß zunächst von einem Übersetzer in Maschinencode übersetzt werden, bevor er ausgeführt werden kann. Die Ausführung des Maschinencodes erzeugt dann die Benutzungsoberfläche. Der Vorteil dieser Lösung liegt im reduzierten Schreibaufwand. Ein Programmierer kann auf diesem Wege eine Vorlage erhalten, die er modifizieren kann. Diese Art der Generierung ist beschränkt dadurch, daß ein Generator nur umfangreicheren, aber nicht mächtigeren Code als den des Generators selbst erzeugen kann.

Bei der *Generierung von Maschinencode* erzeugt ein Generator gleich Maschinencode und spart den Zwischenschritt über einen Übersetzer ein. Dies ist nur sinnvoll, wenn ein Generator für Maschinencode deutlich einfacher zu realisieren ist, als ein Generator für Quelltext. Im Hinblick auf die Erstellung graphischer Benutzungsoberfläche ist dies aufgrund der Komplexität sicher nicht der Fall. Obendrein besteht keine Möglichkeit, hochsprachliche Bibliotheken einzubinden, um den Aufwand für den Generator zu reduzieren. Dieser Ansatz wird hier nicht weiter verfolgt, da er keine Vorteile bietet.

Die *Generierung vom Layout der Benutzungsoberfläche* selbst geschieht durch ein generisches Programm, das den Zwischenschritt über die Generierung von Code einspart. Dieser Entwurf ist nicht so mächtig wie die anderen beiden, da bei den ersten beiden Entwürfen von Hand Veränderungen und Erweiterungen eingepflegt werden können. Der Vorteil beruht vielmehr darin, daß die Funktionalität nicht durch die Anpassung und Duplikation von Code wie in den ersten beiden Ansätzen, sondern durch einen generischen Ansatz erzielt wird.

### 4.4.2 Generierung von Quelltext

Bei der Generierung von Quelltext erzeugt ein Software-Generator zur Übersetzungszeit aus Definitionen von Dokumenttypen Quelltext. Der Quelltext wird in Maschinencode übersetzt und ausgeführt. Zur Laufzeit wird eine Benutzungsoberfläche für die Bearbeitung der Dokumente erzeugt und ausgeführt. Dazu wird für jeden Dokumenttyp in der Dokumenttypdefinition der Quelltext für eine Bearbeitungskomponente für Dokumente generiert. Dieser enthält die graphische Struktur der Attributtypen und unterstützt die Versionierung der Dokumente und den Arbeitsablauf des Systems.

Diese Art der Generierung von Quelltext ist eine Erleichterung für den Programmierer. Er bekommt so eine Vorlage, die er seinen Anforderungen anpassen kann. Alternativ kann erneut generiert werden. Bei einem einfachen Ansatz sind die Änderungen, die am älteren Quelltext gemacht werden, verloren. Die Berücksichtigung von manuellen Änderungen bei der Generierung ist kompliziert und nicht immer eindeutig. Es gilt herauszufinden, welche Änderungen die ältere Version in die neuere Version transformiert haben. Diese Änderungen müssen dann auf den neu generierten Quelltext angewandt werden. Ein solches Vorgehen ist fehlerträchtig, da die manuellen Änderungen sich auf den neuen Quelltext nicht notwendigerweise übertragen lassen. Die Software-Generatoren der integrierten Entwicklungsumgebungen lösen dieses Problem durch entsprechend markierte geschützte

Bereiche, die bei der Generierung unangetastet bleiben. Für diese Arbeit ist dieser Ansatz aufgrund der Änderungsproblematik und der auf Programmierer eingeschränkten Benutzergruppe zu unflexibel und zu aufwendig (siehe Abschnitt 2.2.3). Deshalb wird er nicht weiter betrachtet.

Dadurch, daß dieser Ansatz auf der Ebene von Quelltext arbeitet, steht die Flexibilität und Mächtigkeit zur Verfügung, die eine Programmiersprache bietet. So ist es problemlos möglich, den generierten Quelltext zu erweitern und an neue Anforderungen anzupassen. Damit bleibt auch die der Problemstellung inhärente Komplexität erhalten. Der Vorteil dieses Ansatzes beschränkt sich auf den geringeren Schreibaufwand gegenüber von Hand kodierten Lösungen.

### 4.4.3 Layout-Generierung

Die Layout-Generierung interpretiert zur Laufzeit die logische Struktur der Dokumente, um ihre Bearbeitung mit einer graphischen Benutzungsoberfläche zu ermöglichen. Dieser Ansatz ist generisch, da er die Erzeugung einer Benutzungsoberfläche ohne Kontextinformation oder Konfigurationsdaten erlaubt.

In einem ersten Schritt werden die Attribute eines Dokuments auf Komponenten der Benutzungsoberfläche abgebildet. Diese Komponenten erlauben die Anzeige und die Bearbeitung jeweils eines Attributes durch den Benutzer. Im zweiten Schritt werden die Komponenten zur Bearbeitung der Attribute für die Anzeige in einem Fenster plaziert. Dies geschieht mit Hilfe einiger, fest vordefinierter Regeln.

Bei diesem Ansatz handelt es sich um einen sehr allgemeinen und einfach gehaltenen Lösungsansatz. Er setzt voraus, daß es möglich ist, generische Komponenten zur Bearbeitung von Attributen zu realisieren. Dazu ist es nötig, für alle Attributtypen eine solche Bearbeitungskomponente zu erstellen, die ein Attribut ohne Einschränkungen anzeigen und bearbeiten kann. Sofern dies möglich ist, liegt der Vorteil dieses Ansatzes in der Universalität. Es ist ohne weitere Programmierung, Konfiguration oder Generierung möglich, beliebige Dokumente zu verarbeiten, wenn sie den Regeln einer Dokumenttypdefinition gehorchen. Sichtbare Ergebnisse sind in kürzestmöglicher Zeit realisierbar.

Der Vorteil der Universalität bedingt auch den Nachteil dieser Lösung. Die Flexibilität einer solchen Lösung ist gering. Es ist klar, daß eine generische Bearbeitungskomponente nur der kleinste gemeinsame Nenner aller denkbaren Möglichkeiten darstellen kann. Es ist nicht ohne weiteren Aufwand möglich, Eingabehilfen zu realisieren oder spezielle Datenformate für Attribute zu gewährleisten. Dazu ist es nötig, spezielle Anpassungen zu implementieren. Dies ist mit einer der Techniken zur Kodierung oder Quelltextgenerierung zu realisieren.

## 4.5 Generierung mit Nachbearbeitung

Die Generierung mit Nachbearbeitung ist eine Verfeinerung der Generierung. Sie baut konzeptionell auf der Layout-Generierung auf und versucht deren mangelnde Flexibilität durch Konfiguration zu verbessern. Dazu besteht für den Programmierer oder den versierten Benutzer die Möglichkeit in den Prozeß zur Erzeugung der Benutzungsoberfläche einzugreifen. Die Nachbearbeitung findet statt, nachdem aufgrund des Dokumenttyps ein Layout der Benutzungsoberfläche durch den Layout-Generator generiert wurde. Für jedes einzelne Attribut können Parameter wie

- die Sichtbarkeit eines Attributs,
- die Komponente zur Bearbeitung eines Attributs,

- die Positionierung der Bearbeitungskomponente,
- das Verhalten der Bearbeitungskomponente bei Größenänderungen,
- die Änderbarkeit einer Bearbeitungskomponente durch den Benutzer nach dem Ausleihen des Dokuments und
- die Konfigurationsdaten einer Bearbeitungskomponente

bestimmt werden. Diese Parameter dienen zur Steuerung der Generierung. Die Varianten der Nachbearbeitung unterscheiden sich durch die Art und Weise, wie die Parameter festgelegt werden.

### 4.5.1 Regelbasierte Nachbearbeitung

Die regelbasierte Nachbearbeitung steuert die Generierung über Regeln, die hinterlegt werden. Auch die einfache Layout-Generierung benutzt einige feste Regeln, um den Aufbau der Benutzungsoberfläche zu steuern. Die Regeln werden in diesem Ansatz durch einen extern konfigurierbaren Regelsatz ersetzt.

Der Regelsatz ist in einer sog. *Layout-Definition* mit Hilfe einer *Layout-Sprache* beschrieben. Mit einem Regelsatz erhält man u.a. die Möglichkeit, die graphische Struktur der visualisierten Dokumente detailliert zu beeinflussen. Die Begriffswahl ist hier durch die Analogie zu DSSSL und XSL geprägt. Diese beiden Sprachen definieren Regeln zur Beschreibung der graphischen Struktur von SGML- und XML-Dokumenten. Eine solche Regeldefinition, die die Transformation in eine graphische Struktur beschreibt, wird auch *Stylesheet* genannt (siehe Abschnitt 3.1.3).

Darüber hinaus kann die Layout-Sprache für die Layout-Definition Regeln enthalten, die die Zuordnung von Komponenten zur Bearbeitung von Attributen festlegen. Diese Regeln lassen sich erweitern und auch zur Konfiguration der Bearbeitungskomponenten benutzen.

Mit der regelbasierten Nachbearbeitung wird die mangelnde Flexibilität der Layout-Generierung über einen Konfigurationsmechanismus abgemildert. Zur Beeinflussung der Generierung ist eine eigene Layout-Sprache nötig, um die Möglichkeiten zur Beeinflussung ausnutzen zu können. Hier wird die Software durch eine Kombination von Konfiguration und Programmierung erstellt (siehe Abschnitt 4.2.3). Dieser Ansatz ist vergleichbar mit der Nutzung eines Rahmenwerks. Das Rahmenwerk löst die Problemklasse einmal. Diese Problemlösung wird durch die Definition von Code, der das Rahmenwerk an die eigenen Anforderungen anpaßt, wiederverwendet.

### 4.5.2 Interaktive Nachbearbeitung

Die interaktive Nachbearbeitung beeinflußt die Layout-Generierung durch die Interaktion mit dem Benutzer. Sie realisiert die Anforderung an ein Online-Redaktionssystem, eine graphische Benutzungsoberfläche für die Benutzer zur Verfügung zu stellen. Über eine leicht bedienbare Benutzungsoberfläche werden Konfigurationsdaten zum Layout-Prozeß erfaßt. Die Konfigurationsdaten dienen dazu, während der Generierungsläufe die Layout-Generierung entsprechend anzupassen.

Die interaktive Nachbearbeitung kann auf der regelbasierten Nachbearbeitung beruhen. Dabei vermeidet sie die wenig intuitive Erstellung der Regeln und verlagert sie in die Bedienung einer Benutzungsoberfläche. Die dort erfaßte Konfiguration wird dann wie in der regelbasierten Nachbearbeitung zur Beeinflußung des Layout-Prozesses genutzt. Die folgenden beiden Ansätze realisieren eine solche interaktive Nachbearbeitung. Sie unterscheiden sich durch den Zeitpunkt und die Art der Manipulation durch den Benutzer.

#### Manipulation der Reihenfolge

Die Manipulation der Reihenfolge geschieht durch einen zwischengeschalteten Eingriff des Benutzers. Vor der Layout-Generierung erhält der Benutzer die Gelegenheit, die Reihenfolge, in der die Attribute angezeigt werden, zu beeinflussen. Bei diesem Ansatz entscheidet der Benutzer lediglich, in welcher Reihenfolge die Layout-Generierung die Komponenten zur Bearbeitung der Attribute bei der Plazierung berücksichtigt. Die konfigurierte Reihenfolge wird in Form von Regeln abgelegt und bei weiteren Layout-Generierungen wie bei der regelbasierten Nachbearbeitung benutzt.

In einem nächsten Schritt stellt eine entsprechend erweiterte Benutzungsoberfläche eine Menge von Konfigurationsmöglichkeiten zur Verfügung. Dies sind sämtliche Konfigurationsdaten, die bei der regelbasierten Nachbearbeitung einstellbar sind.

#### Positionierung per direkter Manipulation

Die direkte Manipulation von Programmobjekten (*Drag and drop*) hat in den letzten Jahren weite Verbreitung unter den graphischen Benutzungsoberflächen gefunden. Der Ablauf dabei ist folgender: Zunächst selektiert ein Benutzer mit einem Mausklick ein Objekt der Oberfläche. Bei gedrückter Maustaste zieht er das Oberflächenobjekt an seine neue Position. Dieser Ablauf ist eine verkürzte Variante des Ausschneiden und Einfügen (*Cut and paste*) Ablaufs. Die Schritte dabei sind Selektieren eines Objektes, Ausschneiden des Objektes, Selektieren des Ziels und Einfügen des Objektes.

Die Manipulation der Dokumente per direkter Manipulation geschieht durch den Benutzer, nachdem die Layout-Generierung einen Vorschlag für die Plazierung der Komponenten zur Bearbeitung der Attribute gemacht hat. Dazu selektiert der Benutzer eine Komponente, die an eine andere Stelle plaziert werden soll. Während der Mausknopf gedrückt bleibt, verschiebt der Benutzer die Komponente und läßt sie an der Zielposition los. Dort wird die Komponente eingefügt. Die so konfigurierte Positionierung der Komponenten wird in Form von Regeln abgelegt und bei weiteren Layout-Generierungen wie bei der regelbasierten Nachbearbeitung benutzt.

## 4.6 Eingrenzung

Bei der Eingrenzung einer Lösung spielt neben der Realisierbarkeit vor allem die Zielgruppe eine Rolle. Die kodierten Lösungen sind durchweg nur für den Programmierer implementierbar. Die Anforderung, innerhalb eines Rahmenwerks zu programmieren ist eine andere, als eine Lösung direkt zu kodieren oder Standardwerkzeuge bei der Implementierung einzubinden. Insgesamt ist eine kodierte Lösung für Anwender des Systems nicht benutzbar.

Die generierten Lösungen bieten den Vorteil, daß sie keinerlei Aufwand erfordern und deshalb auch keine Einschränkung der Zielgruppe darstellen. Dieser Vorteil wird mit einer geringen Flexibilität erkauft, so daß mit wachsenden Anforderungen der Wunsch nach Anpassungen aufkommt.

Einen Mittelweg versucht die Generierung mit Nachbearbeitung zu beschreiten. Die interaktive Nachbearbeitung ist für Anwender benutzbar und bietet eine gewisse Flexibilität. Die regelbasierte Nachbearbeitung erfordert etwas mehr Kenntnisse und bietet dafür deutlich mehr Flexibilität.

Unter diesen Voraussetzungen bietet sich ein Lebenszyklus für die generische Visualisierung an. Das Zustandsdiagramm in Abbildung 4.12 zeigt die einzelnen Schritte des Lebenszyklus'.

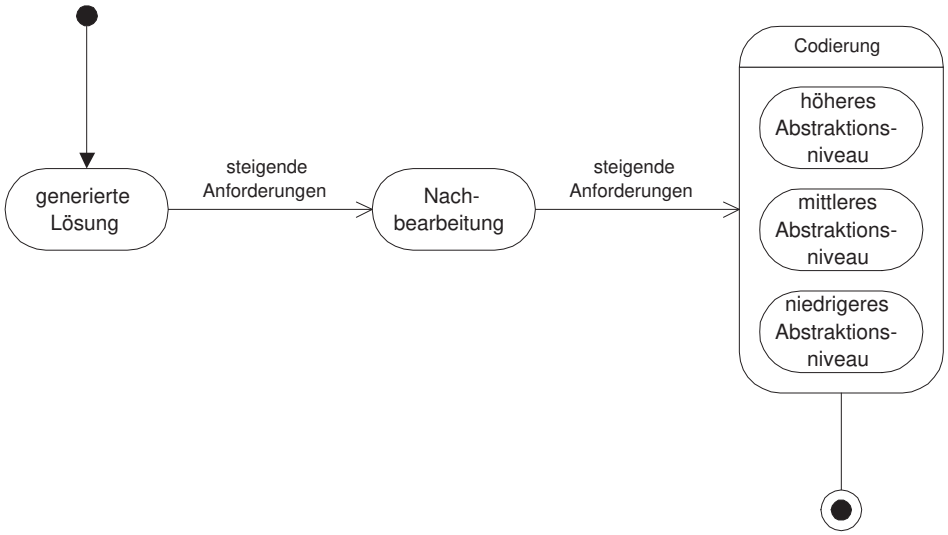


Abbildung 4.12: Anpassung der generischen Visualisierung

Die Visualisierung multimedialer Dokumente wird zunächst von einer generierten Lösung vorgenommen. Dieser Schritt kostet keinen Aufwand. Bei steigenden Anforderungen wird eine Nachbearbeitung des Ergebnisses der generierten Lösung durchgeführt, die bei mäßigem Aufwand eine gewisse Anpassung an die Anforderungen erlaubt. Sollten die Anforderungen weiter wachsen, steht ein Rahmenwerk mit mehreren Abstraktionsebenen bereit. Dies erlaubt es Programmierern, je nach Maßgabe der Anforderungen maßgeschneiderte, spezielle Lösungen zu erstellen. Der Aufwand einer solchen Lösung ist höher als bei der Nachbearbeitung, bleibt jedoch durch die Wahl einer geeigneten Abstraktionsebene des Rahmenwerks überschaubar.

# Kapitel 5

# Systemimplementierung

Die Systemimplementierung der generischen Visualisierung auf Basis des CoreMedia Online-Redaktionssystems basiert auf der bereits vorgestellten Software-Architektur, in der die Grundlagen der generischen Visualisierung und die Anforderungen aus dem Kontext der Arbeit eingegangen sind. Die Architektur ist durch eine Reihe von Systemvoraussetzungen beeinflußt, von denen die Wahl der Model-View-Controller-Implementierung sowohl für das Bearbeitungswerkzeug als auch für die generische Visualisierung entscheidend sind. Deshalb werden die Argumente für die gewählte Implementierung hier nochmal kurz aufgegriffen.

Der umfangreichste Teil der Systemimplementierung besteht in der Implementierung eines Rahmenwerks, das in mehreren Iterationsschritten erstellt wird. Es soll die Kodierung der Systemimplementierung durch ein höheres Abstraktionsniveau vereinfachen. Die Paketstruktur des Rahmenwerks gibt einen ersten Einblick in die Struktur der Implementierung. Mehrere Abstraktionsstufen untergliedern das Rahmenwerk weiter und bieten unterschiedliche Möglichkeiten zur Anpassung an besondere Anforderungen. Die Kernkomponenten, die die Sichten auf den Dokumentenspeicher, auf einzelne Dokumente und deren Attribute realisieren, werden detailliert behandelt. Der Schwerpunkt dabei liegt auf der Anpassung der Komponenten an spezielle Anforderungen.

Das realisierte Rahmenwerk bildet die Basis für die Implementierung des Generators, der Dokumentensichten erzeugt. Der Generator nutzt dazu die in den Grundlagen zur generischen Visualisierung vorgestellten Möglichkeiten des Typsystems. Er liefert eine graphische Struktur, die dem Benutzer das Dokument und seine Bestandteile visualisiert und die Bearbeitung der Attribute des Dokuments erlaubt. Damit ist die generische Bearbeitung der Dokumente im CoreMedia Online-Redaktionssystem möglich. Als Erweiterung dieses Ansatzes wird anschließend die Nachbearbeitung der durch den Generator erzeugten Lösung beschrieben. Die Implementierung der Nachbearbeitung geschieht auf Basis von Regeln oder eines interaktiven Benutzereingriffs. Sie erfüllt die Forderung nach einer weitergehenden Anpassung der Dokumentsichten, die durch den generischen Ansatz des Generators und die damit verbundenen Unzulänglichkeiten aufkommt.

Die Implementierungen der verschiedenen Sichten sind so realisiert, daß sie alle in den Grundlagen beschriebenen Dimensionen der Dokumente - Dokumentenmodell, Versionshistorie und Zustände - darzustellen vermögen. Es wird dargestellt, warum bestimmte Lösungswege zur Erreichung der gesteckten Ziele genommen werden. Die Vor- und Nachteile der vorgestellten Lösungen werden abgewogen.

## 5.1 Model-View-Controller-Implementierungen

Das *Model-View-Controller*-Anwendungsrahmenwerk von Smalltalk-80 wurde in Abschnitt 4.1.2 vorgestellt. Es ist bis heute Vorbild für Rahmenwerke, die die Entwicklung graphischer Benutzungsoberflächen unterstützen. Jedoch besitzt der Model-View-Controller-Ansatz einige strukturelle Probleme:

- Die strikte Trennung in drei Komponenten ist zu rigide.
- Es sind breite Schnittstellen zwischen den Komponenten erforderlich.
- Der einfache Benachrichtigungsmechanismus ist ineffizient.

Dies heißt nicht, daß der Model-View-Controller-Ansatz ungeeignet ist. Ganz im Gegenteil überwiegen die Vorteile des Ansatzes die Nachteile bei weitem. So bauen alle aktuellen Bibliotheken für graphische Benutzungsoberflächen wie MacApp [Knepper 98], die X Toolkit Widgets [Swick, Weissman 88], das Andrew Toolkit [Palay 88], InterViews [Linton et al. 87], ET++ [Weinand et al. 88] oder Swing [Sun Microsystems Inc. 99a] auf dem Model-View-Controller-Prinzip auf.

Die Defizite des Ansatzes werden von den Bibliotheken in unterschiedlicher Art und Weise angegangen, was für diese Arbeit frühzeitig eine Entscheidung für eine konkrete Implementierung des Model-View-Controller-Ansatzes notwendig machte. Die Wahl fiel dabei auf die Swing-Klassenbibliothek. Sie hebt sich durch eine Reihe von Vorteilen von den anderen GUI-Bibliotheken ab:

- Swing ist eine nach software-technischen Gesichtspunkten moderne und offene Klassenbibliothek. Die angebotenen Komponenten entsprechen dem aktuellen Standard für graphische Benutzungsoberflächen.
- Der Quelltext der Bibliothek liegt vor, so daß das Verhalten der Komponenten der Bibliothek verständlicher wird und ggfs. Fehler in der Bibliothek leichter behoben werden können.
- Das Aussehen und Verhalten (*Look and Feel*) der mit Swing erstellten graphischen Benutzungsoberflächen kann an die Rechnerplattform angepaßt werden. Das Erstellen eigener Anpassungen wird explizit unterstützt.
- Der Abstraktionsniveau der Bibliothek ist hoch. Eine graphische Benutzungsoberfläche läßt sich durch das einfache Zusammenfügen von Komponenten innerhalb kurzer Zeit erstellen.
- In Swing sind View und Controller verschmolzen, was die Anzahl der Schnittstellen auf die von Model und View reduziert. Damit ist die Handhabung der Swing-Komponenten deutlich einfacher als beim ursprünglichen Model-View-Controller-Ansatz.
- Das Model besitzt einen fein abgestuften Benachrichtungsmechanismus, um das View mit dem integrierten Controller von Veränderungen der darzustellenden Daten zu unterrichten. Dadurch kann das View den Umfang der Aktionen minimieren, die aufgrund einer Benachrichtigung vorzunehmen sind, um die Darstellung der Daten zu aktualisieren. Dies kommt der Verteilung der Komponenten auf mehrere Rechner entgegen (siehe 4.1.3).

- Das Layout der Benutzungsoberfläche wird mit Swing vollständig zur Laufzeit erstellt. Dies ist die Voraussetzung für die zur Realisierung der generischen Visualisierung anvisierte Generierung mit Nachbearbeitung (siehe 4.5).

- Swing ist vollständig in Java geschrieben, wie der Rest des CoreMedia Systems. Ein Programm wie das Bearbeitungswerkzeug für Dokumente, das über eine mit Swing erstellte graphische Benutzungsoberfläche verfügt, kann damit auf allen Rechnerplattformen mit Java-Unterstützung ausgeführt werden. Dies ist eine wichtige Anforderung an Online-Redaktionssysteme (siehe Abschnitt 2.2.4).

Diese Argumente sprechen deutlich für Swing als die am besten geeignete GUI-Bibliothek zur Realisierung einer graphischen Benutzungsoberfläche für die generische Visualisierung der Dokumente im CoreMedia Online-Redaktionssystem.

## 5.2 Implementierung des Rahmenwerks

Der Entwurf eines Rahmenwerks, das die generische Visualisierung unterstützt, wurde in Abschnitt 4.3.2 beschrieben. Insbesondere ist dort das größte Problem, das bei der Erstellung von Rahmenwerken zu bewältigen ist, beschrieben: die Wahl eines „richtigen“ Abstraktionsniveaus. Die Schwierigkeit liegt darin, herauszufinden, wann von einenm adäquaten, d.h. für Anwendungsprogrammierer benutzbaren, Abstraktionsniveau gesprochen werden kann.

Die vorliegende Implementierung eines Rahmenwerks ist in mehreren Iterationsschritten entstanden. Zunächst bestand noch wenig Erfahrung mit den Anforderungen an solch ein Rahmenwerk. Deshalb wurde das Abstraktionsniveau für das Rahmenwerk sehr niedrig angesetzt. Dadurch wird der Quellcode zur Visualisierung von Dokumenten mit konkreten Dokumenttypen recht umfangreich. Im Laufe der Zeit zeigte sich, daß die Anforderungen an die kodierten Sichten für Dokumentspeicher, Dokumente und Attribute weniger differieren, als zunächst vermutet. Es entstand der Wunsch, den Quellcode zur Visualisierung der Dokumente für konkrete Dokumenttypen möglichst kompakt und so abstrakt wie nötig zu halten.

Aus diesen Erfahrungen wurde das erstellte Rahmenwerk als ein Whitebox-Rahmenwerk mit drei Abstraktionsstufen realisiert. Als Whitebox-Rahmenwerk legt es seine interne Strukturen offen und kann so flexibler an neue Anforderungen angepaßt werden. Verwendet wird das Rahmenwerk durch Vererbung und das gezielte Überschreiben von Methoden. Der Nachteil eines solchen Whitebox-Rahmenwerks ist durch die Offenheit und damit seine Flexibilität begründet: Der Benutzer eines solchen Rahmenwerks muß die interne Struktur des Rahmenwerks erfassen und verstehen.

### 5.2.1 Abstraktionsstufen

Die folgenden drei Abstraktionsstufen sind aus den Erfahrungen mit den Anforderungen an das Rahmenwerk entstanden. Wie sich zeigte, konnte eine allgemein gehaltene Abstraktionsstufe (*Abstract*) alle Anforderungen erfüllen. Doch da sich die allermeisten Anforderungen überdecken, bietet es sich an, eine weitere Abstraktionsstufe (*Basic*) einzuführen, die den Aufwand für die Realisierung dieser üblichen Anforderungen minimiert. Auf dieser Stufe baut auch der generische Ansatz (*Generic*) zur Visualisierung der Dokumente auf. Die Stufen Abstract, Basic und Generic sind in Abbildung 5.1 in Form eines Klassendiagramms dargestellt.

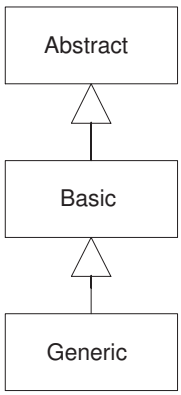


Abbildung 5.1: Drei Abstraktionsstufen

Im folgenden werden die drei aufeinander aufbauenden Abstraktionsstufen kurz vorgestellt:

**Abstract** Die Stufe Abstract ist die niedrigste Abstraktionsstufe, auf der das Rahmenwerk verwendet werden kann. Sie stellt für die Arbeit mit Dokumenten und dem System grundlegende Funktionalität zur Verfügung. Alle darüber hinausgehenden Funktionen sind als abstrakte Methoden ausgeführt, die von Subklassen implementiert werden müssen. Die jeweilige Lösung soll völlig frei von Entwurfsentscheidungen auf der Stufe Abstract sein.

**Basic** Auf einem deutlich höheren Abstraktionsniveau liegt die Stufe Basic. Sie implementiert die meisten der in der Stufe Abstract definierten abstrakten Methoden mit dem häufigsten Lösungsansatz. Die wenigen restlichen abstrakten Methoden sind von Subklassen zur konkreten Anpassung zu implementieren. Die Stufe Basic liefert somit eine Standardlösung, da sie das am häufigsten verwendete Verhalten realisiert. Darüber hinaus können die Subklassen von Basic die Implementierungen von Abstract und Basic gezielt überschreiben, um ihre Anforderungen umzusetzen.

**Generic** Die Stufe Generic baut auf der Stufe Basic auf und implementiert die wenigen abstrakten Methoden, die Basic nicht implementiert. Es handelt sich um eine konkrete Abstraktionsstufe, auf der eine generische Lösung realisiert wird. Komponenten auf der Stufe Generic sind universell verwendbar und unabhängig von Konventionen oder Voraussetzungen, die zur Anpassung des Systems an spezielle Anforderungen festgelegt werden. Subklassen von Generic können das von den Superklassen definierte Verhalten durch das gezielte Überschreiben von Methoden an ihre Anforderungen anpassen.

### 5.2.2 Strukturierung des Rahmenwerks

Ein Rahmenwerk muß klar strukturiert sein, um verständlich und verwendbar zu sein. Diese Strukturierung soll sich nicht nur in Klassendiagrammen wiederfinden, sondern auch bei der praktischen Verwendung des Rahmenwerks sichtbar sein. In C++-Rahmenwerken wird hierzu häufig eine Einteilung in Projekte genutzt, die die integrierten Entwicklungsumgebungen (siehe Abschnitt 4.3.3) zur Strukturierung des Quellcodes anbieten. Java stellt hierzu den *Paket* (*Package*)-Mechanismus bereit, der den Quellcode hierarchisch untergliedert. Zur Strukturierung des Rahmenwerks bieten sich zwei Möglichkeiten an Die Pakete können

- der Vererbungshierarchie folgen oder
- nach Zuständigkeit (*Responsibility*)

aufgebaut sein. Für das Rahmenwerk zur generischen Visualisierung wurde eine Strukturierung nach Zuständigkeiten gewählt, da dies das Verständnis der Pakethierarchie erleichtert und so die Einarbeitung in das Rahmenwerk vereinfacht. In Abbildung 5.2 ist die Pakethierarchie des Rahmenwerks mittels eines *Paketdiagramms* aus der Unified Modeling Language zu sehen. Die ersten Hierarchieebenen *coremedia* und *editor* kennzeichnen die Pakete als zugehörig zum Editor, dem Bearbeitungswerkzeug des CoreMedia Online-Redaktionssystems (siehe Abschnitt 2.2.5).

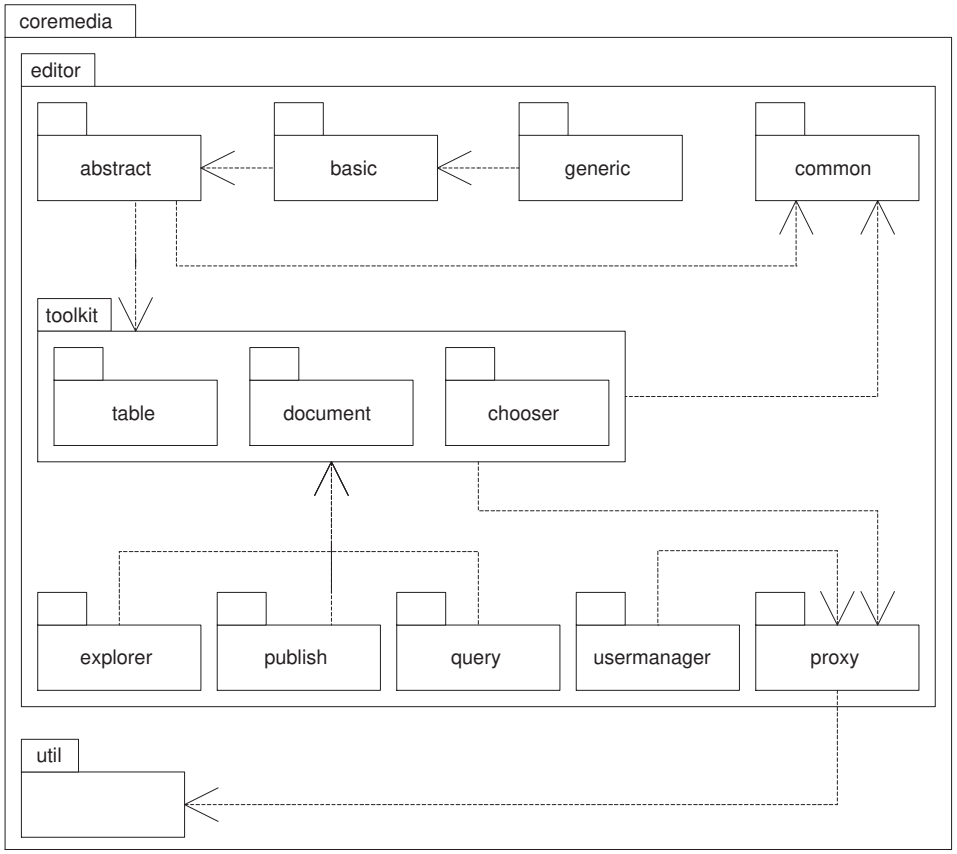


Abbildung 5.2: Hierarchie der Pakete

Zwei zentrale Pakete des Editors sind das *proxy*- und das *toolkit*-Paket. Im *proxy*-Paket ist die im Abschnitt 4.1.3 beschriebene Funktionalität zur Replikation der Daten des Servers enthalten. Das *toolkit*-Paket stellt die GUI-Komponenten des Editors und eine große Menge von Klassen zu ihrer Unterstützung bereit. Nach dem Model-View-Controller-Ansatz (siehe Abschnitt 4.1.2) realisieren die Klassen im *proxy*-Paket die Modelle (Model) für die im *toolkit*-Paket enthaltenen Sichten (View).

Das *common*-Paket enthält eine Menge von Basisklassen, die von den anderen Paketen im Editor benutzt werden. Die drei Abstraktionsstufen sind in den Paketen *abstract*, *basic* und *generic* zu finden. Die anderen Pakete bestehen aus Komponenten zur Visualisierung von speziellen Dokumentlisten für Recherche und Publikation sowie Klassen zur Benutzerverwaltung.

Durch diese Einordnung der Pakete kann sich der Benutzer leicht einen Überblick über das gesamte *editor*-Paket verschaffen. Die Benennung nach Zuständigkeiten erleichtert dies zusätzlich, und die Vererbungshierarchie wird durch diese Art der Strukturierung deutlich. Dies geschieht durch die Einordnung der Abstraktionsstufe in Pakete und durch einen entsprechenden Präfix vor den Klassennamen.

### 5.2.3 Zentrale Komponenten

Die zentralen Komponenten des CoreMedia Bearbeitungswerkzeugs für Dokumente sind im Paket *editor* zusammengefaßt. Es dient als Anlaufpunkt der anderen Komponenten des *editor*-Paketes für die zentralen Daten des Bearbeitungswerkzeugs. Der Suffix *Client* der zentralen Komponenten weist auf die zweite wichtige Funktion dieser Klassen hin: Der Editor ist ein Klient des Servers und bietet umfangreiche Unterstützung für diese Funktionalität. Auch wenn die Klassen den Suffix Client tragen, so ist synonym auch von den *Editor*-Klassen die Rede, um sich von den anderen Klienten des Servers abzusetzen und den Editor zu bezeichnen. In Abbildung 5.3 ist die Hierarchie der Editor-Klassen abgebildet. Sie zeigt die in Abschnitt 5.2.1 eingeführten drei Abstraktionsstufen.

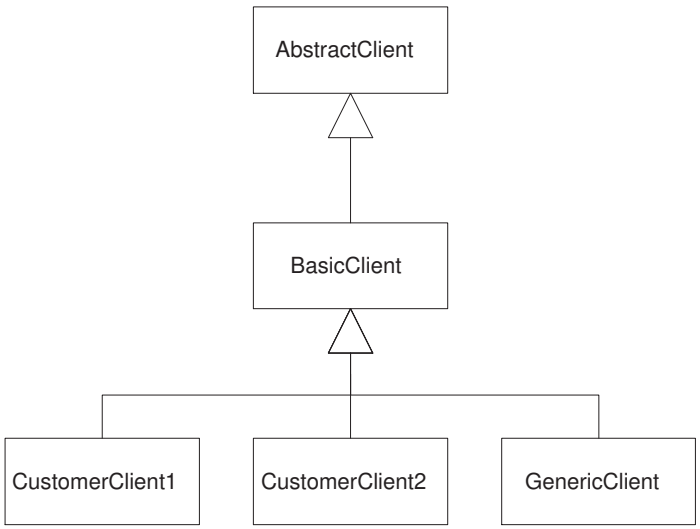


Abbildung 5.3: Hierarchie der Editor-Klassen

Die Editor-Klasse verwaltet die globalen Daten des Bearbeitungswerkzeugs und gestattet den Zugriff darauf. Über Fabrikmethoden nach dem *Factory Method*-Entwurfsmuster [Gamma et al. 94] können andere Komponenten über die Editor-Klasse Objekte erzeugen. Diese Funktionalität ist in Abbildung 5.4 dargestellt.

Die globalen Daten des Editors sind unterteilt in Referenzen auf Serverobjekte und global zugreifbare Objekte des Editors. Auf dem Server werden der Basisordner der Ordnerhierarchie (*Root Folder*), der Anmeldedienst (*LoginService*), die aktuelle Sitzung (*Session*) und die zentrale Verwaltungskomponente für Ressourcen (*Repository*) referenziert. Die global zugreifbaren Objekte des Editors realisieren nach dem *Singleton* Entwurfsmuster z.B. die Verwaltung der geöffneten Fenster (*FrameManager*), stellen Fabrikobjekte für das Erzeugen neuer Fenster bereit und passen die Darstellung der in den Fenstern visualisierten Ressourcen des Servers an. Eine solche Anpassung des Verhaltens ist durch das Implementieren von abstrakten Fabrikmethoden nach dem *Abstract Factory*-Entwurfsmuster möglich.

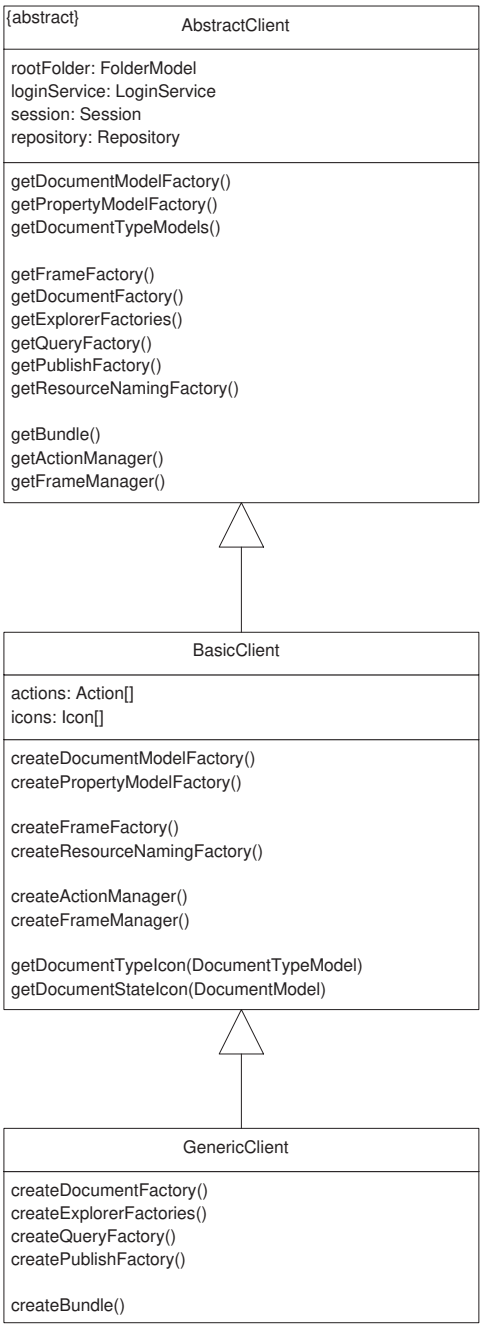


Abbildung 5.4: Die Klasse AbstractClient und ihre Subklassen

Die Klasse *AbstractClient* realisiert die Anbindung des Bearbeitungswerkzeugs an den CoreMedia Server. Dazu gehören

- der Aufbau und die Kontrolle einer Verbindung (*Session*) zum Server,
- die Verwaltung der Einstiegspunkte für die Ressourcen des Servers wie Dokumenttypen, Dokumente und Ordner (*Repository*),
- die Implementierung des Klientendatenmodells (*Proxy*),
- der Zugriff auf die weiteren Komponenten des Servers wie Benutzerverwaltung, Publikation und Bildverarbeitung sowie
- die Unterstützung für Filter- und Sortierdienste lokal beim Klienten oder entfernt auf dem Server (*SortingService*).

Innerhalb der Abstraktionsstufen gehört *AbstractClient* zur Stufe Abstract. Sie realisiert nur die grundlegende Funktionalität. Die meisten Methoden sind als abstrakt deklariert und müssen von Subklassen implementiert werden.

In der Klasse *BasicClient* sind die abstrakten Methoden in Form einer weithin gebräuchlichen Standardlösung implementiert. Dies erleichtert die Implementierung von Subklassen mit konkreten Anforderungen. Im detaillierten Klassendiagramm von Abbildung 5.4 erkennt man diese Funktion von *BasicClient* als Fabrik von Fabriken. Während in der Klasse *AbstractClient* lediglich die Methoden zum Abfragen von Fabrikobjekte (*get*) implementiert sind, realisiert die Klasse *BasicClient* die abstrakten Methoden zum Erzeugen der Fabrikobjekte (*create*). Sie implementiert

- die Bereitstellung und Verwaltung von globalen Aktionen für Benutzerinteraktionen (*Action*),
- die Wahl der Variante der Benutzungsoberfläche (*Look and Feel*),
- die Wahl der Sprachanpassung (*Locale*),
- die Verwaltung der Fenster und Symbole (*Icons*) der graphischen Benutzungsoberfläche und
- die Verwaltung der Benutzerkonfiguration des Bearbeitungswerkzeugs.

Die Klassen *GenericClient* und *CustomerClient* passen die Funktionalität der Superklassen an weitere Anforderungen an. Dazu gehören u.a.

- die Konfiguration der Sichten für Dokumentlisten und Dokumente und
- die Definition von Filtern und Sortierungen für Dokumentlisten.

Während die Klasse *GenericClient* den kleinsten gemeinsamen Nenner aller Anforderungen berücksichtigt, implementiert *CustomerClient* das konkrete Verhalten mit ganz speziellen Anforderungen. Durch die geeignete Wahl der übergeordneten Abstraktionsstufe Basic ist der Umfang des Quellcodes, der für die Anpassung an konkrete Anforderungen notwendig ist, gering. Für Anforderungen, die davon deutlich abweichen, kann die Klasse *AbstractClient* als Basis genommen werden. Sollte dies immer noch nicht ausreichen, ist das ganze Bearbeitungswerkzeug selbst zu erstellen.

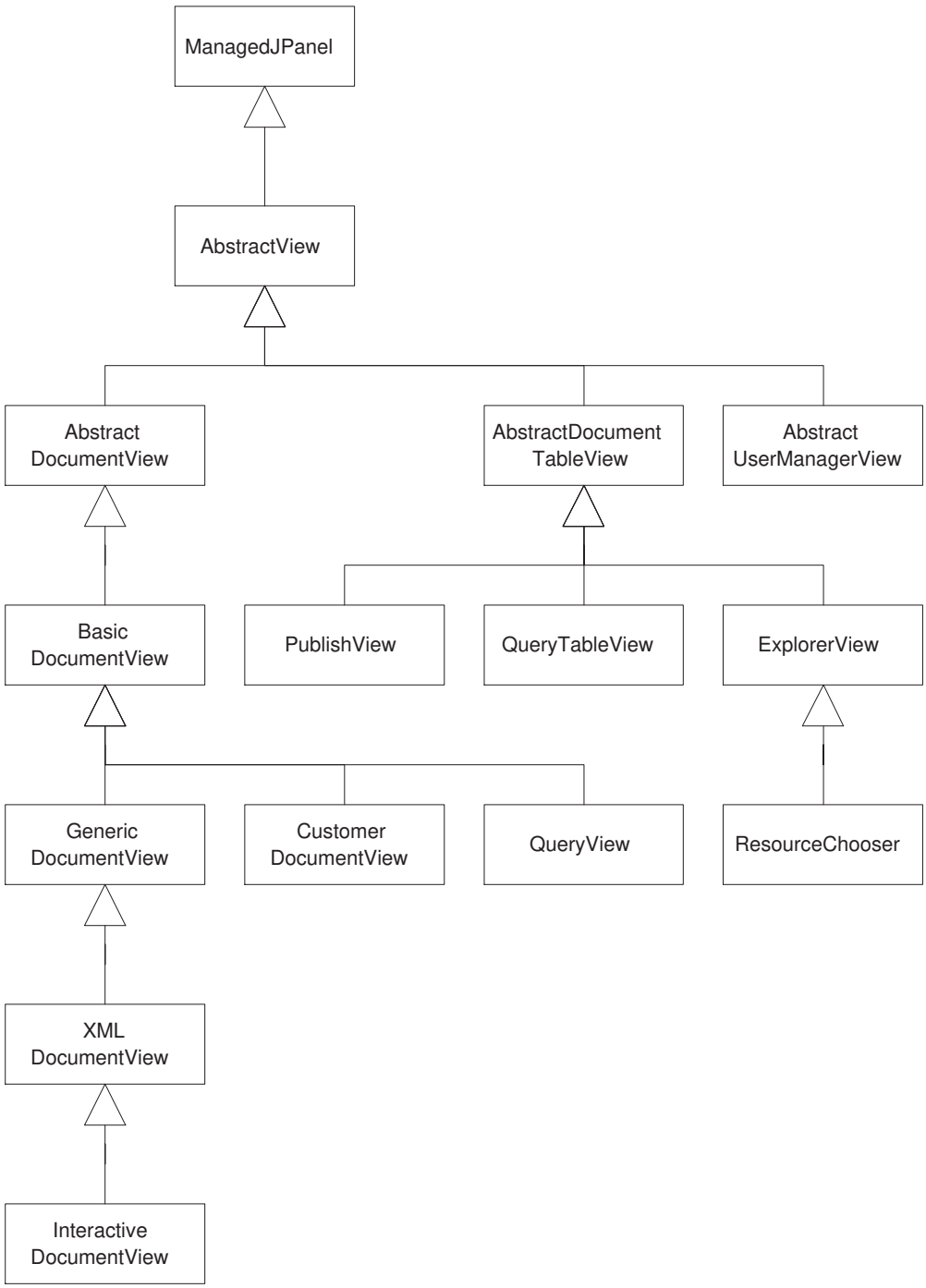


Abbildung 5.5: Hierarchie der Klassen für Dokumentsichten

### 5.2.4 Konfiguration der Sichten auf Dokumente

Die Konfiguration von Sichten auf Dokumente nutzt die in Abschnitt 5.2.3 beschriebenen Möglichkeiten der zentralen Komponenten. Eine *DocumentFactory* legt fest, welche Dokumentsicht für welche Typen von Dokumenten eingesetzt wird. Bei den Dokumentlisten bestimmen *ExplorerFactory*, *PublishFactory* und *QueryFactory*, wie die Dokumente in den Listen angezeigt werden. Wie bei den zentralen Komponenten bildet eine Klassenhierarchie die Basis für die Anpassung der Sichten an die unterschiedlichen Anforderungen. In Abbildung 5.5 ist die Hierarchie der Klassen für Sichten von Dokumenten und Dokumentlisten in Form eines Klassendiagramms abgebildet. Sie zeigt, daß alle Sichten für Dokumente auf einer gemeinsamen Basis beruhen.

Die Subklassen von *AbstractDocumentView* dienen zur Visualisierung von einzelnen Dokumenten. Die Subklassen von *AbstractDocumentTableView* realisieren verschiedenen Ansichten für Mengen von Dokumenten in Form von Listen. Die gemeinsame Basis der Sichten in der Klasse *AbstractView* ist so ausgelegt, daß sie beliebige Sichten auf Anwendungsobjekte zu unterstützen vermag. Die Klassen *ManagedJPanel* und *AbstractView* bilden den gemeinsamen Kern aller Sichten für die Objekte des Systems. Das Klassendiagramm in Abbildung 5.6 stellt sie detaillierter da.

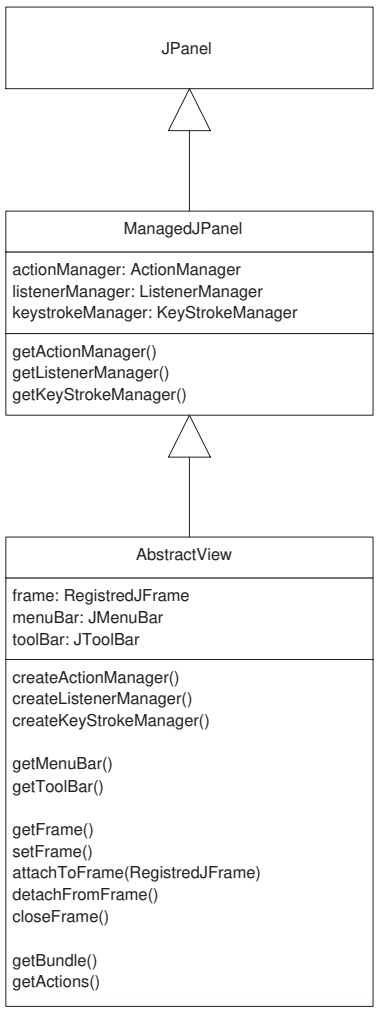


Abbildung 5.6: Die Klasse AbstractView

Die Klasse *ManagedJPanel* verwaltet Benutzerinteraktionen über die Klasse *ActionManager*. Dazu erzeugt sie die GUI-Komponenten für die Interaktion mit dem Benutzer über den *ListenerManager* und verknüpft die GUI-Komponenten mit den Benutzerinteraktionen. Außerdem bietet sie über den *KeyStrokeManager* eine komfortable Unterstützung für Tastaturkürzel (*Shortcuts*). Die Klasse *AbstractView* fügt dem die Einbettung in ein Fenster, eine Menüleiste (*Menubar*) und eine Symbolleiste (*Toolbar*) hinzu. Es ist von Vorteil, wenn die Sichten nicht direkt von der Klasse für Fenster (*JFrame*) abgeleitet werden. So können die Sichten als Container (*JPanel*) für weitere Sichten dienen, die als Komponenten in den Container eingebettet werden. Dies entkoppelt die Sichten von den Fenstern und ermöglicht es, beliebig viele Sichten in einem Fenster (und umgekehrt) anzuzeigen. Die Fenster werden von der Klasse *RegistredJFrame* realisiert, die die Funktionalität des flexiblen Hinzufügens und Entfernens von Sichten zu ihrem Container unterstützt.

Bei allen Gemeinsamkeiten besteht ein konzeptueller Unterschied zwischen den Klassen zur Visualisierung von Dokumentlisten und denen für Dokumente. Die Klassen für Dokumentsichten sind als Whitebox-Rahmenwerk realisiert, das durch Vererbung und gezieltes Überschreiben von Methoden in Subklassen angepaßt wird. Die unterschiedlichen Typen von Dokumentlisten sind als Blackbox-Rahmenwerk realisiert, das durch die Benutzung von Konfigurationsklassen den Anforderungen angepaßt wird. In den folgenden Abschnitten über die Sichten für Dokumentlisten und Dokumente wird darauf näher eingegangen.

## 5.3 Implementierung der Sichten

Für die Sicht auf die Dokumente und ihre Attribute sind die View-Komponenten des Rahmenwerks zuständig. Die Repräsentation der Daten, die den Sichten (View) zugrunde liegen, übernehmen Modelle (Model). Diese Aufteilung entspricht den Prinzipien des in Abschnitt 4.1.3 vorgestellten Verteilungsmodells.

Für jede der Aggregationsebenen, die durch die Typen definiert werden (siehe Abschnitt 4.2.2), steht eine Menge von Sichten und Modellen zur Verfügung. Die Sichten und Modelle für die Aggregationsebenen von Attributen, Dokumenten und dem Dokumentenspeicher werden in den folgenden Abschnitten vorgestellt. Der Schwerpunkt liegt dabei auf den Sichten, da die Modelle mit den Daten analog zu den Modellen des in Abschnitt 4.1.3 vorgestellten Verteilungsmodells aufgebaut sind.

### 5.3.1 Sichten für den Dokumentenspeicher

Die Visualisierung der Aggregationsebene des Dokumentenspeichers hat die Aufgabe, eine große Menge von Dokumenten übersichtlich zu präsentieren. Sie realisiert dies durch eine kombinierte Darstellung eines Baumes und einer Liste. Der Baum stellt die Hierarchie der Ordner, in denen die Dokumente liegen, dar. Mit Hilfe der Ordner kann die Anzahl der dem Benutzer auf einmal präsentierten Dokumente reduziert werden. Das Auf- und Zuklappen von Teilzweigen eines Baums ermöglicht dies für die Ordner selbst.

Die Basisklasse für alle Visualisierungen auf der Ebene des Dokumentenspeichers ist die Klasse *AbstractDocumentTableView*, die im Klassendiagramm in Abbildung 5.7 abgebildet ist. Ihre Leistung ist die Darstellung von Listen von Dokumenten. Die Baumdarstellung fügt die Klasse *ExplorerView* hinzu. Sie erlaubt es dem Benutzer, durch die Ordner und Dokumente des Dokumentspeichers zu navigieren. Die Auswahl einzelner Dokumente und Ordner aus dem Dokumentenspeicher geschieht genau wie bei der Übersicht über eine Baumdarstellung, die mit einer Dokumentenliste kombiniert wird. Dazu nutzt die Klasse *ResourceChooser* die Funktionalität des *ExplorerView*.

Die in Abbildung 5.5 abgebildete Hierarchie der Sichten zeigt noch weitere Subklassen von *AbstractDocumentTableView*. Sie realisieren weitere Anforderungen an das System.

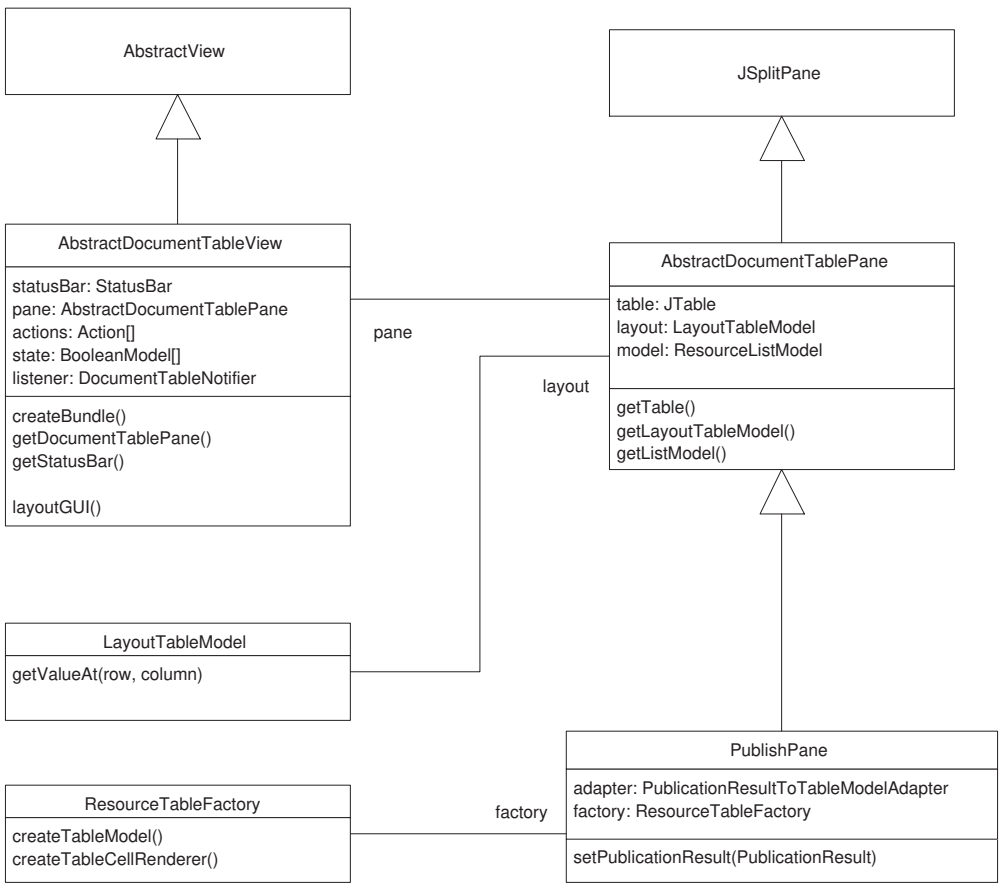


Abbildung 5.7: Die Klasse AbstractDocumentTableView

So stellt die Klasse *QueryTableView* das Ergebnis einer Recherche (siehe Abschnitt 2.1.1) in Form einer Liste mit den gefundenen Dokumenten dar. Der für den redaktionellen Prozeß wichtige Schritt der Publikation (siehe Abschnitt 2.2.2) wird durch die Klasse *PublishView* unterstützt. Sie stellt das Result der Publikation in Form einer Ergebnisliste mit Dokumenten und Kommentaren zu jedem einzelnen Dokument dar.

Für ihre Subklassen bietet die Klasse *AbstractDocumentTableView*:

- die Bereitstellung von Aktionen für Benutzerinteraktionen (*Action*) auf Dokumenten und Ordnern,
- die Verwaltung eines Zustandsmodells, so daß die Aktionen für Benutzerinteraktionen abhängig vom Zustand der selektierten Dokumente und Ordner aktiviert oder deaktiviert werden können, und
- einen Aktualisierungsmechanismus wie in Abschnitt 4.1.3 beschrieben, der eine Änderung der Sichten auf die Dokumente und Ordner aufgrund von Benachrichtigungen des Servers erlaubt.

Bei den Sichten für den Dokumentspeicher ist die Funktionalität aufgeteilt auf zwei Komponenten: Die Klasse *AbstractDocumentTableView* bildet den äußeren Rahmen für die

Sicht, während die Klasse *AbstractDocumentTablePane* die konkrete Anzeige der Dokumente realisiert. Dazu benutzt *AbstractDocumentTablePane* die Swing-Komponente *JTable* als Sicht und ein *LayoutTableModel* als Modell.

Für die Subklassen des *AbstractDocumentTablePane* wird im folgenden stellvertretend die Dokumentenliste im Publikationsfenster (*PublishView*) vorgestellt. Der nur die Publikation betreffende Teil der Funktionalität zur Anzeige der Dokumente ist in der Klasse *PublishPane* realisiert. Hier wird auch die zur Anzeige genutzte Tabellenkomponente *JTable* über eine *ResourceTableFactory* an die Erfordernisse der Publikation angepaßt. Dieser Vorgang wird weiter unten ausführlich erklärt. Zusätzlich muß das Resultat der Publikation, das als eine Liste von Dokumenten mit einem Kommentar zu jedem Dokument dargestellt werden soll, an die standardisierte Liste von Dokumenten, die das *AbstractDocumentTablePane* erwartet, angepaßt werden. Dazu dient ein weiterer Adapter, der *PublicationResultToTableModelAdaptor*, der das Publikationsergebnis auf ein *TableModel* umsetzt, mit dem die *JTable* Komponente arbeiten kann.

Die konkrete Konfiguration der Sicht auf die Dokumente setzt auf diesem Blackbox-Rahmenwerk auf. Bei den Sichten auf Listen von Dokumenten geschieht dies durch die Benutzung von Konfigurationsklassen. Der Grund für diese Ausnahme von der Regel, ein Whitebox-Rahmenwerk zu realisieren, liegt darin, daß die Anforderungen an Sichten auf Dokumentlisten kaum differieren. Es sind einzelne Attribute von Dokumenten, ihr Zustand im redaktionellen Prozeß (siehe Abschnitt 2.2.2) und Informationen über die Versionshistorie anzuzeigen. Dies gilt für alle in Abbildung 5.5 dargestellten Subklassen von *AbstractDocumentTableView*. Sie besitzen einen hohen Anteil an gemeinsamer Funktionalität und nur wenig Raum für Ansätze zur Konfiguration.

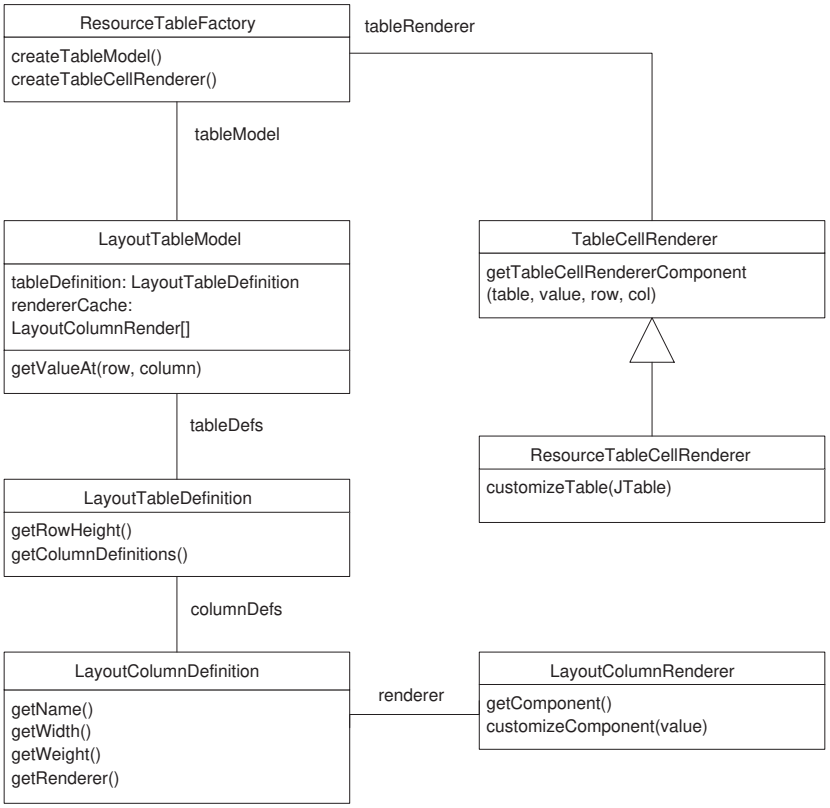


Abbildung 5.8: Die Klasse LayoutTableDefinition

Im Klassendiagramm von Abbildung 5.8 ist die Konfiguration der Sichten auf Dokumentlisten skizziert. Die Klasse *ResourceTableFactory* ist der Ausgangspunkt für die Konfiguration. Für die Umsetzung individueller Anforderungen dient sie dazu, die Sichten auf Dokumentlisten zu konfigurieren. Eine konkrete *ResourceTableFactory* implementiert:

- eine Wiedergabekomponente (*ResourceTableCellRenderer*), die die Zellen der Tabelle mit graphischen Inhalten ausmalen kann und
- ein Datenmodell (*LayoutTableModel*) für die Daten, das die Wiedergabekomponente zum Ausmalen der Tabellenzellen benutzt.

Neben der Aufgabe, die Tabellenzellen mit Inhalten zu füllen, konfiguriert die Wiedergabekomponente (*ResourceTableCellRenderer*) die *JTable*-Komponente mit Parametern aus einer Layoutdefinition für Tabellen (*LayoutTableDefinition*). Eine solche Layoutdefinition enthält:

- Angaben zur Höhe einer Zeile und zur Anzahl der Spalten in der Liste,
- Angaben über Namen, Breite und Ausdehnung einer Spalte und
- ein Wiedergabekomponente für Spalten (*LayoutColumnRenderer*), die für jede Zeile einer Spalte aufgerufen wird, um sie auszumalen.

Für die generischen Visualisierung wurde je eine *ResourceTableFactory* für die drei Sichten auf Dokumentlisten entwickelt. Diese werden von der Klasse *GenericClient* über Methoden zum Erzeugen der Fabrikobjekte (*create*) konfiguriert. Die Aufgabe einer generischen *ResourceTableFactory* ist es, von jedem Dokument die Informationen anzuzeigen, die unabhängig vom Dokumenttyp stets zur Verfügung stehen. Dazu gehört der Name des Dokuments, die Anzahl seiner Versionen, der Zustand im Arbeitsablauf und sein Typ. So ist auf diesem Weg eine generische Visualisierung der Dokumente möglich.

Über den in Abschnitt 5.4.2 beschriebenen Konfigurationsmechanismus können Attribute bestimmt werden, die zusätzlich zu den anderen Informationen in der Liste der Dokumente angezeigt werden. Dadurch kann die Anzeige der Dokumentlisten insoweit angepaßt werden, daß Attribute von besonderer Bedeutung bereits in den Sichten auf den Dokumentenspeicher angezeigt werden. Dies bedeutet eine Verkürzung des Arbeitsablaufs, da relevante Statusinformationen des Dokuments in der Dokumentenliste verfügbar sind, ohne es öffnen zu müssen.

### 5.3.2 Sichten auf einzelne Dokumente

Die Visualisierung auf der Aggregationsebene einzelner Dokumente muß sämtliche Informationen über das Dokument für den Benutzer zugreifbar machen. Die wichtigsten Informationen über den Zustand des Dokuments und alle seine Attribute werden sofort angezeigt. Sie können vom Benutzer, sofern er die Rechte dazu besitzt, manipuliert werden. Der Arbeitsablauf des Systems mit seinem redaktionellen Prozeß (siehe Abschnitt 2.2.2) und den daraus folgenden Wirkungen auf den Zustand der Dokumente muß für den Benutzer visualisiert und durch Benutzerinteraktionen kontrolliert werden. Dazu können detaillierte Informationen über die Versionhistorie der Dokumente, den Zustand innerhalb des redaktionellen Prozesses sowie die Rechte einzelner Benutzergruppen und des Benutzers an dem Dokument auf Verlangen angezeigt werden.

Die Basisklasse für die Visualisierung und Bearbeitung von Dokumenten ist die Klasse *AbstractDocumentView*, die in Abbildung 5.6 dargestellt wird. Sie realisiert die grundlegende Funktionalität, die Dokumentsichten zur Bearbeitung benötigen.

So bietet die Klasse *AbstractDocumentView*:

- Aktionen für Benutzerinteraktionen (*Action*) auf Dokumenten,
- ein Zustandsmodell, das die Aktionen für Benutzerinteraktionen abhängig vom Zustand des Dokuments aktiviert oder deaktiviert,
- einen Aktualisierungsmechanismus wie in Abschnitt 4.1.3, der eine Aktualisierung der Dokumentbearbeitungssicht aufgrund von Änderungen auf dem Server erlaubt,
- Visualisierungen der Versionshistorie, des Zustands und der Zustandsveränderungen für Dokumente,
- Unterstützung für die Attributsichten bei der Veränderung des Bearbeitungszustandes,
- Mechanismen zum Zurücksetzung und Wiederholen (*Undo*/*Redo*) von Aktionen auf Attributsichten und
- Unterstützung für den Arbeitsablauf des Systems bei der Bearbeitung von Dokumenten.

Damit bildet die Klasse *AbstractDocumentView* einen vordefinierten Rahmen für Subklassen, um deren Anforderungen an eine Dokumentbearbeitungssicht umzusetzen. Dafür gibt es eine Reihe von abstrakt deklarierten Methoden, die von dem konkreten Layout und dem Verhalten der Sicht abstrahieren. Die Anpassung an Anforderungen ist durch Vererbung, das Implementieren der abstrakten Methoden und das gezielte Überschreiben von Methoden in Subklassen möglich. Damit Subklassen Zugriff auf die internen Strukturen des *AbstractDocumentView* haben, wurde ein Whitebox-Rahmenwerk implementiert. So ist es möglich, sehr flexibel auf die Umsetzung besonderer Anforderungen einzugehen. Gerade bei den Dokumentbearbeitungssichten sind außergewöhnliche Anforderungen häufig, da die Benutzer die meiste Zeit auf diesen Sichten arbeiten und die Umsetzung der besonderen Anforderungen die Benutzbarkeit des Bearbeitungswerkzeugs deutlich verbessern kann.

Die Klasse *BasicDocumentView* realisiert eine Standardlösung für eine graphische Benutzungsoberfläche zur Bearbeitung von Dokumenten. Dazu implementiert sie die meisten abstrakten Methoden aus der Klasse *AbstractDocumentView*. Ein Klassendiagramm ist in Abbildung 5.9 dargestellt. Die Klasse *BasicDocumentView* ist wie die Klasse *AbstractDocumentView* abstrakt und überläßt die Auswahl, Positionierung und Initialisierung der konkreten Attributsichten eines Dokuments ihren Subklassen. Dies können konkrete Implementierungen im Rahmen eines Projektes wie *CustomerDocumentView* sein oder die generische Lösung der Klasse *GenericDocumentView*. Die Implementierung des Layout-Generators in der Klasse *GenericDocumentView* wird in Abschnitt 5.4.1 detailliert beschrieben. Die Erweiterung zur regelbasierten Nachbearbeitung in der Klasse *XMLDocumentView* ist in Abschnitt 5.4.2 erläutert.

Die Klasse *BasicDocumentView* erlaubt zur Laufzeit Veränderungen an den internen Menü-, Toolbar-, Komponenten- und Unterstützungsstrukturen, um Attributsichten flexibel und dynamisch in die Dokumentsicht einfügen zu können. Eine solche Flexiblisierung stellt einige Anforderungen an eine Attributsicht, damit die Klasse *BasicDocumentView* diese Leistung erbringen kann. Zu den Anforderungen an eine Attributsicht innerhalb des Rahmenwerks gehört, daß die Attributsicht

- dynamisch Einträge in die Menü- und die Symbolleiste einfügen kann,
- über eigene Aktionen für Benutzerinteraktionen (*Action*) manipulierbar ist,

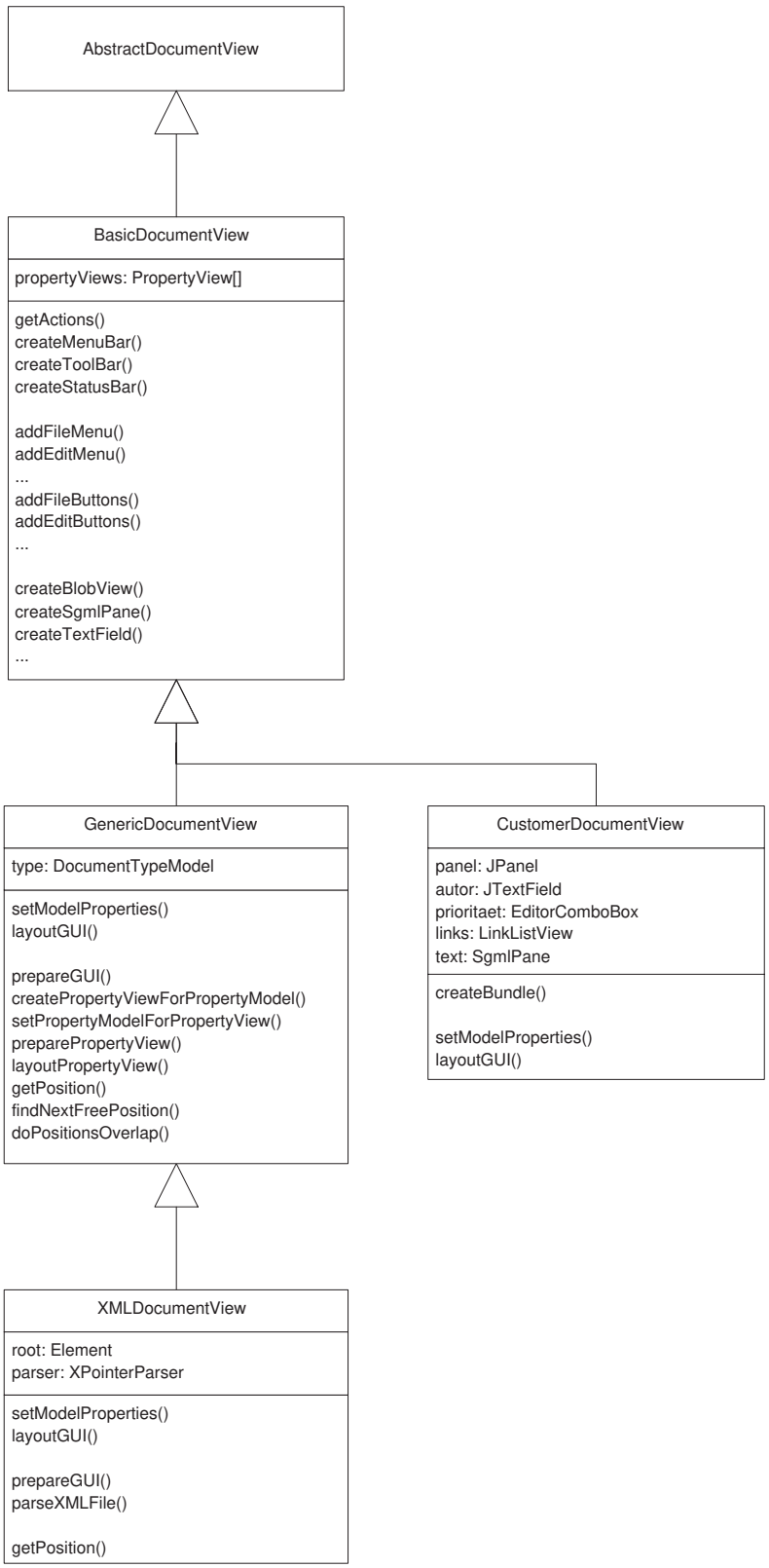


Abbildung 5.9: Die Klasse BasicDocumentView und ihre Super- und Subklassen

- über Tastaturkürzel für die Benutzerinteraktionen ausgelöst werden kann,
- genau dann bearbeitbar ist, wenn das Dokument ausgeliehen ist,
- beim Suchen und Ersetzen innerhalb des Dokuments berücksichtigt wird und
- am Zurücksetzen und Wiederholen (*Undo*/*Redo*) teilnimmt.

Das dynamische Zusammenstellen der Menü- und Werkzeugleisten ist problematisch, wenn Subklassen weiterhin die Möglichkeit offen stehen soll, die Menüs und Menüpunkte zu konfigurieren. So muß, bevor ein Menü oder ein Menüpunkt in die Menüleiste aufgenommen wird, berücksichtigt werden, ob die Konfiguration diesen Menüpunkt darstellen möchte. Diese Anforderung wird durch einen zweistufigen Ansatz gelöst. In einem ersten Schritt wird die Anforderung von Menüs und Menüpunkten durch die Attributsichten aufgenommen. Ausgehend von diesen Anforderungen werden in einem weiteren Schritt genau die Menüs und Menüpunkte in die Menüleiste aufgenommen, die durch die Konfiguration der Dokumentsicht berücksichtigt werden sollen.

Ein weiteres Problem besteht darin, daß die Swing-Klassenbibliothek in ihrer derzeitigen Version eine eindeutige Zuordnung einer Aktion zur Benutzerinteraktion zu einem Menüpunkt erfordert. Dadurch kann ein Menüpunkt nur auf einem Typ von Attributsichten arbeiten. Ein über die Komponenten übergreifenden Selektionsmechanismus kennt Swing nicht. Deshalb müssen die Attributsichten explizit verwalten, auf welche der beliebig vielen Attributsichten gleichen Typs die Aktionen zur Benutzerinteraktionen arbeiten sollen. Für das *BasicDocumentView* bedeutet diese Einschränkung, daß für jede Attributsicht und damit für jeden Attributtyp ein eigenes Menü erstellt werden muß. Die derzeitige Lösung zur Dynamisierung und Flexibilisierung der internen Strukturen ist entsprechend umfangreich, da um die angesprochenen Defizite herumgearbeitet werden muß.

Aufbauend auf der Klasse *BasicDocumentView* lassen sich für spezielle Anforderungen angepaßte Klassen wie die Klasse *CustomerDocumentView*, aber auch Klassen zur generischen Visualisierung (wie *GenericDocumentView*) implementieren. Diese speziellen Klassen unterscheiden sich in der Art und Weise, welche Sichten auf Attribute benutzt und wie sie behandelt werden. Der Ansatz zur generischen Visualisierung mit verschiedenen Nachbearbeitungsschritten, der auf der Klasse *GenericDocumentView* basiert, wird im Abschnitt 5.4 detailliert vorgestellt.

### 5.3.3 Sichten auf Attribute

Die Visualisierung auf der Aggregationsebene von Attributen hat die Aufgabe, einzelne Attribute eines Dokumentes übersichtlich zu präsentieren und für den Benutzer manipulierbar zu machen. Die Plazierung der Attributsichten findet innerhalb einer Dokumentsicht statt. Für jede Attributsicht muß erkennbar sein, welches Attribut sie darstellt und ob dieses Attribut für den Benutzer aktuell manipulierbar ist.

Die Attributsichten sind durchweg von Komponenten aus der Swing-Klassenbibliothek abgeleitet. Deshalb ist ein Ansatz, der auf der Vererbung beruht, nicht gangbar. Stattdessen bildet eine Reihe von Schnittstellen (*Interfaces*) den Zugang zu den Attributsichten. Die Klassen und Schnittstellen sind im Klassendiagramm von Abbildung 5.10 dargestellt.

Die wichtigste Schnittstelle für die Attributsichten ist die Attributsichtschnittstelle (*PropertyView*), die von den beiden Schnittstellen *Detachable* und *ActionComponent* erbt.

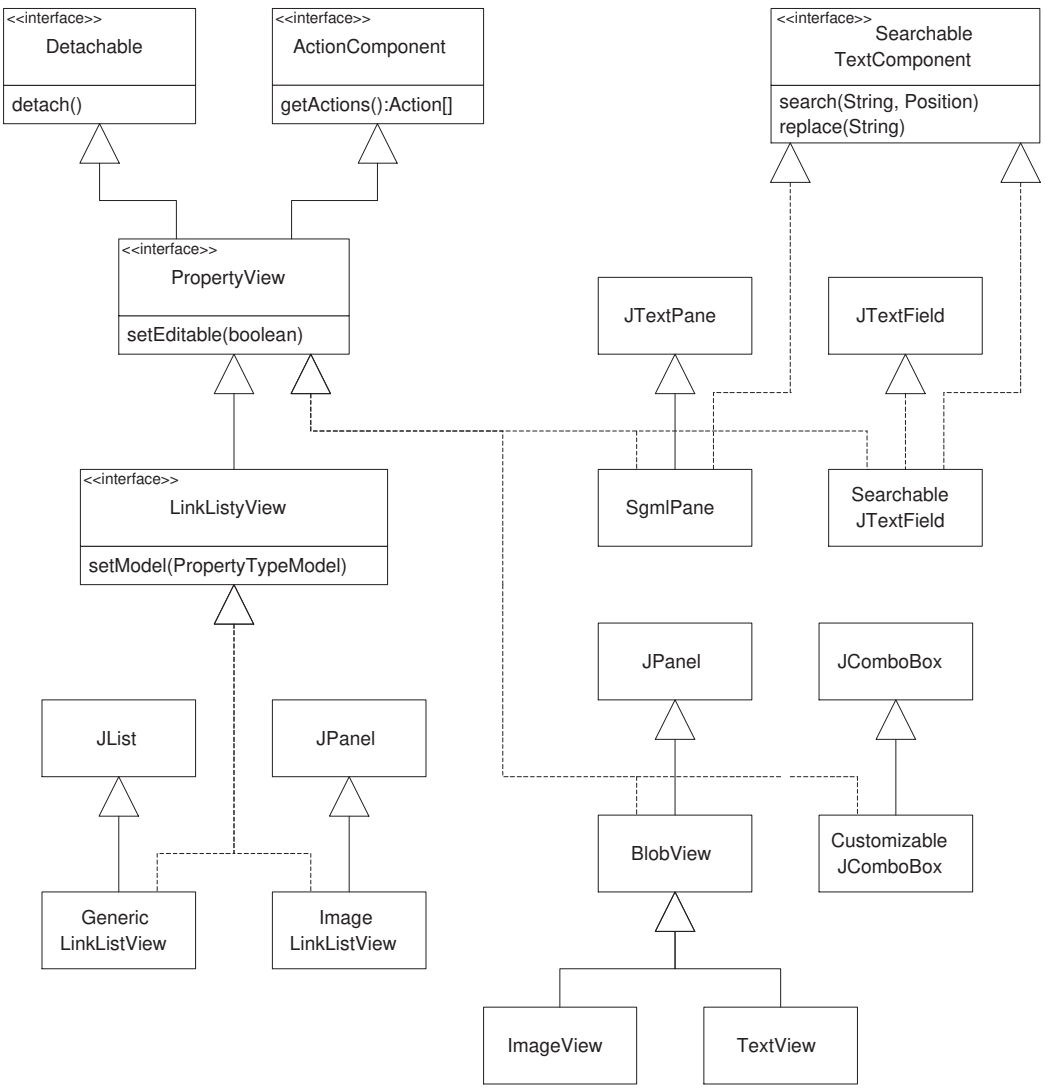


Abbildung 5.10: Hierarchie der Attributsichten

Für jede Attributsicht definiert die Schnittstelle *PropertyView*:

- einen Mechanismus, um die Attributsicht von allen Objekten, auf die sie Zugriff hat, zu trennen (detach), wenn das Attribut nicht mehr visualisiert werden soll,
- eine Methode, mit der die Aktionen für Benutzerinteraktionen (*Action*) einer Attributsicht in die Verwaltung der Aktionen der Dokumentsichten integriert werden können, und
- eine Möglichkeit, für die Attributsicht festzulegen, ob sie das Manipulieren des dargestellten Attributs erlaubt.

Die Schnittstelle *PropertyView* ist von allen Attributsichten zu implementieren. Für Textkomponenten existiert eine weitere Schnittstelle *SearchableTextComponent*, die die Attributsicht beim dokumentweiten Suchen und Ersetzen von Text einbindet. Für Komponenten, die Listen von Verweisen auf andere Dokumente darstellen, existiert eine spezielle Schnittstelle namens *LinkListView*. Über sie wird festgelegt, welches Attribut der in der Verweisliste erscheinenden Dokumente stellvertretend für das Dokument angezeigt wird.

In Abbildung 5.10 sind vor allem die Attributsichten dargestellt, die für die generische Visualisierung von Dokumente zum Einsatz kommen. Bei besonderen Anforderungen ist es nötig, diese durch die Konfiguration existierender Attributsichten, den Einsatz von Adaptern oder das Implementieren neuer Attributsichten zu erfüllen.

## 5.4 Generierung von Dokumentsichten

Die Generierung von Dokumentsichten implementiert die in Abschnitt 4.4 vorgestellte Architektur. In der Pakethierarchie befindet sich der Software-Generator inklusive der Komponenten zur Nachbearbeitung unterhalb des *editor*-Pakets im Paket *generic*. Ein Software-Generator generiert das Layout der Benutzungsoberfläche. Zwei Erweiterungen, die im Anschluß beschrieben werden, erlauben die Nachbearbeitung des generierten Layouts. Ihre Architektur ist in Abschnitt 4.5 beschrieben.

### 5.4.1 Layout-Generator

Der Layout-Generator für generische Dokumentsichten basiert auf dem in den vorigen Abschnitten beschriebenem Rahmenwerk. Die Software-Architektur in Abschnitt 4.4.3 beschreibt, wie der Layout-Generator die Typinformation eines Dokuments zum Aufbau des graphischen Layouts nutzt. Die generische Visualisierung der Dokumente wird durch die Klasse *GenericDocumentView* realisiert. Sie implementiert die abstrakten Methoden der Klasse *BasicDocumentView* (siehe Abschnitt 5.3), die sich mit der Auswahl, Positionierung und Initialisierung der Attributsichten eines Dokuments befassen.

Die Generierung der Dokumentsichten geht vom Typ des Dokuments aus, der zur Parametrisierung der Layout-Generierung und der Nachbearbeitung benötigt wird. Ein Dokumenttyp besitzt eine Menge von Attributtypen (siehe Abschnitt 3.2.1), anhand derer entschieden wird, welche Attributsicht ein Attribut des Dokuments zur Ansicht und Bearbeitung darstellt. Dazu wird eine feste Abbildung der Attributtypen auf die Attributsichten benutzt. Die Auswahl der Attributsichten für die Attribute muß bei einer generischen Lösung nach festen Regeln erfolgen. Die Flexibilität bei der Wahl der Attributsichten ermöglicht erst ein Nachbearbeitungsschritt. Gleichfalls durch den generischen Ansatz bedingt ist, daß die Attributsichten so allgemein gehalten sein müssen, daß sie in der Lage sind, alle Werte eines Attributtyps zu visualisieren und zur Bearbeitung zur Verfügung zu stellen. Für die Abbildung der Attribute auf Attributsichten ist bedeutsam, daß die

Klasse *AbstractClient* eine Möglichkeit bietet, über eine Fabrik (*PropertyModelFactory*) das klientenseitige Datenmodell zu konfigurieren. Dazu können die Verteilungsobjekte des abstrakten Verteilungsmodells (siehe Abschnitt 4.1.3), die auf der Seite des Klienten stellvertretend für die Server-Objekte agieren, angepaßt werden.

Für eine generische Lösung wird eine feste Menge an Verteilungsobjekten benötigt. Die Klasse *BasicClient* stellt dazu mit der Klasse *BasicPropertyModelFactory* eine allgemeine Lösung für die Abbildung der Server-Objekte auf die Verteilungsobjekte bereit, die auch die angepaßten Bearbeitungswerkzeuge wie der *CustomerClient* nutzen. In Tabelle 5.1 ist zu sehen, wie bei dieser allgemeinen Lösung die Verteilungsobjekte auf die Attributsichten abgebildet werden.

| Attributsicht | Adapter | Verteilungsobjekt | Serverobjekt |
|---|---|---|---|
| SearchableJTextField | BoundedDocument | BoundedStringModel | BoundedString |
| SearchableJTextField | IntegerModelToBounded-DocumentAdapter | IntegerModel | Integer |
| SearchableJTextField | CalendarModelToBounded-DocumentAdapter | CalendarModel | Date |
| GenericLinkListView | - | LinkListModel | LinkList |
| BlobView | - | BlobModel | Blob |
| SgmlPane | SgmlDocument | SgmlTextModel | SgmlText |

Tabelle 5.1: Abbildung von Attributsichten auf Serverobjekte

Für die generische Visualisierung ist lediglich eine neue, generische Attributsicht für Attribute vom Typ LinkList zu erstellen. Alle anderen Attributsichten sind bereits so allgemein gehalten, daß sie auch für die generische Visualisierung wiederverwendet werden können. Der Mehrwert, den die Attributsichten bei der Erstellung einer konkreten Dokumentsicht durch Eingabeunterstützung, Validierung der Eingabe, Sichtbarkeit und Manipulierbarkeit usw. bieten, wird durch die Konfiguration der Attributsichten erreicht.

Die Ausnahme bildet wiederum der Attributtyp LinkList, der durch Attributsichten mit ganz speziellen Anforderungen visualisiert wird. Beispielsweise wird als Repräsentation für ein Dokument in der LinkList häufig ein Bild gewählt, das als typisiertes Binärobjekt (Blob) in einem Attribut des referenzierten Dokuments gespeichert ist. Wenn eine Sicht auf eine LinkList generisch sein soll, darf sie nicht von der Existenz bestimmter Attribute ausgehen. Sie darf nur auf die Attribute oder Eigenschaften eines Dokuments zugegreifen, die die Dokumente unabhängig von ihrem Dokumenttyp bereitstellen. Die Klasse *GenericLinkListView* stellt dazu den Typ sowie den Namen des referenzierten Dokuments dar.

Nach der Wahl einer geeigneten Attributsicht muß diese in die Dokumentsicht eingebettet werden. Dazu implementiert die generische Sicht des *GenericDocumentView* einen Mechanismus, der die Attributsichten anhand fester Regeln plaziert. Ein Suchalgorithmus findet die nächste freie Position, an der eine gegebene Attributsicht plaziert werden kann. Vor der Attributsicht, d.h. optisch gesehen links von ihr, wird zur Identifikation der Name des Attributs plaziert. Bei der Anordnung wird darauf geachtet, daß sich die Attributsichten und die gemeinsam mit ihnen plazierten Namen nicht überschneiden. Einige weitere Regeln helfen dabei, eine möglichst ansprechendes Layout zu gestalten. Sie betreffen die Ränder (*Insets*) und das Verhalten bei Größenänderungen (*GridBagConstraints*) der Sichten. Der Name des Dokumenttyps, der Zustand des Dokuments im Arbeitsablauf und eine Komponente mit der Versionshistorie des Dokuments werden oberhalb der Attribute plaziert. Attribute vom Typ BoundedString, die eine gewissen Länge überschreiten, nehmen die komplette Breite der Dokumentsicht ein. Bei Attributen vom Typ Blob und SgmlText

darf die Attributsicht sich zusätzlich auch in der Höhe ausdehnen. Damit wird der größere Platzbedarf dieser Komponenten, der für ihre Bearbeitung benötigt wird, berücksichtigt.

Zwei Bildschirmfotos mit der typischen graphischen Struktur, die der vorgestellte Algorithmus produziert, sind in den Abbildungen 5.11 und 5.12 zu sehen.

### 5.4.2 Regelbasierte Nachbearbeitung

Die regelbasierte Nachbearbeitung, deren Architektur in Abschnitt 4.5 beschrieben wird, erweitert die Möglichkeiten der Layout-Generierung und erlaubt es, in den Layout-Prozeß einzugreifen. Die Klasse *XMLDocumentView* bietet die Möglichkeit, eine externe Layout-Definition in den Layout-Prozeß einzubeziehen. Dazu baut sie auf der Klasse *GenericDocumentView* auf, die den Layout-Generator realisiert. Für die externe Layout-Definition wird eine Beschreibungssprache benutzt, die Hinweise für den Layout-Prozeß in einem Dokument in der eXtensible Markup Language kodiert. Die Layout-Definition entspricht damit den Stylesheets von DSSSL, CSS und XSL (siehe Abschnitt 3.1.3). Jedoch sind im Vergleich dazu die Hinweise einer Layout-Definition für das *XMLDocumentView* deutlich einfacher und von geringerer Mächtigkeit.

Ein Hinweis in einer Layout-Definition beschreibt die Eigenschaften für die Attribute der Dokumente, die sonst unveränderlich von der Layout-Generierung festgelegt werden. Die DTD für Layout-Definitionen befindet sich in Anhang B.1. Sie spiegelt eine Liste von Eigenschaften für die Nachbearbeitung aus Abschnitt 4.5 wieder. Dort wurde allgemein beschrieben, was durch eine Layout-Definition beeinflußbar sein soll, um die für die üblichen Anforderungen notwendige Flexiblität zu erlangen. In Tabelle 5.2 sind die Hinweise dargestellt, mit denen sich die Parameter der Layout-Definition ausdrücken lassen.

| Parameter | Layout-Definition |
|---|---|
| Benennung des Attributs | Label |
| Sichtbarkeit der Attributsicht | IsVisible |
| Bearbeitbarkeit nach Ausleihen | IsEditable |
| Attributsicht zur Bearbeitung | View |
| Adapter zwischen Attributsicht und Verteilungsobjekt | Adapter |
| Positionierung der Attributsicht | Position mit X, Y, Width, Height |
| Verhalten bei Größenänderung | Resize mit Anchor, Fill, WeightX, WeightY |
| Konfigurationsdaten | Parameter mit Name, Value |
| In Übersicht anzeigen | ShowInExplorer |
| In Publikation anzeigen | ShowInPublication |
| In Recherche anzeigen | ShowInQuery |

Tabelle 5.2: Parameter einer Layout-Definition

Ein Beispiel für eine Layout-Definition zu dem im Abschnitt 3.1.3 vorgestellten Dokumenttyp für Gedichte und deren Autoren befindet sich in Anhang B.2. Die Konfiguration der graphischen Struktur über die Layout-Hinweise zu Posititionierung (*Position*), Größenänderung (*Resize*) und Ränder (*Insets*) lehnt sich an den Layout-Mechanismus des Java *Abstract Window Toolkit* (AWT) an. Dieses enthält eine Abstraktion für die Erzeugung graphischer Strukturen: Über sog. *LayoutManager* bestimmen die Behälter (*Container*) der Benutzungsoberfläche, wie ihre Komponenten (*Component*) positioniert und welchen Platz sie einnehmen dürfen.

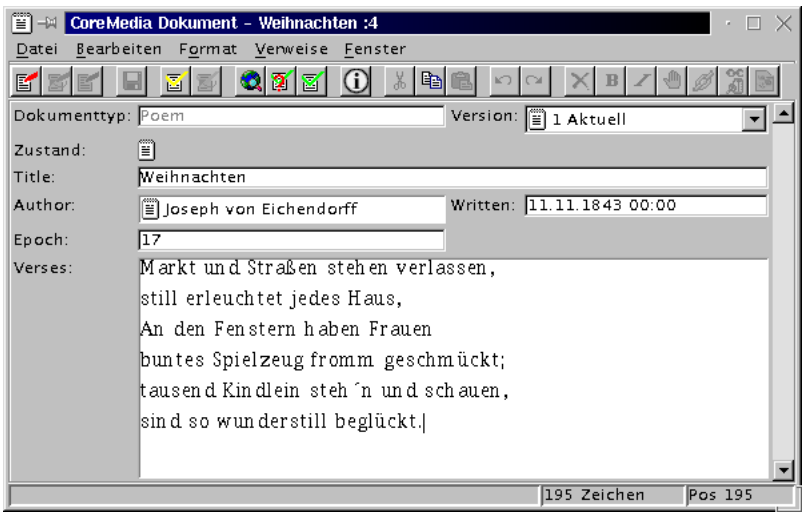


Abbildung 5.11: Bildschirmfoto zur graphischen Struktur (Poem)

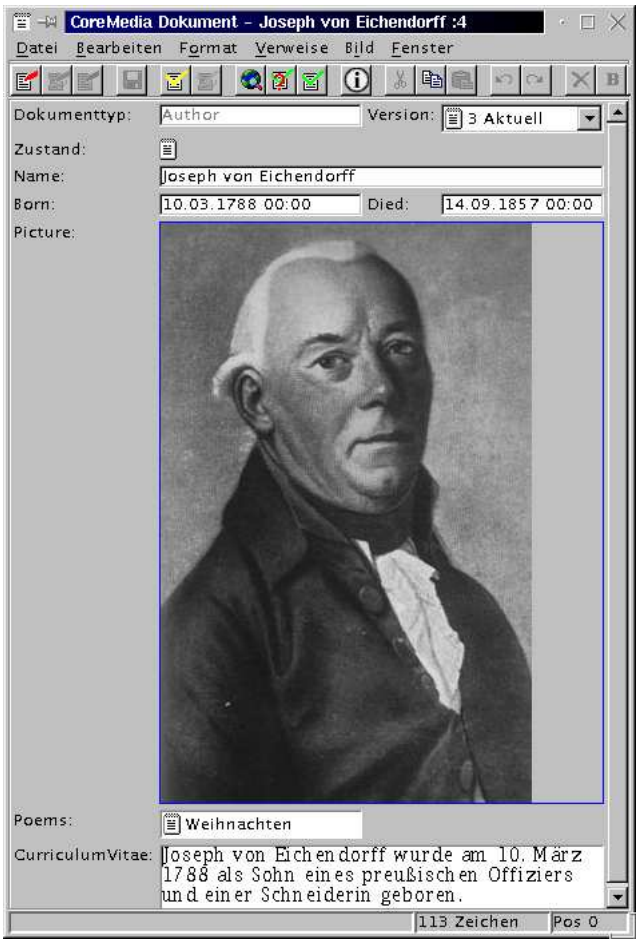


Abbildung 5.12: Bildschirmfoto zur graphischen Struktur (Author)

Der mächtigste *LayoutManager* im AWT ist das sog. *GridBagLayout*. Er teilt den Behälter in ein Gitter auf, in dem die Komponenten plaziert werden. Dabei ist eine Komponente nicht auf eine Gitterzelle beschränkt, sondern sie darf sich über mehrere Zeilen und Spalten erstrecken. Der Layout-Hinweis namess Position bietet diese Funktionalität zur Positionierung von Attributsichten. Für das Verhalten bei Größenänderungen benutzt das *GridBagLayout* Einschränkungen, die für jede Komponente als *GridBagConstraints* festgelegt werden können. Der Resize-Layout-Hinweis realisiert diese Einschränkungen der Ausdehnungen bei Größenänderungen für Attributsichten. Ränder werden beim *GridBagLayout* über *Insets*-Objekte festgelegt. Der gleichnamige Insets-Layout-Hinweis erlaubt es, Ränder für Attributsichten festzulegen.

Die Hinweise zur Anzeige von Attributen in den Dokumentlisten sind bereits in Abschnitt 5.3.1 erwähnt worden. Sie stellen einen Konfigurationsmechanismus dar, mit dem die Anzeige der Dokumentlisten an spezielle Anforderungen angepaßt werden kann. Die Integration dieser Konfigurationsoptionen in die regelbasierte Nachbearbeitung ist vorteilhaft, da alle externen Konfigurationsdaten an einer Stelle zusammengefaßt werden können.

Diese Lösung zur regelbasierten Nachbearbeitung bietet keine graphische Benutzungsoberfläche für wenig erfahrene Benutzer. Die Konfiguration über XML-Dokumente ist für Anwendungsprogrammierer und versierte Benutzer gedacht. Sie kann jedoch als Ausgangspunkt für die Speicherung der Ergebnisse einer interaktiven Nachbearbeitung dienen, die mit einer graphischen Benutzungsoberfläche die Erstellung der Regeln unterstützt. Alternativ könnten Standardwerkzeuge zum Bearbeiten von XML-Dokumenten eingesetzt werden.

### 5.4.3 Interaktive Nachbearbeitung

Die interaktive Nachbearbeitung bietet eine graphische Benutzungsoberfläche für die Nachbearbeitung der vom Layout-Generator erzeugten graphischen Struktur. Sie erweitert damit die Möglichkeiten der Layout-Generierung und erlaubt es, in den Layout-Prozeß einzugreifen. In Abschnitt 4.5.2 wird zwischen zwei Varianten unterschieden: der Positionierung per direkter Manipulation und der Manipulation der Reihenfolge.

Die Positionierung per direkter Manipulation ist durch ihre intuitive Bedienung vor allem für wenig erfahrene Benutzer geeignet. Sie beruht auf der direkten Manipulation der *LayoutManager*-Strukturen des AWT. Während die Dokumentsicht angezeigt wird, wählt der Benutzer eine Attributsicht aus und verschiebt sie an die Zielposition. Daraufhin wird die Attributsicht von ihrer ursprünglichen Position entfernt, indem auf ihrem Behälter (*Container*) Objekt die Methode *remove* aufrufen wird. An der Zielposition wird der neue Behälter für die Attributsicht und die Position der Attributsicht innerhalb des Behälters bestimmt. Die Methode *add* fügt die Attributsicht an der neuen Position ein.

Eine solche Implementierung der Positionierung per direkter Manipulation ist einfach zu realisieren und zu bedienen, leider stellt sie keine Lösung dar. Durch Fehler im AWT dauert das Entfernen von Komponenten aus Behältern, während der Behälter angezeigt wird, mehrere Sekunden. Oftmals sind mehrere Neuberechnungen des Layouts durch den *LayoutManager* notwendig. Das Hinzufügen von Komponenten dagegen zeigt überhaupt keine Wirkung. Das Fehlverhalten bleibt auch bestehen, wenn man den *LayoutManager* und den Behälter über einen Aufruf zur Validierung ihrer graphischen Struktur zum Neuzeichnen ihres Inhalts zwingt. Es bleibt zu hoffen, daß diese Fehler des AWT in der nächsten Version behoben werden.

Aufgrund dieser Defizite muß eine alternative Lösung zur interaktiven Nachbearbeitung mit einem Zwischenschritt auskommen. Bei der Manipulation der Reihenfolge sieht dies so aus, daß der Benutzer vor einer Layout-Generierung die Reihenfolge der Attribute beeinflußt. Dazu bietet die Klasse *InteractiveDocumentView* eine Liste der Attribute des

Dokuments und erlaubt dem Benutzer, per direkter Manipulation die Reihenfolge der Attribute zu verändern. Anschließend wird eine Layout-Generierung unter Berücksichtigung der vom Benutzer erstellten Reihenfolge durchgeführt. Diese beiden Schritt müssen solange wiederholt werden, bis der Benutzer eine zufriedenstellende Reihenfolge gefunden hat. Dann kann die Reihenfolge als Einstellung des Benutzers gespeichert werden, so daß sie bei der nächsten Visualisierung desselben Dokumenttyps zur Verfügung steht.

Diese mit dem *InteractiveDocumentView* realisierte Lösung ist für wenig erfahrene Benutzer geeignet, da sie eine intuitive graphische Benutzungsoberfläche bietet. Diesem Vorteil steht der Nachteil gegenüber, daß die Manipulationsmöglichkeiten nur sehr gering sind. Es wäre wünschenswert, alle Parameter, die die Visualisierung einer Dokumentsicht beeinflussen, manipulieren zu können, so wie dies bei der Kodierung oder bei der in Abschnitt 5.4.2 regelbasierten Nachbearbeitung möglich ist.

# Kapitel 6

# Zusammenfassung und Ausblick

Dieses Kapitel faßt das Thema der Arbeit zusammen. Es wird ein Bogen über die Kapitel geschlagen. Die Architektur zur generischen Visualisierung und die Entwurfsentscheidungen bei der Implementierung werden kritisch betrachtet. Die erreichten Anforderungen und die eingesetzten Werkzeuge werden beschrieben und bewertet. Im Ausblick wird erläutert, wie die realisierte Lösung verbessert, erweitert und auf andere Bereiche übertragen werden kann. Es werden Ansätze aufgezeigt, an denen im Bereich der generischen Visualisierung und im Anwendungsfeld zwischen Kodierung und Konfiguration weitergearbeitet werden kann.

## 6.1 Zusammenfassung

Diese Arbeit befaßte sich mit Unterstützungssystem für die digitale Publikation von redaktionellen Inhalten in den neuen Medien. Der Schwerpunkt lag auf der Problemklasse der Visualisierung multimedialer Dokumente mittels generischer Ansätze. Da dieser Kontext der Arbeit den Lösungsraum deutlich einschränkt, wurde die Problemklasse der Online-Redaktionssysteme mit ihren Anforderungen, Eigenschaften und Einsatzgebieten sowie ihrer technologischen Basis betrachtet. Der für die Leistung solcher Systeme zentrale Prozeß einer Redaktion über die zu veröffentlichenden Inhalte wurde beschrieben. Es wurden verschiedene Möglichkeiten, eine Schnittstelle des Systems zum Benutzer zu schaffen, erläutert und der Lösungsraum im Hinblick auf die Benutzergruppe der Redakteure auf eine graphische Benutzungsschnittstelle eingeengt. Ein Vergleich von ausgewählten Systemen führte zur Entscheidung, die Implementierung einer generischen Visualisierung auf Basis des CoreMedia Online-Redaktionssystems durchzuführen.

In den Grundlagen der generischen Visualisierung wurden Möglichkeiten vorgestellt, Dokumente durch Dokumentenmodelle und Typsysteme zu definieren. Auf Basis der Strukturinformationen, die bei CoreMedia das Typsystem für Dokumente und Attribut bereitstellt, kann eine graphische Struktur für die zu bearbeitenden Dokumente visualisiert werden. Die Trennung von logischen und graphischen Strukturen erlaubt eine Menge von Visualisierungsebenen, auf denen Dokumente erstellt werden können. Einige Visualisierungsebenen wurden vorgestellt und eine für den Kontext geeignete ausgewählt.

Auf der Basis dieser Vorarbeiten wurde eine Software-Architektur für ein graphisches Bearbeitungswerkzeug entworfen, das die generische Visualisierung und Bearbeitung multimedialer Dokumente erlaubt. Um den Lösungsraum möglichst gleichmäßig abzudecken, wurden eine Reihe von Alternativen zur generischen Visualisierung entworfen. Nach der Entscheidung für einen Entwurf wurden die Erfahrungen, Erkenntnisse und Schwierigkeiten bei der Implementierung beschrieben. Die Ideen, die in das zur Lösung realisierte Rahmenwerk und dessen Komponenten eingegangen sind, wurden dargestellt.

Für die Aufgabenstellung dieser Arbeit war der Kontext von großer Bedeutung. Durch die Problemklasse der Online-Redaktionssysteme bestand ein enger Rahmen für die generische Visualisierung. Innerhalb dieses Rahmens wurde eine Software-Architektur entworfen, die die an ein Online-Redaktionssystem gestellten Anforderungen möglichst optimal erfüllt. Eine wesentliche Komponente eines solchen Systems ist die Benutzungsschnittstelle. Die Benutzergruppe der Redakteure, die mit einem Online-Redaktionssystem innerhalb eines redaktionellen Prozesses arbeiten, beschränkt den Rahmen der Lösungen auf eine graphische Benutzungsoberfläche. Diese gelten als intuitiv zu bedienen, sind aber durch ihre Vielzahl von Funktionen aufwendig zu erstellen.

Bei der Erstellung einer Software-Architektur, die diese Eigenschaften besitzt, sind widersprüchliche Ziele zu erfüllen. Einerseits wird eine Leistungsfähigkeit von einem Online-Redaktionssystem verlangt, deren Umsetzung ein großes, komplexes System zur Folge hat. Der Aufwand für die Erstellung und Wartung einer solchen Lösung ist erheblich. Andererseits sollen innerhalb eines sinnvollen Rahmens beliebige Anforderungen flexibel realisierbar sein, ohne die technischen Grenzen der Architektur zu überschreiten. Der Aufwand, der für die Umsetzung solcher Anforderungen erforderlich ist, ist normalerweise hoch und steigert die Komplexität des Systems weiter. Um eine geeignete Lösung zu finden, wurden mehrere Alternativen zur generischen Visualisierung entworfen und prototypisch realisiert.

Anschließend wurde entschieden, den Generatoransatz weiterzuverfolgen, da dieser Ansatz sich von einer allgemeinen, aber unflexiblen Lösungen zu einer angepaßteren, spezielleren Variante weiterentwickeln läßt. Zur Realisierung des Generators und der Anpassungsschritte wurden ein Rahmenwerk, ein Layout-Generator und zwei Varianten zur Nachbearbeitung implementiert. Die Implementierung dieser Komponenten sowie die Entwurfsentscheidungen während der Realisierung wurden beschrieben.

## 6.2 Bewertung der Ergebnisse

Insgesamt ist eine generische Visualisierung für die Bearbeitung von Dokumenten erwünscht, da sie einfacher und weniger fehlerträchtig, also insgesamt wartungsfreundlicher ist. Als der größte Vorteil zeigt sich, daß ein generischer Ansatz praktisch augenblicklich Ergebnisse produziert. Nach der Definition eines neuen oder geänderten Dokumenttypen kann sofort mit Dokumenten von diesem Typ gearbeitet werden, ohne daß eine Anpassung oder Codierung nötig ist. Mit der Trennung von Inhalt und Layout, die auch im CoreMedia Online-Redaktionssystem realisiert ist, lassen sich viele Anwendungsfälle elegant lösen. Eine solche „late binding of presentation“ hat den Vorteil, daß auf die unterschiedlichen Anforderungen der verschiedenen Ausgabemedien flexibel reagiert werden kann, ohne die Datenquelle - das Dokument - zu verändern. Der implementierte Generator macht aus den ausgabeunabhängigen, logisch strukturierten Dokumenten eine konkrete graphische Struktur. Hier ist eine Verwandtschaft zur Generierung von externen Sichten für Internet Websites zu erkennen. Es zeigt sich, daß innerhalb der Rahmenbedingungen ein generischer Ansatz für die Visualisierung und Bearbeitung von Dokumenten realisierbar und im produktiven Umfeld einsetzbar ist.

In anderen Anwendungsfeldern gab es immer wieder ambitionierte Projekte zur generischen Visualisierung, die jedoch ohne nennenswerte Ergebnisse blieben. Im Unterschied zu dieser Arbeit stellte der Anwendungskontext bei den anderen Projekten keine Einschränkung dar. Folglich hatten diese anderen Projekte das ambitionierte Ziel, eine generische Visualisierung von Dokumenten ohne weitere Einschränkungen zu realisieren. Diese Anforderung führte zu einer Komplexität der Lösungen, die mit den bekannten technischen Mitteln nicht mehr handhabbar ist [Schreiber 94]. Im Unterschied dazu erwies sich bei dieser Arbeit das einheitliche Aussehen und Verhalten der Benutzungsoberfläche sowie der deutlich eingeschränkte Kontext als Vorteil, der die Realisierung überhaupt erst ermög-

lichte. Während die Arbeit an den Forschungsprototypen eingestellt wurde, wird die hier vorgestellte Lösung in einem Produkt eingesetzt, gepflegt und weiterentwickelt.

Der Vorteil der Universalität der Layout-Generierung, die es ermöglicht, schnell Ergebnisse zu produzieren, bedeutet auch einen erheblichen Nachteil, wenn es um weitere Anpassungen für die tägliche Arbeit mit dem Bearbeitungswerkzeug geht. Zum einen läßt das unattraktive, sehr regelmäßige Layout schnell den Wunsch nach einer Anpassung aufkommen. Zum anderen kann ein generischer Ansatz immer nur den kleinsten Nenner aller denkbaren Möglichkeiten realisieren. Die Vorteile, die eine graphische Benutzungsoberfläche verspricht (siehe Abschnitt 2.3.2) und die als Pluspunkt gegenüber einer Terminallösung (siehe Abschnitt 2.3.1) hervorgehoben wurden, können durch einen Layout-Generator nicht realisiert werden. Eingabehilfen, Auswahlfelder, interaktive Hilfen, Tooltips oder Assistenten sind nicht möglich. Dieses Ziel kann selbst mit einem Nachbearbeitungsschritt nur bedingt erreicht werden. Der Grund liegt darin, daß die im Rahmenwerk zur generischen Visualisierung implementierten Layout-Hinweise zu Adaptern und Views nicht geeignet parametrisiert werden können (siehe Abschnitt 5.4.1).

Das Ziel einer Nachbearbeitung für alle Benutzer wurde nicht erreicht. Der praktische Einsatz der Nachbearbeitung zeigt ein von der Benutzergruppe abhängiges Bild von der Akzeptanz der implementierten Lösung. Für Redakteure erweist sich die Nachbearbeitung über ein XML-Dokument und ohne graphische Benutzungsoberfläche als zu kompliziert. Dagegen kommen erfahrene Benutzer und Programmierer mit dem Nachbearbeitungsschritt gut zurecht. Dieser Benutzergruppe gehen die angebotenen Konfigurationsmöglichkeiten nicht weit genug. Zur Erfüllung anspruchsvollerer Anforderungen ist die Nachbearbeitung nicht leistungsfähig genug. Hier tritt das Dilemma auf, daß alle Funktionen zur Nachbearbeitung zunächst in der Programmiersprache implementiert werden müssen. Anschließend ist die Auswertung des XML-Dokuments mit der Layout-Definition sowie die DTD für Layout-Definitionen entsprechend zu erweitern. Schließlich muß die neu implementierte Funktionalität in einer konkreten Layout-Definition parametrisiert werden. Der Aufwand für diese Schritte ist so groß, daß die Nachbearbeitung über die Konfiguration in einer Programmier- oder Skriptsprache deutlich geeigneter erscheint. Doch dies schränkt die Benutzergruppe weiter ein.

Als eine Alternative zum Generatoransatz wurde die Nutzung von Standardwerkzeugen (siehe Abschnitt 4.3.3) gesehen. Wie sich jedoch zeigt, ist eine solche Lösung für den wenig erfahrenen Benutzer nicht einsetzbar. Die Funktionalität eines GUI-Editors aus einer IDE ist nur für solche Benutzer attraktiv, die sich ausschließlich auf die Konfiguration einer Dokumentsicht mittels einer graphischen Benutzungsoberfläche konzentrieren. Eine graphische Konfigurationsoberfläche für alle Sichten des Bearbeitungswerkzeugs bis zur Benutzungsreife zu implementieren, hätte den Zeitrahmen dieser Arbeit gesprengt und deshalb wurde diese Alternative nicht in Betracht gezogen. Andererseits gibt es zu diesem Ansatz keine andere Lösung, wenn eine möglichst große Benutzergruppe eine Anpassung des Bearbeitungswerkzeugs vornehmen können soll.

Zu diesem Ergebnis kommen auch andere Arbeiten aus der Forschung. Der Texteditor *Grif* [Quint, Vatton 92] ist als beliebig anpaßbarer Texteditor für strukturierte Texte entworfen worden. Die realisierte Lösung verwandte einen generischen Ansatz mit einer Konfigurationskomponente. In einer Diplomarbeit über die Konzeption einer Benutzungsoberfläche zur Erstellung von DSSSL Style Sheets [Mekelburg 99] wurde aufgrund des enormen Aufwands nur ein generischer, horizontaler Prototyp erstellt. In dem Projekt *Individualised Electronic Newspaper*[Hülser, Weber 92], waren im Unterschied zu den anderen Projekten genug Ressourcen zur Entwicklung vorhanden, so daß eine graphische Konfigurationsoberfläche erstellt werden konnte.

## 6.3 Bewertung der eingesetzten Werkzeuge

Die Modellierung der Software-Architektur und die Implementierung wurde durchgängig mit Elementen in der *Unified Modeling Language* [Rumbaugh et al. 99] durchgeführt. Sie dienen zur Visualisierung der statischen, dynamischen und funktionalen Struktur über Klassen-, Interaktions- und Anwendungsfalldiagramme. Da UML eine objektorientierte Modellierungssprache ist und mit Java eine objektorientierte Programmiersprache eingesetzt wurde, gab es keinen Bruch in der Notation. Insgesamt erschien UML als geeignet für die Notation der Konzepte und Strukturen in dieser Arbeit.

Die Programmiersprache *Java* [Sun Microsystems Inc. 96b] konnte bei der Implementierung des Rahmenwerks, der Generatoren und der Nachbearbeitung überzeugen. Die Syntax ist sehr leicht zu erlernen, wenn Vorkenntnisse in den Programmiersprachen C oder C++ bestehen. Besonders im Vergleich zu den genannten Programmiersprachen fallen die wesentlichen Entwurfsprinzipien von Java - Einfachheit und Robustheit - auf. Das CoreMedia Online-Redaktionssystem ist eine Client-Server-Anwendung mit sehr kleinen Objekten (*fine grained objects*), die über *CORBA* [Pope 98] kommuniziert und intensiv von *Multithreading* Gebrauch macht, um die Nebenläufigkeit zu erhöhen. Hier bietet Java mit der Unterstützung von *Threads*, *Synchronisation* und *Monitoren* [Sun Microsystems Inc. 97a] auf der Sprachebene gute Voraussetzungen, um ein verteiltes System zu implementieren. Jedoch schwankt die Qualität dieser Mechanismen in Abhängigkeit von der Systemplattform und der Implementierung der virtuellen Java-Maschine. Dies erschwert die Programmierung verteilter Systeme mit ihren inhärenten Problemen wie Synchronisation, Unteilbarkeit, Fairness und Verklemmungen (*Deadlock*) [Jessen, Valk 86] deutlich. Die für die Kommunikation mit CORBA benötigten ORBs sind überwiegend stabil, jedoch läßt die Geschwindigkeit und die Konformität zu aktuellen Versionen des CORBA-Standards zu wünschen übrig. Zusammengefaßt ist ein verteiltes System mit Java mit einem gewissen Aufwand realisierbar. Die Voraussetzungen dazu sind deutlich besser als bei den Programmiersprachen C oder C++. Bis der allseits gerühmte „ease of use“ von Java auch für die Implementierung von verteilten Systemen zutrifft, ist noch einiges zu verbessern.

Die Plattformunabhängigkeit von Java erwies sich als wesentlicher Vorzug, da zum einen die Programmierung in einer heterogenen Rechnerumgebungen stattfand und zum anderen bei der Ausführung der Programme keine Rücksicht auf Unterschiede zwischen den Systemplattformen genommen werden muß. Die graphische Benutzungsoberfläche wurde mit Komponenten der Swing-Klassenbibliothek realisiert. Die Swing-Komponenten sind komplett in Java implementiert und daher plattformunabhängig. Diesen Vorteil erkauft man sich mit einem erhöhten Bedarf an Ressourcen, die eine Swing-Benutzungsoberfläche benötigt. Nur wenn diese Ressourcen vorhanden sind, ist die Geschwindigkeit einer Swing-Benutzungsoberfläche ausreichend. Für die Benutzer ist das Bearbeitungswerkzeug der einzige Programmteil, mit dem sie direkt in Berührung kommen. Eine niedrige Geschwindigkeit oder mangelnde Stabilität des Bearbeitungswerkzeugs werden deshalb häufig auf das gesamte System bezogen. Hier fiel vor allem die mangelnde Reife der Swing-Komponenten auf. Deren Instabilität und Fehleranfälligkeit macht umfangreiche Tests und Umgehungen der Fehler (*workaround*), die den Einsatz einer externen Klassenbibliothek ad absurdum führen. Für den produktiven Einsatz scheint Swing im Moment nicht geeignet.

Ein weiterer Aspekt, den es zu bewerten gilt, ist der Einsatz von Swing-Komponenten, die nach dem verbreiteten Model-View-Controller-Paradigma arbeiten, in Kombination mit einer Client-Server-Anwendung wie dem CoreMedia Online-Redaktionssystem. Abgesehen von der fehlenden Reife der Swing-Komponenten ist der Aufwand, der mit Stellvertreter-Objekten und Zwischenspeichern getrieben werden muß, um eine annehmbare Arbeitsgeschwindigkeit zu erzielen, sehr hoch. Die Replikation der Server-Objekte erfordert eine Verdopplung des Quellcodes für die Server-Objekte und die klientenseitigen Stellvertreter-Objekte. Außerdem ist nicht nur die Server-Implementierung als zentrale Instanz prinzipbedingt aufgrund der Synchronisation der Klienten anfällig für Verklemmungen. Da die

Swing-Komponenten nur von einem Thread zur Zeit aktualisiert werden dürfen, müssen die Benachrichtigungen der Server-Objekte über Zustandsänderungen auf der Klientenseite serialisiert werden. Um Verklemmungen zu vermeiden, dürfen während der Abarbeitung der Benachrichtigungen keine Aufrufe an die entsprechenden Server-Objekte abgesetzt werden.Insgesamt ist die Kombination von Model-View-Controller-Komponenten für die graphische Benutzungsoberfläche und Client-Server-Anwendung technisch sehr anspruchsvoll und mit einer Vielzahl von Problemen behaftet. Der Aufwand, um eine solche Kombination produktiv einsetzen zu können, ist hoch. Eine Alternative könnte der Einsatz einer generischen Lösung für die Verteilung von Benutzungsoberflächen wie der in Abschnitt 4.1.2 vorgestellte Ultra Lightweight Client sein.

Während der Implementierung trat an vielen Stellen die Verteilung der Komponenten des Systems zu Tage. Es zeigt sich, daß die Kommunikationstechnik zwischen Klienten und Server die Verteilung der Komponenten nicht komplett verdecken kann. Dies ist jedoch der Anspruch der eingesetzten CORBA-Kommunikationstechnik. Zu den Versprechen, die CORBA gibt und nicht hält, zählt auch die Unabhängigkeit von Klient und Server, die durch die Definition der öffentlichen Schnittstellen gewährleistet werden soll. Es zeigt sich, daß die Klienten, allen voran das Bearbeitungswerkzeug, als ein Teil des Systems betrachtet werden muß, bei dem die Server-Seite den anderen Teil darstellt. Die Server-Seite und ihr Verhalten *ist* relevant für das Bearbeitungswerkzeug und alle anderen Klienten. Ein Fehlverhalten oder ein nicht auf die Klienten optimiertes Verhalten des Servers wirkt sich fast unmittelbar auf die Klienten aus.

Für die Nachbearbeitung der durch den Layout-Generator erstellten graphischen Struktur werden XML-Dokumente gelesen, geschrieben und ausgewertet. Dazu werden externe XML-Parser als austauschbare Komponenten eingebunden. Die Reife der bekannten XML-Parser ist für diese Aufgabe vollkommen ausreichend. Sofern Konfigurationsdaten wie die Layout-Definitionen in Form eines Baumes abgespeichert werden können und das Einlesen der XML-Dokumente nicht zeitkritisch ist, lohnt sich der Einsatz von XML-Dokumenten als universelle Konfigurationssprache.

## 6.4 Ausblick

Für die generische Visualisierung ist die Architektur des Gesamtsystems von Bedeutung, da sie die Basis darstellt, auf der die Lösungsvarianten zur generischen Visualisierung aufbauen müssen. Die Grenzen, die die Systemarchitektur setzt, lassen sich für die generische Visualisierung kaum aufweiten. Deshalb nimmt die Implementierung und Vorstellung der Systemarchitektur neben der Architektur zur generischen Visualisierung in dieser Arbeit einen relativ hohen Anteil ein.

Die eigentliche Lösung dieser Arbeit, die Implementierung des Generators, ist wie praktischen Erfahrungen aus dem Einsatz des Layout-Generators und der Nachbearbeitung zeigen, noch verbesserungsfähig:

**geschicktere Plazierung von Attributsichten** Die Plazierung mittels einiger fest kodierter Regeln ist wenig flexibel und erzeugt ein einheitliches, maskenähnliches Layout. Eine flexiblere Plazierungsstrategie kann die Typen der Attribute berücksichtigen sowie durch das Zusammenlegen von kurzen Textfelder ein kompakteres Layout erzeugen.

**Speicherung der Nachbearbeitungsregeln im Server** Die Regeln zur Nachbearbeitung werden lokal auf dem Rechner gespeichert, auf dem das Bearbeitungswerkzeug ausgeführt wird. Wechselt ein Benutzer des Systems den Rechner, an dem er arbeitet, sind die Nachbearbeitungsregeln nicht verfügbar. Zudem steht auf einem Rechner immer nur ein Satz von Nachbearbeitungsregeln zur Verfügung. Eine Differenzierung

nach Benutzergruppen ist so nicht möglich. Durch die lokale Speicherung können die Vorteile, die sich durch den Einsatz einer zentralen Instanz bei der Verteilung und der Synchronisation ergeben, nicht genutzt werden.

**weitere Informationen zu den Attributen** Die Definition der Dokumenttypen enthält neben dem Namen und dem Typ der Dokumente bzw. Attribute kaum Informationen. Für den Generator und die Attributsichten gibt es so keine Ansatzmöglichkeit, um z.B. aktiv Hilfestellung bei der Bearbeitung zu geben. Für Auswahllisten, die Überprüfung und Vervollständigung von Benutzereingaben, kontextsensitive Hilfe sind in den Attributtypen weitere Informationen erforderlich. Der Attributtyp SgmlText ist um eine DTD des verwendeten SGML-Textes zu erweitern, der Attributtyp Blob um eine Angabe über den Typ des Inhalts, die LinkList um den Typ der verweisenden Dokumente, der Typ Integer um Wertebereiche für die Eingabe sowie die Typen BoundedString und Date um Funktionen zur Überprüfung des Inhalts.

**reicheres, orthogonales Typsystem** Das Typsystem ist sehr einfach und eingeschränkt in seinen Möglichkeiten. Es gibt keine Aufzählungstypen, die eine variable Anzahl der atomaren Attributtypen BoundedString, Integer und Date aufnehmen können. Boole'sche Wahrheitswerte und Aufzählungen müssen auf die bestehenden Attributtypen abgebildet werden. Aufzählungstypen können nur linearisiert und mit fester Anzahl definiert werden. Der Generator weiß von diesen Entscheidungen nichts und kann keine geeignete Bearbeitung durch entsprechend angepaßte Attributsichten anbieten. Dies ist erst durch die Konfiguration von entsprechenden Adaptern in einem Nachbearbeitungsschritt möglich.

**intuitiv bedienbare Nachbearbeitung** Die regelbasierte Nachbearbeitung der durch den Generator erzeugten Layouts geschieht über ein XML-Dokument mit der Layout-Definition. Der wenig erfahrene Benutzer ist auf die unflexible, interaktive Nachbearbeitung beschränkt, in der nur die Reihenfolge der Attribute beeinflußt werden kann. Eine graphische Benutzungsoberfläche für alle Funktionen der regelbasierten Nachbearbeitung kann auch weniger erfahrene Benutzern die volle Mächtigkeit der regelbasierten Nachbearbeitung eröffnen.

**individuelle Dokumentsichten für Benutzergruppen** Die Kombination einer intuitiv bedienbaren Nachbearbeitung mit der Speicherung der Nachbearbeitungsregeln im Server ermöglicht es, Dokumentsichten individuell für einzelne Benutzer und für Benutzergruppen zu realisieren. Dadurch können sich Benutzer ihre Sicht auf die Dokumente an ihre Bedürfnisse anpassen und auf die Attribute der Dokumente konzentrieren, die für ihre Rolle im redaktionellen Prozeß wichtig sind.

**Mehrsprachigkeit** Die Attributsichten werden üblicherweise mit dem Namen des Attributs benannt. Auf den Bildschirmfotos in Abbildung 5.11 und 5.12 ist dies gut zu erkennen. Über die Nachbearbeitung ist lediglich eine Umbenennung dieser Namen möglich. Für ein vollständig lokalisiertes Bearbeitungswerkzeug muß die Definition von beliebig vielen lokalisierten Namen für die Namen der Dokument- und Attributtypen möglich sein.

**Erweitertes Regelkonzept** Eine Erweiterung des Regelkonzeptes kann die Regeln zur Definition des Layouts unabhängiger von den Dokumenttypen machen. So sind allgemeinere Regeln denkbar, die sich auf die Plazierung von Attributsichten beziehen. So könnte z.B. definiert werden, daß SgmlText-Attribute stets hinter Blob-Attributen, daß Date-Attribute im amerikanischen Datumsformat oder daß eine bestimmte Anzahl von BoundedString, Date und Integer Attributen horizontal nebeneinander angezeigt werden sollen.

Was die gesammelten Erfahrungen angeht, ist die vorgestellte Lösung durchaus übertragbar auf andere Problemstellungen. Eine generische Visualisierung benötigt zwingend Strukturinformationen über die darzustellenden Daten. Eine Möglichkeit, die Strukturinformationen bereitzustellen, besteht darin, ein Typssystem zur Beschreibung und Definition der Strukturinformationen zu nutzen. Steht keine Strukturinformation bereit wie z.B. bei den drei anderen Beispielen für Online-Redaktionssysteme (siehe Abschnitt 2.2.5), dann ist nur noch eine eingeschränkte generische Visualisierung möglich, die analog zu einem auf den Attributtyp Blob reduzierten CoreMedia-System funktionierte. Zur Verbesserung der Darstellung und Bearbeitung ließen sich externe Werkzeuge einbinden, so daß nach einer Analyse des Inhaltes der Blobs das passende Werkzeug aufgerufen und zur Darstellung aufgefordert werden könnte.

Das Typsystem des CoreMedia-Systems begrenzt die generische Visualisierung recht bald, da sie nur auf den kleinsten gemeinsamen Nenner aller Lösungsmöglichkeiten eingehen kann. Die oben vorgeschlagenen Erweiterungen können die Benutzbarkeit eines generischen Bearbeitungswerkzeugs nochmals verbessern. Doch langfristig führt kein Weg an einer graphischen Konfigurationsoberfläche, die mit Komponenten wie z.B. *Java Beans* [Sun Microsystems Inc. 97b] erweitert werden kann, vorbei. Dies ist auch der Weg, den alle größeren kommerziellen Softwareprodukte gehen.

Eine weitere Erkenntnis aus dieser Arbeit, die sich übertragen läßt, ist, daß eine Erweiterung der Leistungsfähigkeit einer generischen Lösung nur über weiteren Aufwand für Kodierung oder Konfiguration möglich ist. Die Frage, welches der bessere Ansatz ist, läßt sich nicht eindeutig beantworten. Beide Ansätze bilden entgegengesetzte Pole, von denen aus Lösungen entwickelt werden können, die das Feld zwischen Kodierung und Generierung abdecken. In dieser Arbeit wurde der Konfiguration der Vorzug gegeben, da ihr wesentlicher Vorteil ist, ohne eine Programmiersprache auszukommen. Dieser Vorteil wurde mit einer geringen Flexiblität und einem eingeschränkten Bearbeitungskomfort erkauft.

# Anhang A

# Dokumenttypen

## A.1 XML-DTD

Dies ist die Dokumenttypdefinition (DTD) für XML-Dokumente, die die Dokumenttypen im CoreMedia-System festlegen:

```
<!--
     this document type definition defines the structure
     of documents describing document types for the
     CoreMedia Content Publisher
-->

<!ENTITY % Property "StringProperty | IntProperty | DateProperty |
                     LinkListProperty | BlobProperty | SgmlTextProperty">

<!ELEMENT DocTypes (DocType)+>
<!ELEMENT DocType ( %Property; )*>
<!ATTLIST DocType Name ID #REQUIRED
                  Parent NMTOKEN #IMPLIED>

<!ELEMENT StringProperty EMPTY>
<!ATTLIST StringProperty Name NMTOKEN #REQUIRED
                         Length NMTOKEN #REQUIRED>

<!ELEMENT IntProperty EMPTY>
<!ATTLIST IntProperty Name NMTOKEN #REQUIRED>

<!ELEMENT DateProperty EMPTY>
<!ATTLIST DateProperty Name NMTOKEN #REQUIRED>

<!ELEMENT LinkListProperty EMPTY>
<!ATTLIST LinkListProperty Name NMTOKEN #REQUIRED>

<!ELEMENT BlobProperty EMPTY>
<!ATTLIST BlobProperty Name NMTOKEN #REQUIRED
                       MimeType CDATA #REQUIRED>

<!ELEMENT SgmlTextProperty EMPTY>
<!ATTLIST SgmlTextProperty Name NMTOKEN #REQUIRED>
```

## A.2 XML-Dokument

Dies ist ein Beispiel für ein XML-Dokument, das die Dokumenttypen im CoreMedia-System festlegt:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<!--
     this is a sample document type definition of the
     CoreMedia Content Publisher for a list of poems
-->

<!DOCTYPE DocTypes SYSTEM "../../lib/xml/corem-doctypes.dtd">

<DocTypes>
  <DocType Name="Author">
        <StringProperty Name="Name" Length="100"/>
        <DateProperty Name="Born"/>
        <DateProperty Name="Died"/>
        <BlobProperty Name="Picture" MimeType="image/jpeg"/>
        <LinkListProperty Name="Poems" LinkType="Poem"/>
        <SgmlTextProperty Name="Curriculum Vitae"/>
  </DocType>

  <DocType Name="Poem">
        <StringProperty Name="Title" Length="100"/>
        <LinkListProperty Name="Author" LinkType="Author"/>
        <DateProperty Name="Written"/>
        <IntProperty Name="Epoch"/>
        <SgmlTextProperty Name="Verses"/>
  </DocType>
</DocTypes>
```

# Anhang B

# Layout-Definitionen

## B.1 XML-DTD

Dies ist die Dokumenttypdefinition (DTD) für XML-Dokumente, die die regelbasierte Nachbearbeitung zur generischen Visualisierung im CoreMedia-System konfigurieren:

```
<!--
     this document type definition defines the layout of
     documents which are made visible by the editor of the
     CoreMedia Content Publisher
-->

<!ENTITY % Directive "Property | DocumentTypeName | State | Version">
<!ENTITY % Constraint "Position | Resize | Insets">
<!ENTITY % Class "Adapter | View">

<!ELEMENT LayoutDefs (DocType)+>
<!ELEMENT DocType ( %Directive; )*>
<!ATTLIST DocType Name ID #REQUIRED>

<!ELEMENT Property ( %Constraint; %Class; )*>
<!ATTLIST Property Name NMTOKEN #REQUIRED
                   Label NMTOKEN #IMPLIED
                   IsEditable NMTOKEN #IMPLIED
                   IsVisible NMTOKEN #IMPLIED
                   ShowInExplorer NMTOKEN #IMPLIED
                   ShowInPublication NMTOKEN #IMPLIED
                   ShowInQuery NMTOKEN #IMPLIED>

<!ELEMENT DocumentTypeName ( %Constraint; )*>
<!ELEMENT State ( %Constraint; )*>
<!ELEMENT Version ( %Constraint; )*>

<!ELEMENT Position EMPTY>
<!ATTLIST Position X      NMTOKEN #IMPLIED
                   Y      NMTOKEN #IMPLIED
                   Width  NMTOKEN #IMPLIED
                   Height NMTOKEN #IMPLIED>
```

```
<!ELEMENT Resize EMPTY>
<!ATTLIST Resize Anchor  NMTOKEN #IMPLIED
                 Fill    NMTOKEN #IMPLIED
                 WeightX NMTOKEN #IMPLIED
                 WeightY NMTOKEN #IMPLIED>

<!ELEMENT Insets EMPTY>
<!ATTLIST Insets Top    NMTOKEN #IMPLIED
                 Left   NMTOKEN #IMPLIED
                 Bottom NMTOKEN #IMPLIED
                 Right  NMTOKEN #IMPLIED>

<!ELEMENT Adapter (Parameter)*>
<!ATTLIST Adapter Class NMTOKEN #REQUIRED>

<!ELEMENT View (Parameter)*>
<!ATTLIST View Class NMTOKEN #REQUIRED>

<!ELEMENT Parameter EMPTY>
<!ATTLIST Parameter Name NMTOKEN #REQUIRED
                    Value NMTOKEN #IMPLIED>
```

## B.2 XML-Dokument

Dies ist ein Beispiel für ein XML-Dokument, das das Layout für die Dokumente im Bearbeitungswerkzeug des CoreMedia-Systems festlegt:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<!--
     this is a sample layout definition of the
     CoreMedia Content Publisher
-->

<!DOCTYPE DocTypes SYSTEM "../../lib/xml/corem-layoutdefs.dtd">

<DocTypes>
  <DocType Name="Author">
        <Property Name="Name" ShowInExplorer="true"/>
        <Property Name="Born" Label="Geboren">
           <View Class="my.DateView">
             <Parameter Name="Format" Value="DD.MM.YYYY"/>
           </View>
        </Property>
        <Property Name="Died">
           <Adapter Class="my.DateConverter">
             <Parameter Name="Format" Value="DD.MM.YYYY"/>
           </Adapter>
        </Property>
        <Property Name="Poems"
          <Position X="2"/>
        </Property>
```

```
        <Property Name="Picture">
          <Position Y="4"/>
          <Resize Anchor="CENTER"
                  Fill="BOTH"
                  WeightX="1.0"
                  WeightY="1.0"/>
          <Insets Top="10" Bottom="10"/>
        </Property>
        <Property Name="Curriculum Vitae"
          <Position Fill="BOTH"/>
                    WeightX="4.0"
                    WeightY="4.0"/>
        </Property>
  </DocType>

  <DocType Name="Poem">
        <Property Name="Title" ShowInExplorer="true"/>
        <Property Name="Author" IsVisible="false"/>
        <Property Name="Written" IsEditable"true"/>
        <View>
          <Parameter Name="Format" Value="DD.MM.YYYY"/>
        </View>
        <Property Name="Epoch"/>
        <Property Name="Verses"/>
  </DocType>
</DocTypes>
```

# Literaturverzeichnis


*Aho et al. 87:* Alfred V. Aho, Ravi Sethi und Jeffrey D. Ullmann. *Compilers: Principles, Techniques and Tools.* Addison-Wesley Publishing Company, 1987.

*Aksoy, Haralick 98:* Selim Aksoy und Robert M. Haralick. *Content-Bases Image Database Retrieval using Variances of Gray Level Spatial Dependencies.* In: Horace H.S. Ip und Arnold W.M. Smeulders (Hrsg.). *Proceedings of the IAPR International Workshop on Multimedia Information Analysis and Retrieval*, Lecture Notes in Computer Science, Bd. 1464, Hong Kong, China, August 1998, S. 3–19.

*Apple Computer Inc. 88:* Apple Computer Inc. *MacIntosh HyperCard.* The MacIntosh technical library. Addison-Wesley Publishing Company, Januar 1988.

*Berlin, O'Day 90:* Lucy M. Berlin und Vicki L. O'Day. *Platform and Application Issues in Multi-User Hypertext.* In: Simon Gibbs und Alex A. Verrijn-Stuart (Hrsg.). *Proceedings of the IFIP WG 8.4 Conference on Multi-User Interfaces and Applications*, Heraklion, Crete, Greece, September 1990, S. 293–309. North-Holland, Amsterdam New York Oxford Tokyo.

*Blittkowski 97:* Ralf Blittkowski. *Multimedia-Nuggets - Beim Geschäft mit Multimedia-Inhalten winken Milliardenumsätze.* c´t, Jg. 14, September 1997, Nr. 15, S. 128–131.

*Boehm 88:* Barry W. Boehm. *A Spiral Model of Software Development and Enhancement.* IEEE Transactions on Computers, Jg. 19, Mai 1988, Nr. 5, S. 61–72.

*Brockhaus 86: Brockhaus Enzyklopädie in vierundzwanzig Bänden.* Der große Brockhaus. F.A. Brockhaus, Mannheim, 19. Ausgabe, 1986.

*Cardelli, Wegner 85:* Luca Cardelli und Peter Wegner. *On Understanding Types, Data Abstraction, and Polymorphism.* Association for Computing Machinery Computing Surveys, Jg. 17, Dezember 1985, Nr. 4, S. 471–522.

*Chen 76:* Peter Pin-Shan Chen. *The Entity-Relationship Model – Toward a Unified View of Data.* ACM Transactions on Database Systems, Jg. 1, März 1976, Nr. 1, S. 9–36.

*Choi et al. 98:* Young-Il Choi, Yoo-Mi Park, Hun-Soon Lee und Seong-Il Jin. *An Integrated Data Model and a Query Language for Content-Based Retrieval of Video.* In: Sushil Jajodia, M. Tamer Özsu und Asuman Dogac (Hrsg.). *Proceedings of the 4th International Workshop on Advances in Multimedia Information Systems*, Lecture Notes in Computer Science, Bd. 1508, Instanbul, Turkey, September 1998, S. 192–198.

*Conklin 87:* Jeff Conklin. *Hypertext - An Introduction and Survey.* IEEE Transactions on Computers, Jg. 20, September 1987, Nr. 9, S. 17–41.

*Corders, Streitz 92:* Ralf Corders und Norbert Streitz (Hrsg.). *Hypertext und Hypermedia 1992: Konzepte und Anwendungen auf dem Weg in die Praxis; Fachtagung und Tutorien, München, 14.-16. September 1992*, München, Deutschland, September 1992. Gesellschaft für Informatik, Informatik aktuell.

*Denert 91:* Ernst Denert (Hrsg.). *Software-Engineering - Methodische Projektabwicklung.* Springer-Verlag, Berlin Heidelberg New York Tokyo, 1991.

*DIN31631 84: Kategorienkatalog für Dokumente - Begriffe und Gestaltung.* DIN 31631-1984, Deutsches Institut für Normung, 1984.

*Döbele-Martin 97:* Claudia Döbele-Martin. *Ergonomie-Prüfer: Handlungshilfe zur ergnomischen Gestaltung der Bildschirmarbeit*, Reihe Arbeit, Gesundheit, Umwelt, Technik, Bd. 35. Technologie-Beratungsstelle beim DGB Landesbezirk NRW, 1997.

*DuCharme 99:* Bob DuCharme. *XML: The Annotated Specification.* The Charles F. Goldfarb Series on Open Information Management. Prentice Hall, Upper Saddle River, New Jersey, Februar 1999.

*Ellis, Stroustrup 90:* Margaret A. Ellis und Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley Publishing Company, 1990.

*Floyd 95:* Christiane Floyd. *Theory and Practice of Software Development - Stages in a Debate.* In: P. D. Mosses, M. Nielsen und M. I. Schwartzbach (Hrsg.). *Theory and Practice of Software Development*, Lecture Notes in Computer Science. Springer-Verlag, Berlin Heidelberg New York Tokyo, 1995, S. 25–41.

*Gamma et al. 94:* Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides. *Design Patterns.* The Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, August 1994.

*Gamma 99:* Erich Gamma. „ULC - Eine Client/Server-Architektur für dünne Java Klienten". http://www.informatik.uni-hamburg.de/Info/Kolloquium/ WS98/gamma.html, Januar 1999.

*Gauss Interprise AG 99:* Gauss Interprise AG. „VIP 3.0". http://www.gauss.de/, Juli 1999. Broschüre und Datenblatt.

*Gibbs, Verrijn-Stuart 90:* Simon Gibbs und Alex A. Verrijn-Stuart (Hrsg.). *Proceedings of the IFIP WG 8.4 Conference on Multi-User Interfaces and Applications in Heraklion, Crete, Greece, 24-26 September 1990*, Heraklion, Crete, Greece, September 1990. IFIP WG 8.4, North-Holland, Amsterdam New York Oxford Tokyo.

*Goldfarb 95:* Charles F. Goldfarb (Hrsg.). *The SGML Handbook.* Clarendon Press, Oxford, Januar 1995.

*Grønbæk 88:* K. Grønbæk. *Rapid Prototyping with Fourth Generation Systems.* Technical Report DAIMI PB - 270, Computer Science Department, Aarhus University, 1988.

*Gulbins, Obermayr 99:* Jürgen Gulbins und Karl Obermayr. *Desktop-Publishing mit FrameMaker.* Springer-Verlag, Berlin Heidelberg New York Tokyo, Juni 1999.

*Heuer 92:* Andreas Heuer. *Objektorientierte Datenbanken.* Addison-Wesley Publishing Company, Juli 1992.

*Higher-Order Informations- und Kommunikationssysteme GmbH 99:* Higher-Order Informations- und Kommunikationssysteme GmbH. „CoreMedia 3.0". http://www.higher-order.de/, August 1999. 25 HTML-Pages.

*Hülser, Weber 92:* Christoph Hülser und Anja Weber. *The Individualised Electronic Newspaper: An Application Challenging Hypertext Technology.* In: Ralf Corders und Norbert Streitz (Hrsg.). *Proceedings of the Hypertext and Hypermedia 1992*, Gesellschaft für Mathematik und Datenverarbeitung (GMD), Integrated Publication and Information Systems (IPSI), Dolivostr. 13, D-6100, Darmstadt, September 1992, S. 62–74.

*Hopcroft, Ullman 93:* John E. Hopcroft und Jeffrey D. Ullman. *Einführung in die Automatentheorie, formale Sprachen und Komplexitätstheorie.* Internationale Computer-Bibliothek. Addison-Wesley Publishing Company, 2. Ausgabe, März 1993.

*HTML-4.0 98: W3C Document Type Definition for the HyperText Markup Language version 4.0 as ratified by a vote of W3C member companies.* HTML 4.0, World Wide Web Consortium, April 1998. First published 18.12.1997, revised 24.04.1998, update to 4.01 at 24.12.1999.

*Imperia 99:* Imperia. „Online Redaktionssystem 3.0". http://www.imperia.de/, Juli 1999. 5 HTML-Pages.

*Inprise Corp. 99:* Inprise Corp. „JBuilder 3.0". http://www.inprise.com/jbuilder/, August 1999. 20 HTML-Pages.

*ISO10179 96: Information Processing - Text and Office Systems - Document Style Semantics and Specification Language (DSSSL).* ISO 10179-1996 (E), International Organization for Standardization, 1996.

*ISO8613 89: Information Processing - Text and Office Systems - Office Document Architecture (ODA).* ISO 8613-1989 (E), International Organization for Standardization, 1989.

*ISO8879 86: Information Processing - Text and Office Systems - Standardized General Markup Language (SGML).* ISO 8879-1986 (E), International Organization for Standardization, 1986.

*ISO9075 87: Information Processing Systems - Database Language (SQL).* ISO 9075-1987 (E), International Organization for Standardization, 1987.

*Jajodia et al. 98:* Sushil Jajodia, M. Tamer Özsu und Asuman Dogac (Hrsg.). *Advances in Multimedia Information Systems: Proceedings of the 4th International Workshop, MIS´98 in Instanbul, Turkey, September 1998.* Springer-Verlag, Berlin Heidelberg New York Tokyo, September 1998.

*Jessen, Valk 86:* Eike Jessen und Rüdiger Valk. *Rechensysteme - Grundlagen der Modellbildung.* Studienreihe Informatik. Springer-Verlag, Berlin Heidelberg New York Tokyo, Juni 1986.

*Khoshafian et al. 92:* Setrag Khoshafian, Brad A. Baker, Razmik Abnous und Kevin Shepherd. *Intelligent Offices - Object-Oriented Multi-Media Information Management in Client/Server Architectures.* The Wiley Professional Computing Series. John Wiley & Sons, Inc., Juli 1992.

*Knepper 98:* Chris Knepper. *MacApp Documentation.* Apple Computer Corporation, Juni 1998.

*Krasner, Pope 88:* Glen E. Krasner und Stephen T. Pope. *A cookbook for using the Model-View-Controller User Interface Paradigm in Smalltalk-80.* Journal of Object-Oriented Programming, Jg. 1, 1988, Nr. 3.

*Krönert 88:* Günter Krönert. *Genormte Austauschformate für Dokumente.* Informatik-Spektrum, Jg. 11, April 1988, Nr. 2, S. 71–84.

*Lewis 95:* Ted Lewis. *Object-oriented Application Frameworks.* Manning Publications Co., Greenwich, November 1995.

*Linton et al. 87:* Mark A. Linton, Paul R. Calder und John M. Vlissides. *InterViews: A C++ Graphical Interface Toolkit.* Technical report, Center for Integrated Systems, Stanford University, Stanford, California 94305, November 1987.

*Lippert 97:* Martin Lippert. *Konzeption und Realisierung eines GUI-Frameworks in Java nach der WAM-Metapher.* Diplomarbeit, Fachbereich Informatik, Universität Hamburg, Germany, November 1997.

*Lockemann, Schmidt 87:* Peter C. Lockemann und Joachim W. Schmidt (Hrsg.). *Datenbank-Handbuch.* Springer-Verlag, Berlin Heidelberg New York Tokyo, März 1987.

*Lucarella et al. 92:* Dario Lucarella, Jocelyne Nanard, Marc Nanard und Paolo Paolini (Hrsg.). *European Conference on Hypertext and Hypermedia 1992 proceedings: Milano, Italy, November 30 - December 4, 1992.* Association of Computing Machinery, Special Interest Group on Hypertext, Special Interest Group on Information Retrieval, Special Interest Group on Office Information Systems, Association for Computing Machinery, November 1992.

*Lumbrecht 93:* J.L. Lumbrecht. *SRQ Leitfaden zur Normenreihe SN IN 2900/9000.* Bihrs, Hamburg, 1993.

*Manthey 91:* Rainer Manthey. *Declarative Languages – Paradigm of the Past or Challenge of the Future?* In: *Proceedings of the Kiev East/West Workshop on Next Generation Database Technology*, Lecture Notes in Computer Science, Bd. 504, April 1991.

*Mekelburg 99:* Florian Mekelburg. *Konzeption einer Benutzungsoberfläche zur Erstellung von DSSSL Style Sheets für SGML-Dokumente.* Diplomarbeit, Fachbereich Informatik, Universität Hamburg, Germany, März 1999.

*Meyer-Wegener 91:* Klaus Meyer-Wegener. *Multimedia-Datenbanken.* B.G. Teubner, Stuttgart, Februar 1991.

*Nelson 81:* Brent J. Nelson. *Remote Procedure Call.* Dissertation, Department of Computer Science, Carnegie-Mellon University, 1981.

*NetBeans Inc. 00:* NetBeans Inc. „NetBeans Developer 2.0 Professional Edition". http://www.netbeans.com/developer2/, Januar 2000. 10 HTML-Pages.

*Orfali et al. 96:* Robert Orfali, Dan Harkey und Jeri Edwards. *The Essential Distributed Objects Survival Guide.* John Wiley & Sons, Inc., Januar 1996.

*Palay 88:* Andrew J. Palay. *The Andrew Toolkit: An overview.* In: *Proceedings of the 1988 Winter USENIX Technical Conference, Dallas, Texas, February 1988*, Februar 1988.

*Pope 98:* Alan Pope. *The CORBA Reference Guide: Understanding the Common Object Request Broker Architecture.* Addison-Wesley Publishing Company, 1998.

*Quint, Vatton 92:* Vincent Quint und Irène Vatton. *Combining Hypertext and Structured Documents in Grif.* In: *Proceedings of the European Conference on Hypertext and Hypermedia 1992*, IMAG-INRIA, 2, rue de Vignate, F-38610 Gières, France, November 1992, S. 23–32.

*Rice et al. 95:* James Rice, Adam Farquhar, Philippe Piernot und Thomas Gruber. *Lessons Learned Using the Web as an Application Interface.* Technical report, Knowledge Systems Lab., 701 Welch Rd, bldg. C, Palo Alto, CA 94304, September 1995.

*Rumbaugh et al. 99:* James Rumbaugh, Ivar Jacobsen und Grady Booch (Hrsg.). *The Unified Methode Language reference manual.* The Addison-Wesley Object Technology Series. Addison-Wesley Publishing Company, 1999.

*Scheer 97:* August-Wilhelm Scheer. *Wirtschaftsinformatik: Referenzmodell für industrielle Geschäftsprozesse.* Springer-Verlag, Berlin Heidelberg New York Tokyo, 1997.

*Schreiber 94:* Siegfried Schreiber. *Specification and Generation of User Interfaces with the BOSS-System.* Technical report, Institute of Computer Science, Munich University of Technology, Arcisstr. 21, 80290 Munich, Germany, Oktober 1994.

*Sean Cotter 95:* Mike Potel Sean Cotter. *Inside Taligent Technology.* Addison-Wesley Publishing Company, Februar 1995.

*Srihari, Zhang 98:* Rohini K. Srihari und Zhongfei Zhang. *Finding Pictures in Context.* In: Horace H.S. Ip und Arnold W.M. Smeulders (Hrsg.). *Proceedings of the IAPR International Workshop on Multimedia Information Analysis and Retrieval*, Lecture Notes in Computer Science, Bd. 1464, Hong Kong, China, August 1998, S. 109–123.

*Steffens 97:* Ulrike Steffens. *Information Retrieval: Grundlagen, Systeme und Integration.* Forschung und Praxis, Jg. 21, Januar 1997, Nr. 1.

*Stevens, Wright 95:* W. Richard Stevens und Gary R. Wright. *TCP/IP Illustrated, Volume 2 - The Implementation.* The Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, März 1995.

*Stevens 96:* W. Richard Stevens. *TCP/IP Illustrated, Volume 3 - TCP for Transactions, HTTP, NNTP, and the UNIX Domain Protocols.* The Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, April 1996.

*Sun Microsystems Inc. 96a:* Sun Microsystems Inc. *Java Database Connectivity.* Sun Microsystems Inc., 2550 Garcia Ave, Mtn. View, CA 94043-1100 USA, version 1.10 from 21st october 1996. Ausgabe, Oktober 1996.

*Sun Microsystems Inc. 96b:* Sun Microsystems Inc. *Java Overview.* Technical report, Sun Microsystems Inc., 2550 Garcia Ave, Mtn. View, CA 94043-1100 USA, Januar 1996.

*Sun Microsystems Inc. 97a:* Sun Microsystems Inc. *Java Developer Kit.* Sun Microsystems Inc., 2550 Garcia Ave, Mtn. View, CA 94043-1100 USA, version 1.1 from january 1997. Ausgabe, Januar 1997.

*Sun Microsystems Inc. 97b:* Sun Microsystems Inc. *JavaBeans.* Sun Microsystems Inc., 2550 Garcia Ave, Mtn. View, CA 94043-1100 USA, version 1.0 from 1st december 1997. Ausgabe, Dezember 1997.

*Sun Microsystems Inc. 99a:* Sun Microsystems Inc. *Java Foundation Classes (aka Swing).* Sun Microsystems Inc., 2550 Garcia Ave, Mtn. View, CA 94043-1100 USA, version 1.1.1 from 6th april 1999. Ausgabe, April 1999.

*Sun Microsystems Inc. 99b:* Sun Microsystems Inc. *Java Server Pages.* Sun Microsystems Inc., 2550 Garcia Ave, Mtn. View, CA 94043-1100 USA, version 1.0 from 27th september 1999. Ausgabe, September 1999.

*Sun Microsystems Inc. 99c:* Sun Microsystems Inc. „Java Workshop 3.0 Professional“. http://www.sun.com/workshop/java/, Oktober 1999. 15 HTML-Pages.

*Swick, Weissman 88:* Ralph R. Swick und Terry Weissman. *X Toolkit Widgets - C Language Interface.* Digital Equipment Corporation, März 1988.

*Tolksdorf 96:* Robert Tolksdorf. *Die Sprache des Web: HTML 3.* dpunkt-Verlag für digitale Technologie GmbH, Mai 1996.

*Tsichritzis 85:* Dennis Tsichritzis. *Office Automation.* Topics in Information Systems. Springer-Verlag, Berlin Heidelberg New York Tokyo, März 1985.

*Vignette Inc. 99:* Vignette Inc. „StoryServer 4.0“. http://www.vignette.com/, Januar 1999. Broschüre und Datenblatt.

*Vlissides, Linton 90:* John M. Vlissides und Mark A. Linton. *Unidraw: A Framework for Building Domain-Specific Graphical Editors*, Topics in Information Systems, Bd. 8, S. 237–268. Springer-Verlag, Berlin Heidelberg New York Tokyo, 3. Ausgabe, Juli 1990.

*Vogel, Duddy 97:* Andreas Vogel und Keith Duddy. *Java Programming with CORBA.* John Wiley & Sons, Inc., März 1997.

*Voss, Nentwig 98:* Josef Voss und Dietmar Nentwig. *Entwicklung von graphischen Benutzungsschnittstellen: Modelle, Techniken und Werkzeuge der User-Interface-Gestaltung.* Hanser Verlag, München, Oktober 1998.

*Waldo et al. 97:* Jim Waldo, Geoff Wyant, Ann Wollrath und Sam Kendall. *A Note on Distributed Computing.* In: Jan Vitek und Christian Tschudin (Hrsg.). *Mobile Object Systems: Towards the Programmable Internet*, Lecture Notes in Computer Science, Bd. 1222. Springer-Verlag, Berlin Heidelberg New York Tokyo, Heidelberg, April 1997, S. 49–64.

*Watt 90:* David A. Watt. *Programming Language Concepts and Paradigms.* Prentice Hall International Series in Computer Science. Prentice Hall, Englewood Cliffs, New Jersey, Oktober 1990.

*Weinand et al. 88:* Andre Weinand, Erich Gamma und Rudolf Marty. *ET++ - An Object-oriented Framework for interactive Data Graphics.* In: *Proceedings of the ACM OOPSLA 1998 Conference, San Diego, CA, September 1988*, September 1988.

*Wessel 98:* Ivo Wessel. *GUI-Design: Richtlinien zur Gestaltung ergonomischer Windows-Applikationen.* Hanser Verlag, München, November 1998.

*Wilhelm, Heckmann 96:* Reinhard Wilhelm und Reinhold Heckmann. *Grundlagen der Dokumentenverarbeitung.* Addison Wesley Longman Verlag GmbH, August 1996.

*Wöhe 81:* Günter Wöhe. *Einführung in die Allgemeine Betriebswirtschaftslehre.* Vahlens Handbücher der Wirtschafts- und Sozialwissenschaften. Verlag Vahlen, September 1981.

*Zschau 00:* Oliver Zschau. *Web Content Management - Einführung, Konzepte, Software.* Diplomarbeit, Fachbereich Medien- und Elektrotechnik, University of Applied Sciences Mittweida (FH), Germany, Januar 2000.

## Erklärung

Hiermit erkläre ich, daß ich die vorliegende Diplomarbeit selbständig durchgeführt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Hamburg, den 28. August 2000

(Christian Pesch)