Aus dem Institut für Informationssysteme
der Medizinischen Universität zu Lübeck
Direktor:
Prof. Dr. rer. nat. Volker Linnemann

Ein XML-basiertes Modell zur Modellierung und Strukturierung von Informationen im Kontext eines Medienarchivs für die Lehre

Inauguraldissertation
zur
Erlangung der Doktorwürde
der Medizinischen Universität zu Lübeck
- Aus der Technisch-Naturwissenschaftlichen Fakultät -

Vorgelegt von
Ralf Behrens
aus Hohenhameln

Lübeck, November 2000

Berichterstatter:	Prof. Dr. rer. nat. V. Linnemann
	Prof. Dr. rer. nat. D. Hogrefe
Tag der mündlichen Prüfung:	23. Januar, 2001
Vorsitz des Prüfungsausschusses:	Prof. Dr. rer. nat. B. Fischer
zum Druck genehmigt:	23. Januar, 2001

gezeichnet Prof. Dr. rer. nat. W. Dosch
Dekan der Technisch-Naturwissenschaftlichen Fakultät.

Zusammenfassung

Die Integration und Beschreibung großer Mengen von unterschiedlich stark strukturierten Daten ist eine aktuelle Herausforderung für die Datenbankwelt. In vielen Bereichen werden lediglich speziell zugeschnittene ad-hoc Lösungen verwendet. Es fehlt an allgemeinen Konzepten, welche den universellen Einsatz und die Wiederverwendbarkeit einmal entworfener Lösungen gewährleisten.

Bei der Integration und Beschreibung der anfallenden Daten fällt auf, dass diese oftmals eine inhärente Struktur besitzen. Daher liegt es nahe, diese Struktur entsprechend zu verwalten. Da die auftretenden Strukturen in der Regel tief geschachtelt, unregelmäßig oder auch implizit vorhanden sind, erweisen sich konventionelle Datenmodelle für deren Beschreibung als unzureichend. Als geeignete Alternative stellt sich die Extensible Markup Language (XML) dar, die zunächst im Internet als globale Datenaustauschsprache ihren Ursprung hatte.

Im Rahmen dieser Arbeit werden XML Document Type Definitions (DTDs) bezüglich ihrer formalen Eigenschaften untersucht. Ausgehend von dieser Analyse wird eine Erweiterung definiert, die DTDs zur Beschreibung von Schemata tauglich machen. Die vorgestellte Analyse und Erweiterung kann über diese Arbeit hinaus als Grundlage für XML-basierte Modelle dienen. Ausgehend von der XML-Erweiterung wird ein Modell zur Beschreibung von Objekten, Objektstrukturen und -repositorien entwickelt. Bei der praktischen Umsetzung des Modells werden neben Anfrageoperationen auch Migrations- und Restrukturierungsmöglichkeiten betrachtet.

Die Arbeit mündet in der Erstellung eines Prototypen, der zur Unterstützung von Forschung und Lehre an der Medizinischen Universität zu Lübeck einem breiten Anwenderfeld zu Gute kommt. Dieser Prototyp verifiziert die theoretischen Ergebnisse dieser Arbeit und zeigt deutlich den praktischen Nutzen auf, den die entwickelten Konzepte dem Benutzer liefern. Es stellt sich heraus, dass XML in Kombination mit angrenzenden Standards als Basis für die Strukturierung, Klassifikation und inhaltliche Beschreibung im Rahmen eines Medienarchivs eine wesentliche Verbesserung und Flexibilisierung gegenüber herkömmlichen Datenmodellen darstellt.

Abstract (english)

The integration and description of vast amounts of semistructured data presents a real challenge to the database community. In general only proprietary solutions are applied. There is a lack of common concepts which would provide the universal deployment and reusability of once defined solutions.

We can assume that the information usually bears an inherent structure. Therefore, it suggests itself to try to manage this information efficiently. Since the hierarchical structures are often deep, irregular or implicit, conventional data models seem to be insufficient for this problem. The Extensible Markup Language (XML), which was first used in the internet area, seems to be a promising alternative.

In this thesis XML Document Type Definitions (DTDs) are investigated closely, specifically focusing on their formal properties. Based on this analysis an extension is proposed, making DTDs more appropriate for schema definition. The presented analysis and extension can be used beyond this thesis as a formal framework for XML-based modeling. Starting from this extension, a model for describing objects, object structures and repositories is designed. Beyond the practical implementation, searching and restructuring issues are considered.

This thesis leads to the development of a prototype, which is used for research and lectures in the Medical University of Luebeck. The prototype verifies the theoretical results presented here and shows clearly the benefit of the new concepts provided to the user. It is shown that, in contrast to conventional models, the combination of XML with other standards greatly improves the structuring, classification and textual description in the context of a media archive.

Danksagung

Die vorliegende Arbeit entstand während meiner knapp vierjährigen Tätigkeit als wissenschaftlicher Mitarbeiter am Institut für Informationssysteme der Medizinischen Universität zu Lübeck unter der Leitung von Prof. Dr. Volker Linnemann.

Ich möchte an dieser Stelle allen danken, die mir bei der Erstellung dieser Arbeit mit Rat und Tat zur Seite gestanden haben. Als erstes danke ich meinem Betreuer Prof. Dr. Volker Linnemann, der stets ein kompetenter Diskussions- und Ansprechpartner war. Gleichermaßen danke ich Prof. Dr. Dieter Hogrefe für seine Bereitschaft zur Übernahme des Co-Referats.

Darüber hinaus danke ich den Mitgliedern der Projektgruppe Medienarchivserver für die fruchtbaren Anregungen und Diskussionen. Ebenso danke ich meinem ehemaligen Kollegen Carsten Lecon für die geleistete Vorarbeit am Medienarchiv. Allen Studenten und Hilfskräften, deren Anstrengungen und Arbeiten in das Projekt eingeflossen sind, sei hiermit gedankt. Besonders sei dabei M. Kreutzberger erwähnt, der einen nicht unerheblichen Anteil der Implementierung realisiert hat. PD Dr. Gerhard Buntrock und Dr. Oliver Schmitt danke ich für die wissenschaftliche Zusammenarbeit, die in entsprechenden Publikationen mündete.

All denjenigen, die durch intensives Korrekturlesen verbunden mit fruchtbaren Diskussionen die Arbeit zu verbessern halfen, bin ich zu innigstem Dank verpflichtet. Insbesondere sei unsere Sekretärin Angela König erwähnt, welche stark zur orthographischen Verbesserung dieser Arbeit beigetragen hat.

Schließlich gilt mein besonderer Dank meiner Freundin Julia und meinen Eltern für deren mir jederzeit zu Teil gewordene Unterstützung.

Lübeck, im November 2000 Ralf Behrens

Inhalt

1 Einleitung

„Ich habe das ganze Land bereist und mit allen Experten gesprochen, und ich sage Ihnen: Datenverarbeitung ist ein Mode-Fimmel, der nicht einmal das Jahr überstehen wird.“

DER LEKTOR FÜR WIRTSCHAFTSBÜCHER BEIM RENOMMIERTEN VERLAG PRENTICE HALL, 1957

Die Verwaltung von textuellen und medialen Informationen ist eine Aufgabe, die in vielen Gebieten der heutigen Zeit eine zentrale Bedeutung erfährt. Im industriellen Bereich hängt der Erfolg oder Mißerfolg unternehmerischer Entscheidungsfindung maßgeblich von der Verfügbarkeit des relevanten Informationsmaterials ab. Im universitären und forschungsorientierten Umfeld ist ein angemessener und zügiger Austausch von Lehr- und Forschungsmaterialien die Grundlage für konstruktive und zukunftsorientierte Arbeit. Doch gerade im Universitätsbereich stehen dem interdisziplinären Austausch verschiedener Institutionen oft infrastrukturelle Hemmnisse entgegen. Dieses ist umso verwunderlicher, als doch der Austausch der Ergebnisse unterschiedlicher Forschungsdisziplinen die Grundlage von fruchtbaren Synergien bildet.

An der Medizinischen Universität zu Lübeck wurde 1997 im Rahmen des *MESH* [MESa] die Errichtung eines Medienarchivservers beschlossen. Dieser *Medienarchivserver* („Multimedia Online Teaching and Research Archive“ (MONTANA)) soll der Archivierung und Verwaltung unterschiedlichster, im Universitätsumfeld entstehender, Mediendaten dienen. Die Archivierung von Medienobjekten und die diese beschreibende Information muss dabei deren große Datenvolumina, die mit hohen Datenraten bei der Übertragung und Präsentation einhergehen, sowie deren wesentlich höheren semantischen Gehalt berücksichtigen. Insofern stellen sich traditionelle Datenbanktechniken, welche im Wesentlichen die Speicherung kurzer Datensätze mit eng umrissener Bedeutung unterstützen, als nicht ausreichend heraus.

Bei der Definition der Anforderungen an das Medienarchiv wurden folgende zentrale Funktionalitäten vereinbart [HSP97]:

- Das Suchen von medialen und multimedialen Informationen nach formalen Kriterien und inhaltlichen (wahrnehmerischen) Gesichtspunkten ist eine dem Archivieren inhärente Aufgabe. Hier ist die Integration von Verfahren der Inhaltserschließung von Medienobjekten in den Archiv-Server erforderlich.
- Die Darbietung medialer Information durch Teleteaching-Anwendungen macht die Bereitstellung entsprechender Lade- und Übertragungsbandbreiten für den Datentransport notwendig. Besonders bei zeitbehafteten Medien (Audio und Video) ist eine isochrone Übertragung von Bedeutung.
- Es sollten je nach Bedarf verschiedene „Qualities of Service“ angeboten werden. Beispielsweise kann Bildmaterial in unterschiedlichsten Auflösungen und Qualitäten angefragt werden.
- Zur Realisierung telekooperativer Aktivitäten ist es wichtig, das Medienarchiv mehrbenutzerfähig und netzweit zugreifbar zu machen.
- Das Archiv soll als Schnittstelle zwischen der Medizin und der Informatik fungieren und so den Austausch zwischen diesen Bereichen verbessern helfen.

Die Entwicklung eines Medienarchivs für Forschung und Lehre erfordert das Verständnis und die Anwendung von Wissen und Ergebnissen unterschiedlichster Forschungsbereiche wie z. B. Multimedia-Datenbanken, Anfragesprachen, Anfrageoptimierung, textuelle Informationsgewinnung, strukturierte Dokumente u. v. m. Es scheint daher kaum möglich, die ganze Fülle der Fragestellungen in einer Arbeit abzuhandeln. Die vorliegende Arbeit soll aus dem breiten Spektrum dieses Gebietes den Aspekt der adäquaten Beschreibung und Strukturierung der anfallenden Informationen behandeln. Die Beschreibung medialer Informationen und deren Verwaltung schlägt dabei die Brücke zum Gebiet der textuellen Informationsverarbeitung.

1.1 Kontext der Arbeit

Die Entwicklung geeigneter Sprachelemente, sowohl zur Klassifikation von hierarchischen Strukturen als auch zur Beschreibung textueller Objekte, bedarf der Betrachtung unterschiedlicher Aspekte der heutigen Datenbankwelt. Ziel unseres Modells ist es dabei, die Vorteile bestehender Ansätze zu vereinen. Den Kontext dieser Arbeit bilden Projekte in den Bereichen der multimedialen Systeme und der strukturierten Dokumente.

Medienarchive / Multimedia-Datenbanksysteme: Das erste Referenzmodell für die Konzipierung von Hypermedia-Systeme war die *Hypertext Abstract Machine* (HAM) in [CG88]. Es beschreibt eine Client/Server-Architektur für Hypermedia-Systeme basierend auf einer Unterteilung in drei Schichten (externe Sicht, logische Sicht, physikalische Ebene), wie sie auch bei allgemeinen Informationssystemen vorgenommen wird. Unser Medienarchiv MONTANA genügt im Wesentlichen dieser Schichteneinteilung.

Das *Dexter-Referenzmodell* kann als Verfeinerung der HAM gesehen werden [HS94]. Der wesentliche Vorteil dieses Systems liegt in dessen erweitertem Linkkonzept und in der Möglichkeit, komplexe Komponenten definieren zu können. Es enthält neben bidirektionalen Links auch dynamische Links, n:m-Links und Links auf Links. Der Linkmechanismus kommt in ähnlicher Weise auch im Archiv MONTANA zur Anwendung, wobei insbesondere der Sicherung der Linkkonsistenz zentrale Bedeutung zukommt.

Das Projekt *PRIME* [BMB95, MMFB97] befasst sich mit der Speicherung von medizinischen Daten. Zielgruppe sind Studenten und Ärzte, die Krankheiten mithilfe dieses Archivs analysieren wollen. Die Daten sind strukturierte Dokumente, wobei Patientendaten und diesen zugeordnete MR-Bilder (Magnetresonanz-Bilder) mittels einer kontextfreien Grammatik beschrieben sind. Zum Zugriff auf den Datenbestand wird ein WWW-Browser verwendet. Das *Browsing* erfolgt anhand der internen Struktur der Dokumente. Die deskriptive Suche bezieht sich auf den Patientennamen, die Patienten-Id oder auf den Inhalt von Untersuchungen oder MR-Bildern. Eine Integration der Struktur und inhaltlichen Beschreibung von MR-Bildern erlaubt strukturbezogene Anfragen z. B. der Form „A enthält B“ oder „A grenzt an B“. Die Mediendaten befinden sich, entsprechend der durch die Grammatik definierten Struktur, in einem mittlerweile nicht mehr erhältlichen objektorientierten Datenbanksystem (O_2).

Im *Dermatologischen Atlas* [BKPD95] wird dermatologisches Bildmaterial bereitgestellt. Zielgruppe sind Ärzte, die Unterstützung bei einer dermatologischen Diagnose suchen, Studenten in der medizinischen Ausbildung, sowie Patienten. Die symbolische Beschreibung der Bilder erfolgt anhand vorgegebener Attribute, die hierarchisch aufgebaut sind (eine Suche nach einer Hand wird auch Finger- und Fingernägel-Erkrankungen berücksichtigen). Die Suche erfolgt durch Auswahl von Attributwerten aus Listeneinträgen ausgewählter Kategorien (Lokation, Alter, etc.). Die Eingabe neuer Daten ist nur für privilegierte Personen möglich. Alle Daten befinden sich in einer relationalen Datenbank (MS-Access [MSA]).

Das Projekt *AMPHORE* [Sül96] dient zur Speicherung und Verwaltung von Filmen. Die Zielgruppe sind Film-Dokumentare. Der Zugriff erfolgt über ein eigenentwickeltes Dokumentations-Tool und ein Recherche-Werkzeug. Die Filmdaten werden durch eine Beschreibung im Format der *Standard Generalized Markup Language* (SGML) [Gol90] strukturiert. Inhaltliche Merkmale bestehen aus filmographischer Information, Verweise auf die Filmdatei, ggf. Verweise auf Audio- Begleittext, textuelle Begleitinformation und sequenzbasierte Beschreibungen des Filminhalts. Freitexte werden durch Subjekt-Prädikat-Objekt-Konstrukte beschrieben, wobei die Suchqualität durch Verwendung zweier Thesauri (für Subjekte/Objekte und Prädikate) erhöht wird. Die eigentlichen Filme werden als Dateien in einem Dateisystem gespeichert, während die inhaltliche Information in einer Volltext- Datenbank abgelegt ist.

MeDoc [Dre95, BDLM96] hat als Ziel, Informatiker -insbesondere in der Wissenschaft tätige- bei der Fachliteratur-Recherche zu unterstützen. Es gibt verschiedene Kategorien, nach denen Objekte mit jeweils unterschiedlichen Attributen beschrieben werden können. Dabei können die Attribute (nicht die -werte) vererbt werden, indem eine Kategorie als Spezialisierung einer anderen definiert wird. Die Mediendaten (in diesem Fall Fachliteratur) befinden sich an verschiedenen Orten (Datenbanken oder WWW-Dokumente), über sogenannte *broker* wird eine einheitliche Schnittstelle zu allen Datenquellen geschaffen, sogenannte *agents* stellen die Beziehung zu den Benutzern her. Der Zugriff auf den Datenbestand erfolgt mittels eines WWW-Browsers über das Internet.

Hyperwave [Mau96] ist im Gegensatz zu den bisherigen Projekten nicht für eine bestimmte Anwendung konzipiert, sondern stellt eine flexible Datenbank mit WWW-Schnittstelle bereit. Nachdem zuerst spezielle Programme (Amadeus und Harmony) für die Benutzung der von Hyperwave verwalteten Daten nötig waren, ist der Zugang auf den Datenbestand über ein Gateway inzwischen mit herkömmlichen WWW-Browsern möglich. Zur Klassifikation von Objekten lassen sich sogenannte Kollektionen definieren, zur symbolischen Beschreibung der Objekte lassen sich Attribute (aus einer vordefinierten Menge von Attributen) den Kollektionen zuordnen. Die Daten werden getrennt nach Objekten und Beziehungen zwischen Objekten in einem eigenentwickelten Datenbanksystem verwaltet, zusätzlich gibt es einen Volltext-Server zur Speicherung von invertierten Indexen aus Wort-Listen.

Im Gegensatz zu den vorgestellten Projekten soll das Medienarchiv nicht auf einen bestimmten Datentyp (medizinische Daten in PRIME und dem Dermatologischen Atlas, Filmdaten in AMPHORE und Literaturdaten in MeDoc) beschränkt sein. Außerdem soll die Klassifikation nicht vorgegeben sein. Im Projekt in PRIME erfolgt dies durch eine einmal definierte kontextfreie Grammatik und bei AMPHORE ebenso durch eine SGML *Document Type Definition* (DTD) (s. Abschnitt 2.2.1).

Eine benutzerdefinierte Wahl von Attributen, die der Klassifikation entsprechen, ist in den Projekten PRIME, AMPHORE und dem Dermatologischen Atlas nicht möglich. In Hyperwave und MeDoc können zwar Strukturen definiert und diesen Strukturen Attribute zugewiesen werden, doch ist die Menge an möglichen Attributen eingeschränkt. Die automatische Berechnung von Attributwerten ist nicht vorgesehen, im Projekt PRIME jedoch angedacht [Chi97]. Das Medienarchiv MONTANA erlaubt dagegen die Annotierung von Medienobjekten mit einer beliebigen Anzahl an Attributen und deren (semi-) automatische Berechnung.

Verarbeitung textueller Informationen: Die Verarbeitung textueller Daten spielt in vielen Gebieten der Informatik eine wichtige Rolle. Bereiche wie Document-Processing, Software-Engineering oder Online-Informationssysteme profitieren von hier erzielten Forschungsergebnissen. Gerade die rasche Entwicklung des World Wide Web erhöhte das Interesse an strukturierten und semistrukturierten Daten drastisch. Obwohl die Menge an textuellen Informationen kaum noch überschaubar ist, werden Dokumente in der Regel nicht durch Datenbank-Managementsysteme verwaltet, sondern durch Werkzeuge wie Editoren, Textretrieval-Systeme (IRS) [Fuh96, Fer96] oder *Hypertext-Systeme* [DBHDV94]. Darüber hinaus werden diese oftmals strukturierten Dokumente in Filesystemen abgelegt. Die Möglichkeiten dieser Systeme sind insofern beschränkt, als dass sie im Wesentlichen navigierenden Zugriff erlauben und Datensuche und -manipulation über Pattern-Matching-Verfahren realisieren.

Dokumentverarbeitung schließt die Erstellung und Visualisierung von Dokumenten ein. Die meisten Dokumentverarbeitungssysteme enthalten daher eine Fülle verschiedener Werkzeuge, wie Editoren, Rechtschreibprüfer, Formatierer usw., um eine geeignete Präsentation auf unterschiedlichen Medien zu ermöglichen. Kommerzielle Dokumentverarbeitungssysteme versuchen ein Höchstmaß an Unabhängigkeit zu gewährleisten. Versucht der Benutzer aber Dokumente zwischen unterschiedlichen Systemen auszutauschen, wird er schnell erkennen, dass Dinge wie Fontgrößen, Randeinstellungen oder auch logische Gliederungen wie Kapitel und Abschnitte durch Transformationen verloren gegangen sind.

Datenbankbasierte Verwaltung textueller Informationen: Die gezielte Verwendung von Techniken aus dem Datenbankbereich kann bei der Verarbeitung von Dokumenten zu großen Erleichterungen führen. Datenbanksysteme dienen definitionsgemäß dem dauerhaften Speichern großer Datenmengen. Diesen Daten ist gemein, dass sie oftmals eine innere Struktur haben, die beim Speichern nicht berücksichtigt wird. In vielen Fällen stellen kontextfreie Grammatiken ein adäquates Hilfsmittel zur Beschreibung solcher Strukturen dar. Daher liegt es nahe, diese beiden Konzepte der Informatik für den Benutzer gewinnbringend zusammenzufassen. Das Anwendungsfeld kontextfreier Grammatiken ist groß. So lässt sich mit ihnen nicht nur die Syntax von Programmiersprachen definieren, sondern auch graphische Darstellungen beschreiben oder der strukturelle Aufbau von Dokumenten wie Büchern und Artikeln, aber auch Web-Seiten festlegen.

Da es bei der Umsetzung strukturierter Daten, insbesondere mithilfe von Grammatiken, zu tief geschachtelten Strukturen kommt, bietet sich das objektorientierte Datenmodell hierfür an. [CACS94] liefern einen Ansatz zum Speichern strukturierter Dokumente auf der Basis einer objektorientierten Datenbank. Allen bekannten Ansätzen ist jedoch gemein, dass sie bei der Umsetzung der Daten von der konkret vorliegenden Grammatik ausgehen. Dieses hat den Nachteil, dass oftmals Strukturen, die sich zwar in ihrer konkreten Form unterscheiden, jedoch die gleiche oder zumindest ähnliche abstrakte Syntax besitzen, individuell umgesetzt werden müssen.

Modellierung von strukturierten Daten: Grammatiken und Bäume wurden schon frühzeitig als geeignetes Hilfsmittel für die Modellierung von Datenstrukturen insbesondere im Übersetzerbau erkannt. Im Hinblick auf die Datenbankforschung haben sie jedoch erst in neuerer Zeit an Bedeutung gewonnen. Sie spielen für die Verarbeitung strukturierter und semistrukturierter Daten eine wichtige Rolle. [GT87] liefern einen Ansatz für die Verwaltung strukturierter Dokumente auf der Basis von Grammatiken. [GPVG89] beschreiben ein Konzept zur Repräsentation allgemeiner hierarchischer Strukturen auf der Basis von Bäumen. Es werden Möglichkeiten der Baummanipulationen mithilfe einer Algebra und eines Kalküls angesprochen. [Lin93] zeigt die Integration von Grammatiken und Bäumen in ein reales Datenmodell und liefert darüber hinaus Operationen für deren praktische Verwendung.

Konkrete und abstrakte Grammatik: Der Zusammenhang zwischen konkreter und abstrakter Syntax spielt bei der Entwicklung syntax-basierter Editoren [Sch95] eine Rolle. [BBG88] zeigen Möglichkeiten der Abstraktion im Hinblick auf Programmiersprachen. [KW95] stellen ein Werkzeug für die Transformation zwischen abstrakter und konkreter Grammatiken vor, wobei Grenzen bei der Einsetzbarkeit aufgezeigt werden. [Nij80] verfolgt frühzeitig ähnliche Überlegungen unter der Bezeichnung $Grammatical\ Covers$.

Hat man einmal die Struktur eines Objekts auf der Basis einer Grammatik bestimmt, so ergeben sich vielfältige Möglichkeiten. Zum einen können Anfragen gestellt werden, die neben dem Inhalt den logischen Aufbau berücksichtigen. Zum anderen können Transformationsregeln definiert werden, die der Umwandlung unterschiedlicher Formate dienen. Diese Regeln können ebenfalls in der Datenbank abgelegt werden. Für die Transformation zwischen Beschreibungsformaten gibt es eine Menge von ad-hoc-Lösungen. Es fehlt jedoch eine allgemeine Lösung, die theoretisch fundiert universelle Möglichkeiten zur Umwandlung bereitstellt.

Strukturgenerierung auf der Basis von Grammatiken: Auch die Erzeugung von Objekten vor dem Hintergrund einer wohldefinierten Grammatik erweckt durch den abstrakten Standpunkt neues Interesse. [Lin79] beschreibt allgemeine Sprachkonstrukte zur Erzeugung von Programmstrukturen. Hierfür wird die konkrete Grammatik der zugrunde liegenden Sprache verwendet. Für die Generierung ist es dabei notwendig durch vergleichsweise umständliche Operationen, wie automatische Operatoreinsetzung, grammatikalische Korrektheit zu erzwingen. Gerade diese Arbeit zeigt, dass eine Generierung auf der Basis einer abstrakten Grammatik große Erleichterungen bringen würde. Die sich künstlich aufblähenden Grammatiken würden stark vereinfacht und dem Anwender würde die Arbeit erleichtert werden. Die Beschreibung mit grammatikalischen Mitteln sollte sich dabei nicht nur auf die interne Struktur von Objekten beschränken, sondern auch deren hierarchische Anordnung repräsentieren. So können allgemeine Strukturen, wie sie in vielen Anwendungen vorkommen, durch Grammatiken modelliert werden [SL95]. Zusammenhänge, die meist in Objektschemata oder ER-Diagrammen formalisiert werden, lassen sich adäquat darstellen. Die interne und globale Sicht sollte hierfür integriert und unter einem gemeinsamen Modell betrachtet werden. Diese würde den Zugriff auf hierarchische Klassifikation und Dokumentteile mit identischen Mitteln erlauben.

Attributierte Grammatiken: [NVdB97, NVdB98, Nev98, CM95] untersuchen die Anwendung einer Teilklasse attributierter Grammatiken zur Anfrageformulierung. Die Beschreibung von Strukturen mithilfe von Grammatiken wird in [Lin93, GT87, GPVG89] behandelt. Zur Definition semantischer Beziehungen werden kontextfreie Grammatiken um Attributierungen erweitert [SL95]. In [LL98, BLLS98] wird dieser Ansatz auf die Beschreibung interner und globaler Eigenschaften angewendet.

Integration heterogener Daten: Die Modellierung multimedialer Objekte erfordert die Integration unterschiedlicher Paradigmen. Ein allgemeiner Ansatz zur Verschmelzung heterogener Datenbestände wird in [PAGM96] vorgestellt. Dieses Modell wird innerhalb des TSIMMIS-Projekts [GMQR+97] eingesetzt. Hier werden verschiedene Modelle auf der Basis eines Objektgerüsts integriert.

Anfragen an das Web: Die Wahl des geeigneten Datenmodells erweist sich als die zentrale Fragestellung im Hinblick auf die Entwicklung geeigneter Anfrageoperationen. Im Hinblick auf die Modellierung des Webs betrachten Systeme der ersten Generation [MMM96, KS97, LSS96, GLS98] das WWW als eine große Datenbank. Meist werden hierfür Graphenmodelle verwendet, welche als einziges internes Merkmal die Verbindung von Seiten über *Links* berücksichtigen.

Alternativ werden WWW-Seiten als einzelne Objekte modelliert, wobei oftmals Grammatikansätze [GT87, GPVG89, ACM93, ACM95, ACC+97, ACM98] zur Anwendung kommen. Die Umsetzung in semantisch reichhaltigere Strukturen führt aber schnell zu wichtigen Unentscheidbarkeitsaussagen, die die beschriebenen Verfahren in der Praxis unbrauchbar werden lassen. [AM98] geht auf die interne Struktur von Web-Seiten ein. Die Anfragesprache gestaltet sich jedoch vergleichsweise kompliziert und lässt kaum typsichere Transformationen zu. In [Beh97] wird geklärt, inwieweit sich Anfragen an Objekte auf lokale Bereiche einschränken lassen. Hierzu wird ein vereinfachtes objektorientiertes Datenmodell entwickelt.

Der von uns entwickelte Ansatz nimmt eine Zwitterstellung ein. Er bietet einen flexiblen und hochgradig skalierbaren Beschreibungsmechanismus, der es erlaubt, einzelne Objekte zu strukturieren, Objektmengen mit ihren Beziehungen darzustellen oder ganze Objektrepositorien (wie z. B. das WWW) umfassend zu modellieren.

(Semi-) strukturierte Daten: Daten können unterschiedlich stark strukturiert sein. So reicht die Bandbreite von strukturierten Informationen (ODMG, RDBMS), über semistrukturierte Daten (BiBTeX, HTML, XML), bis hin zu unstrukturierten Objekten (Image-, Audio-Dateien). [ACM97] geht auf die Verwandtschaft unterschiedlicher Datenformate ein. Über die Definition von Korrespondenzen werden Transformationen abgeleitet. [CDSS98, SC98] liefern einen Ansatz, um solche Daten im Mediator-Kontext zu integrieren und zu transformieren. [Clu97] beschreibt Möglichkeiten, Anfragen über semistrukturierte Daten zu formulieren. [CACS94] liefern einen Ansatz zum Speichern von strukturierten Dokumenten auf der Basis einer objektorientierten Datenbank.

Extraktion von (Meta-)Informationen: Textuelle Informationen werden sehr häufig anhand vordefinierter Strukturen aufgebaut. So werden viele dynamisch erzeugte HTML-Seiten durch schablonenartige Vorgaben spezifiziert (entweder statisch oder als Ergebnis einer Anfrage). Anderseits haben Seiten innerhalb eines Servers oftmals einen vorgegebenen äußeren Rahmen. Durch die Beschreibung der inneren Struktur eines Objekts können einerseits Daten gezielt extrahiert werden [AMM97] und zum anderen können

Meta-Informationen automatisch generiert werden. Diese Meta-Informationen können zur deskriptiven Suche über Objekte benutzt werden.

Transformation und Restrukturierung: *Araneus* [AMM97] und *Strudel* [FFK+97] nutzen Wissen über die Struktur von HTML-Seiten für die Informationsgewinnung aus. Die gewonnenen Daten werden in einer relationalen Datenbank gespeichert. Diese Daten können manipuliert und umstrukturiert werden, um sie anschließend wieder als HTML-Seiten zu präsentieren. Obwohl ein im Vergleich zu Grammatiken flexiblerer Ansatz gewählt wird, können fehlerhafte HTML-Seiten auch zu fehlerhafter Informationsgewinnung führen. Strudel geht von einem Graphmodell aus, ohne die interne Struktur der Dokumente zu berücksichtigen. Bei beiden Ansätzen sind jedoch keine Mechanismen für unterschiedliche Granularitäten von Dokumenten vorgesehen.

Web-basierte Werkzeuge: Verschiedene Hersteller kommerzieller Datenbanksysteme bieten Tools für die Entwicklung von Web-Servern und -Anwendungen an [Wm, Cor97, Inf, O2W]. Diese Produkte bieten meist proprietäre Lösungen für die Verbindung der jeweiligen Datenbanksysteme mit dem WWW.

Unter Berücksichtigung der oben aufgeführten Gesichtspunkte werden in dieser Arbeit verschiedene Fragen untersucht und entsprechende Lösungsansätze formuliert:

- Die gängigsten Verfahren zur Beschreibung textueller und (semi-)strukturierter Informationen werden untersucht. Gemeinsamkeiten und Unterschiede werden herausgearbeitet, sowie Vor- und Nachteile für die Integration in den zu entwickelnden Ansatz erörtert.

- Die Markupsprache XML wird auf ihre Anwendbarkeit für die Zwecke des Medienarchivs hin untersucht. Ein formaler Rahmen wird gebildet, der eine tiefergehende Untersuchung erlaubt. Auf der Basis der erkannten Einschränkungen von XML wird eine Erweiterung entwickelt, für die ein geeigneter Parser implementiert wird.

- Ein Datenmodell basierend auf XML wird entwickelt, welches eine adäquate interne Repräsentation von Daten gewährleistet. Auf der Basis dieses Datenmodells wird eine geeignete Anfragesprache vorgestellt, die es erlaubt, sowohl über die Struktur als auch über den Inhalt von Dokumenten anzufragen.

- Für die Umsetzung der Überlegungen in einen lauffähigen Prototypen wurden unterschiedlichste, aktuelle Techniken auf ihre Anwendbarkeit untersucht. Die getroffene Auswahl wird begründet und eine Einführung in die Schlüsseltechnologien gegeben.

- Eine prototypische Implementierung des Medienarchivs MONTANA wird vorgestellt, welche die Ergebnisse verifiziert und darüber hinaus die Anwendbarkeit der entwickelten Verfahren unter Beweis stellt. Basierend auf diesem Prototypen werden in Eigenentwicklung entstandene Anwendungen und Systeme vorgestellt.

1.2 Organisation dieser Arbeit

Die vorliegende Arbeit gliedert sich wie folgt: Im zweiten Kapitel werden Datenmodelle und Dokumentbeschreibungssprachen beschrieben und näher untersucht, wobei intensiver auf XML eingegangen wird. Das dritte Kapitel analysiert XML ausgehend von einer formalen Beschreibung und entwickelt eine Erweiterung. Im vierten Abschnitt wird ein auf XML basierendes Datenmodell für die Beschreibung von Objekten und Hierarchien vorgestellt, welches die vorgeschlagene Erweiterung benutzt. Im fünften Kapitel wird das am Institut für Informationssysteme entwickelte Medienarchiv MONTANA vorgestellt. Abgeschlossen wird diese Arbeit durch eine Zusammenfassung und einen Ausblick über weiter zu verfolgende Ziele.

2

Modellierung textueller und (semi-) strukturierter Daten

„Jeder, der von der Quantentheorie nicht schockiert ist, hat sie nicht verstanden.“

NIELS BOHR

Die Bereiche Datenbanken und Dokumentverwaltung sind Forschungsdisziplinen, die lange Zeit unabhängig voneinander geführt wurden. Erst durch die Verbreitung des World Wide Web sind diese beiden Richtungen enger zusammengewachsen [ABS00]. Die Entwicklung des Medienarchivs MONTANA erfordert die eingehende Betrachtung von Verfahren sowohl zur Speicherung großer Datenmengen als auch zur Erstellung, Verwaltung und Manipulation von strukturierten textuellen Informationen. Daher sollen in diesem Kapitel zunächst die grundsätzlichen Eigenschaften, Parallelen und Nachteile der unterschiedlichen Ansätze herausgearbeitet werden.

2.1 Konventionelle und moderne Datenmodelle

Datenbanken dienen dem dauerhaften Speichern, Manipulieren, Organisieren und Anfragen großer Datenmengen. Ihr Ursprung liegt in der Notwendigkeit, Daten über die reale Welt, die von Anwendungsprogrammen verarbeitet werden, als von diesen Programmen unabhängige Information zu behandeln. Im Laufe der Zeit entstanden unterschiedliche Ansätze zur Modellierung von Daten, die sowohl in kommerziellen als auch in frei verfügbaren Datenbanksystemen mündeten. Die wichtigsten Paradigmen sollen hier kurz erläutert werden.

2.1.1 Das Entity-Relationship-Modell

Das *Entity-Relationship-Modell (ER-Modell)* wurde ursprünglich in [Che76] vorgeschlagen. Dieses Modell ist einfach genug, um von Nicht-Experten schnell verstanden zu werden und trotzdem so ausdrucksstark, um es in wissenschaftlichen und technischen Anwendungen einsetzen zu können.

Aus der Sicht des ER-Modells besteht die Welt aus Entitäten (*Entities*) und Beziehungen (*Relationships*) zwischen ihnen. Eine Entität ist ein „Seiendes“, welches eindeutig identifiziert werden kann, z. B. eine Person, ein Buch, ein Raum usw. Eine Entität wird ihrerseits durch Attribute näher charakterisiert (Name einer Person, ISBN-Nr. eines Buchs). Schließlich spiegelt eine Beziehung das Verhältnis zwischen Entitäten wider. Beispielsweise

- eine Person „liest“ ein Buch , oder
- eine Person „liest“ ein Buch „in“ einem Raum.

Zur Visualisierung werden einfache graphische Elemente eingesetzt, wie sie in Abbildung 2.1 zu sehen sind.

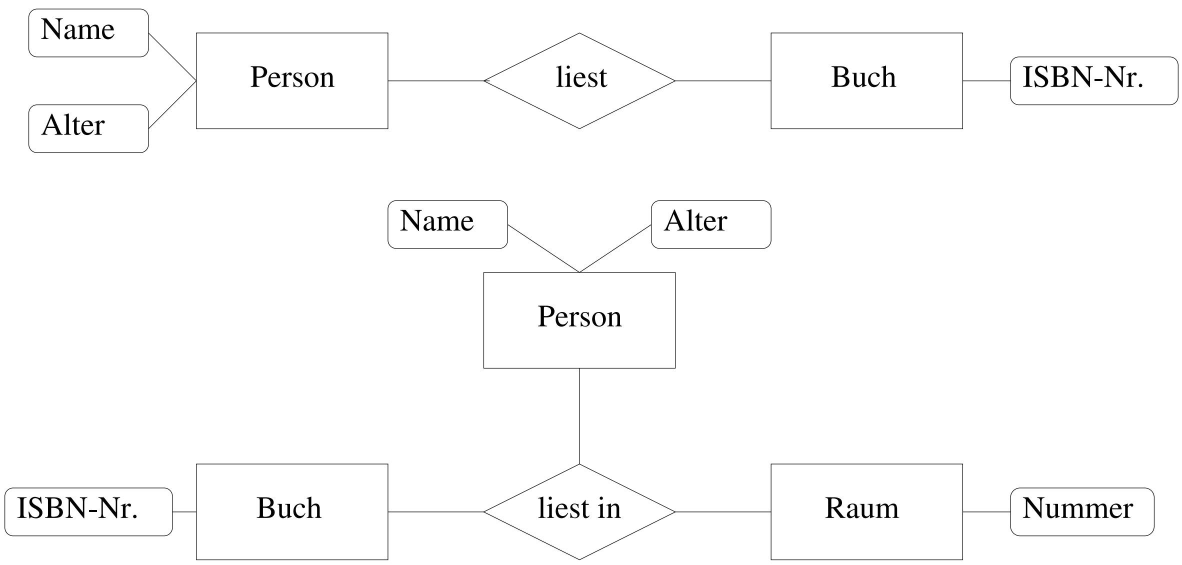


Abbildung 2.1: Graphische Darstellung zweier Szenarien im ER-Modell

Das ER-Modell hat als Hilfsmittel zum Datenbank-Schemaentwurf weite Verbreitung gefunden. Dagegen wird es als tatsächliches Datenmodell zur Speicherung von Informationen äußerst selten genutzt.

2.1.2 Das Relationale Datenmodell

Das Relationale Datenmodell basiert auf den Arbeiten von Codd [Cod70]. Im kommerziellen Bereich hat sich dieses Modell Mitte der 80'er Jahre zum Standard etabliert. Daten werden im Relationalen Modell innerhalb von Tabellen (sogenannten Relationen) abgelegt [Ull88]. Ein Eintrag in einer solchen Tabelle wird als Tupel bezeichnet. Die Spalten einer Tabelle sind mit Attributen bezeichnet, denen jeweils ein eigener Wertebereich zugeordnet ist. Diese Wertebereiche sind flach, was bedeutet, dass hier keine strukturierten Typen verwendet werden können. Abbildung 2.2 zeigt zwei Tabellen mit Personen und Büchern, die von Personen gelesen werden.

Um Anfragen auf Relationen zu formulieren, wurde die Relationenalgebra bzw. der gleichmächtige Relationenkalkül als theoretische Basis definiert. Als konkrete Benutzeranfragesprache hat sich die *Structured Query Language (SQL)*, welche vom *American National Standards Institute (ANSI)* genormt wurde, herausgebildet. Mittlerweile liegt sie in der dritten Version als SQL1999 vor. SQL ist eine Mischung von tupelorientiertem Relationenkalkül mit einigen algebraischen Bestandteilen, angereichert um arithmetische und textverarbeitende Elemente [Bis95].

PERSON	
Name	Alter
Hans Meier	46
Otto Schäfer	39
Karl Müller	53

LIESST_BUCH	
Name	ISBN-Nr.
Hans Meier	191-222-101
Hans Meier	081-15-191
Otto Schäfer	393-100-200

Abbildung 2.2: Relationale Darstellung mit zwei Tabellen

Die Verwaltung von strukturierten textuellen Informationen mittels relationaler Systeme gestaltet sich aufgrund der hierarchischen Strukturen schwierig, wie die Arbeit [Gün98] zeigt. Die Abbildung der Strukturen auf Tabellen wirkt künstlich und ist bei vielen Operationen umständlich zu handhaben. Eine Normalisierung von Relationen, wie sie sonst zur Optimierung verwendet wird, ist nicht mehr praktikabel.

2.1.3 Der ODMG-Standard

Der objektorientierte Ansatz sieht die Welt als eine Sammlung von Objekten. Diese Objekte sind gekapselt, was bedeutet, dass ein Zugriff auf sie nur über eine vordefinierte Schnittstelle möglich ist. Jedes Objekt hat eine Identität, die sich während der Lebenszeit nicht verändert. Objekte werden zu Klassen zusammengefasst, zwischen denen es eine Vererbungshierarchie gibt. Abbildung 2.3 zeigt die schematische Darstellung des vorangegangenen Beispiels als Klassen.

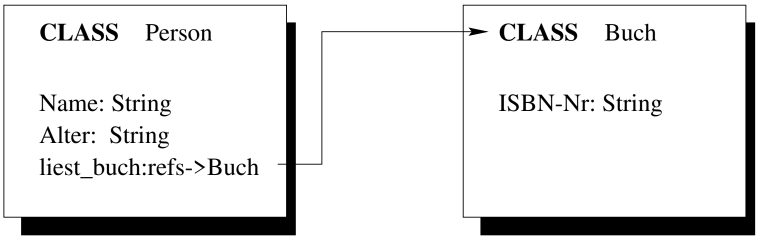


Abbildung 2.3: Graphische Darstellung zweier Klassen

Breite Verwendung fand der objektorientierte Ansatz durch die Einführung der Programmiersprache C++ [Str86]. Anfang der 90'er Jahre kamen dann die ersten objektorientierten Datenbanken. Sie sollten den sogenannten *impedance mismatch*, also die Kluft zwischen Datenmodell und Programmiersprache, die besonders beim Relationalen Modell auftritt, überwinden [Uml95]. Die unterschiedlichen Ausführungen des objektorientierten Ansatzes wurden von der *Object Database Management Group (ODMG)* zu einem quasi-Standard vereint [Cat93], der im Laufe der Zeit erweitert wurde [Cat97]. Die bekanntesten objektorientierten Datenbanken sind heutzutage ObjectStore von eXcelon Corp. [eC], die Object Server Suite von Poet [Sof] und Objectivity/DB von Objectivity Inc. [Obj].

Trotz des anfänglichen Enthusiasmus für das objektorientierte Konzept spielen objektorientierte Datenbanksysteme (OODBMS) heute nur noch eine Nebenrolle. Als Gründe dafür werden oft die mangelnde Stabilität, geringe Performance und das Fehlen einer allgemeinen Anfragesprache (ähnlich SQL) genannt.

2.1.4 Erweitert-relationale und objektrelationale Systeme

Erweitert-relationale und objektrelationale Datenbanksysteme sind solche, die sowohl Aspekte des relationalen als auch der objektorientierten Modelle benutzen. Es gibt keine exakte Definition, welche Eigenschaften ein System erfüllen muss, so dass es als objektrelational zu bezeichnen ist.

Den objektrelationalen Systemen ist gemein, dass sie die Forderung des Relationalen Modells nach flachen Tabellen aufheben. So können Wertebereiche einzelner Attribute wiederum strukturiert sein. Darüber hinaus wurden verschiedene objektorientierte Paradigmen übernommen (z. B. Methoden, Objektidentitäten, Referenzen usw.). Abbildung 2.4 zeigt eine verschachtelte Tabelle, die das relationale Beispiel aus Abbildung 2.2 umsetzt.

Erste Projekte, die in die Richtung objektrelationaler Systeme gingen, waren das AIM-P [LKD+88], welches eine erweiterte Form des NF^2-Modells [SS86] umsetzte. Inzwischen haben nahezu alle Anbieter kommerzieller relationaler Datenbanksysteme ihre Produkte mit objektrelationalen Bestandteilen erweitert. Der SQL1999-Standard besitzt bereits objektrelationale Sprachelemente.

PERSON_LIESST_BUCH		
Name	Alter	ISBN-Nr
Hans Meier	46	191-222-101 081-15-191
Otto Schäfer	39	393-100-200
Karl Müller	53	

Abbildung 2.4: Objektrelationale Darstellung mit einer verschachtelten Tabelle

2.2 Dokumentbeschreibungssprachen

Die Verarbeitung elektronischer Dokumente umfasst das Erstellen, Verändern, Formatieren, Speichern, Anfragen und Verteilen von textuellen Informationen. Das Konzept des strukturierten Dokuments hängt eng mit der Entwicklung der computergestützten Dokumentverarbeitung zusammen. Die Überlegungen in diesem Bereich beruhten auf der Erkenntnis, dass die Modellierung von Dokumenten einer Abstraktion bedarf. Die Modellierung soll dabei sowohl die typographischen Eigenschaften bewahren als auch die Austauschbarkeit textueller Daten ermöglichen. Um dieses zu gewährleisten, ist die Trennung eines Dokuments in den physischen, syntaktischen und semantischen Teil erforderlich.

Im physischen Teil wird die Formatierung eines Dokuments beschrieben. Hier wird genau festgelegt, wie Absätze angeordnet werden, wie einzelne Wörter auszusehen haben (*italic, bold*) und wie Seiten aufgeteilt werden. Im syntaktischen Teil folgt dann die logische Gliederung, die die Einteilung z. B. in Kapitel, Paragraphen und Absätze definiert. Der semantische Teil ordnet die einzelnen Passagen des Dokuments der logischen Gliederung zu.

Die ersten Systeme zur Erstellung von Dokumenten waren sogenannte *Textsatzprogramme* (TEX,*troff*). Diese meist ASCII-orientierten Anwendungen reichern Dokumente um sichtbare Formatierungsbefehle an. Ein solches Vorgehen wird auch als *prozedurales Markup* bezeichnet. Der Vorteil dieser Systeme liegt vor allen Dingen darin, dass keine eigenen Datenformate benötigt werden. Änderungen sind jederzeit mit einfachen Editoren möglich. Als nachteilig erweist sich aber, dass die logischen Bestandteile eines Dokuments nicht kenntlich gemacht sind. Während der Benutzer in der Lage ist, die verschiedenen Teile zu erkennen, ist dieses für ein elektronisches System in der Regel unmöglich.

Textverarbeitungssysteme (z. B.*Word* [Cor]) erstellen, durch den Benutzer gesteuert, formatierte Dokumente. Diese Dokumente werden in proprietären Dateiformaten innerhalb von Filesystemen abgelegt. Textverarbeitungssysteme bieten keine Möglichkeit, Dokumente mit bestimmten Inhalten deskriptiv zu suchen. Auch erlauben sie nur die Transformation ihrer Dokumente in vordefinierte Formate. Für andere Transformationen sind spezielle Implementierungen notwendig. Aus diesem Grund können die von Textverarbeitungssystemen erstellten Dokumente nicht von zeichenorientierten Retrieval-Systemen verarbeitet werden. Selbst ein Update dieser Dokumente mit eigenen Werkzeugen ist nur schwer zu realisieren. Durch die unterschiedlichen Zielsetzungen der einzelnen Systeme ist der Benutzer nicht nur dazu gezwungen verschiedene Darstellungen zu benutzen, sondern auch, sich mit den zugrunde liegenden Philosophien der Systeme auseinanderzusetzen. Dieses stellt gerade dann ein Problem dar, wenn es nötig wird, Dokumente auf mehreren Programmsystemen oder gar Plattformen zu verarbeiten und zu manipulieren.

Dieser Mißstand führte zur Entwicklung von Systemen zur plattformunabhängigen Verarbeitung strukturierter Informationen (*SGML* [ISO86],*ODA* [ISO93]). In diesen Systemen besteht eine strikte Trennung zwischen der logischen Gliederung und präsentationsspezifischen Details. Durch ein sogenanntes *deskriptives Markup* wird es möglich, für den gleichen Inhalt unterschiedliche Layouts zu definieren. Diese Klasse von Systemen erfüllt die dreistufige Unterteilung des Dokumententwurfs, wie sie oben beschrieben wurde.

2.2.1 Standard Generalized Markup Language (SGML)

Die *Standard Generalized Markup Language* (SGML) [ISO86] ist ein im Jahr 1986 entwickelter quasi-Standard für die Erstellung und den Austausch strukturierter Dokumente. Die Kennzeichnung der logischen Struktur geschieht durch sogenannte *Tags*. Diese werden in SGML definiert und in einem Dokument benutzt. SGML besitzt einige nachteilige Eigenschaften, die neben dem großen Definitionsumfang der Sprache Dinge, wie formalsprachliche Unentscheidbarkeiten, umfassen.

Da SGML eine Obermenge von XML ist, welches wir später noch ausführlicher beschreiben werden, soll hier nur auf die wichtigsten Eigenheiten von SGML eingegangen werden.

Für die Definition der Struktur einer Dokumentklasse werden *Document Type Definitions (DTDs)* verwendet. Eine *DTD* beschreibt, ähnlich einer kontextfreien Grammatik, die mögliche Schachtelung von Elementteilen. Ein SGML-Dokument wird nach Regeln, die von der DTD vorgegeben sind, aufgebaut. Wichtig ist, dass in diesem Teil der Dokumenterstellung nur inhaltliche und logische Teile des Dokuments verfasst werden.

Zur Beschreibung des Layouts von Elementen wurde die *Document Style Semantics and Specification Language (DSSSL)*) entwickelt, die zwar sehr mächtig ist, aber für den Benutzer i. d. R. schwer zu erlernen und anzuwenden ist. DSSSL ist sehr flexibel; so ist die Ausgabe grundsätzlich in beliebige Formate (z. B. PDF, Postscript [Ado] usw.) möglich.

DSSSL besteht im Wesentlichen aus zwei Komponenten; einer Transformationssprache (*transformation language*) und einer Stilsprache (*style language*). Erstere soll (SGML-)Dokumente, die nach einer bestimmten DTD kodiert sind, in ein Format wandeln, in dem sie anderen DTDs entsprechen. Die zweite Sprache legt für logische Teile eines Dokuments deren Aussehen fest.

Wichtigste Bestandteile von DSSSL sind die *Flow Object Classes*, die *Standard Document Query Language (SDQL)* und die *Expression Language*, wobei sich diese Sprachen teilweise überlappen.

Der Verarbeitungsprozeß wird in Abbildung 2.5 skizziert. Graue Teile der Skizze werden vom DSSSL-Standard definiert.

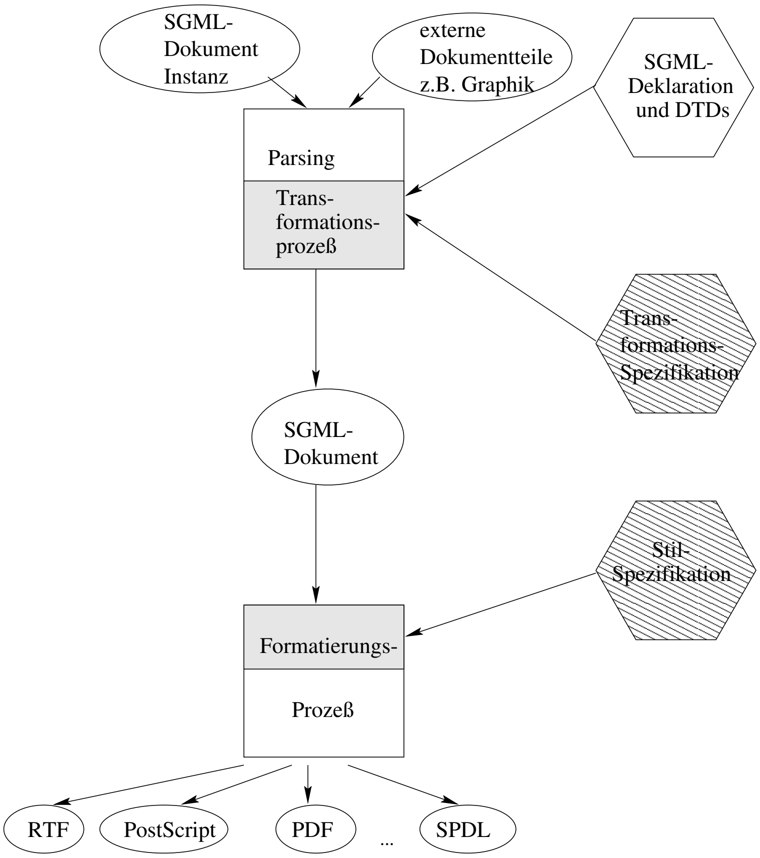


Abbildung 2.5: Der Verarbeitungsprozeß von DSSSL

2.2.2 HTML und LaTeX

Die Sprachen HTML [BLC] und LaTeX[Lam86] fallen in gewisser Weise aus dem anfangs dieses Abschnitts beschriebenen Raster heraus. Obwohl sie die Gliederung eines Dokuments in logische Teile unterstützen, bieten sie darüber hinaus Formatierungsanweisungen, die Teile eines Dokuments in ein benutzergesteuertes Layout wandeln. Die beiden Ansätze stellen somit eine Mischform des deskriptiven und prozeduralen Markups dar.

HTML ist eine Auszeichnungssprache, die selbst in SGML definiert wurde. Sie dient als Dokumentbeschreibungssprache für das World Wide Web. Das Abweichen von der ursprünglichen Idee des deskriptiven Markups liegt bei den Browserherstellern (*Microsoft*, *Netscape*) begründet, die den Wünschen der Benutzer nach ständig neuen Formatierungsmöglichkeiten Rechnung trugen.

Bei LaTeX handelt es sich um einen TeX-Aufsatz, der neben neuen logischen Elementen auch weiterhin die Formatierungsbefehle der TeX-Schicht zur Verfügung stellt.

2.3 Semistrukturierte Daten

Für die Integration und Verwaltung heterogener Datenquellen, wie sie auch im WWW vorhanden sind, tritt derzeit ein Ansatz zunehmend in den Vordergrund, welcher eine angemessene Behandlung der uneinheitlichen Struktur der betreffenden Informationen und Informationsquellen erlaubt. Da man es im Kontext heterogener Quellen mit hochstrukturierten Informationen, aber auch mit unstrukturierten sowie diversen Zwischenformen zu tun haben kann, hat sich hier die Bezeichnung *semistrukturierte Daten* durchgesetzt. Semistrukturierte Daten sind weder so stark strukturiert wie z. B. relationale Daten, noch sind sie strukturlos. Es existiert keine genaue Definition, wann Informationen als semistrukturiert anzusehen sind. [ABS00, Abi97, Clu97] geben einen Überblick über diesen Bereich. Klassische Beispiele für semistrukturierte Daten sind BiBTEX-Dokumente, strukturierte Texte und insbesondere WWW-Dokumente (HTML, XML).

Den meisten Ansätzen zur Beschreibung semistrukturierter Daten liegt ein Baum- oder Graphenmodell zugrunde. Informationen werden mit diesen Hilfsmitteln selbstbeschreibend dargestellt. Selbstbeschreibend bedeutet dabei, dass neben den reinen Informationen auch Schemadaten auf gleicher Ebene modelliert werden.

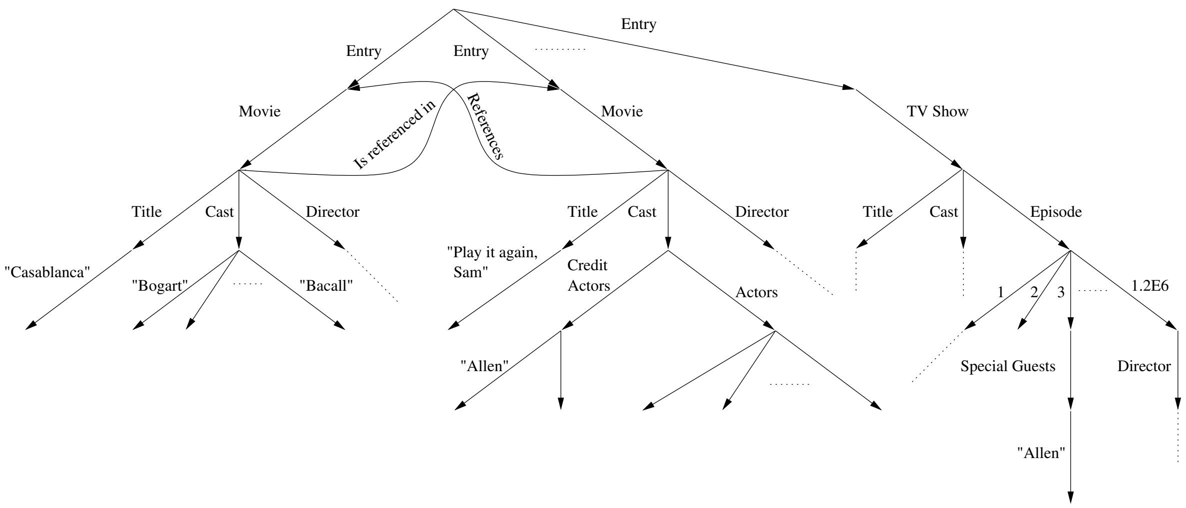


Abbildung 2.6: „Movie-Actor-Database“ (aus [Bun97])

Im Wesentlichen sind drei Motivationen hinter der Idee der semistrukturierten Daten zu finden [Bun97]:

- Der Hauptgrund für das Konzept der semistrukturierten Daten liegt in dem Bedürfnis, neue Formen von Daten mit herkömmlichen Datenbanktechnologien zu verwalten. Während sich einige dieser Formen (z. B. strukturierte Texte) in der Regel noch mit objektorientierten ([Cat93, Cat97]) oder objektrelationalen [CMN97, Sto96] Systemen verwalten lassen, sind beispielsweise WWW-Dokumente so unstrukturiert, dass dieses Vorgehen zu sehr umständlichen Lösungen führen würde.

- Die zweite Motivation liegt in der Notwendigkeit für den Austausch und die Transformation von Daten, denen unterschiedliche Paradigmen zugrunde liegen. So verfolgt das *Object Exchange Model* (OEM) [PGMJ95] des *TSIMMIS*-Projekts [GMQR+97] das Ziel, Informationen und Objekte unterschiedlicher Modelle mit einheitlichen Konstrukten zu beschreiben. Die zugehörige Anfragesprache lautet *LOREL* [AQM+97, GMW99]. Mit ihr können Anfragen an Quellen gestellt werden, die über das OEM spezifiziert wurden. Die Anfragesprache ist in der Lage mit unterschiedlichen, den Quellen zugrunde liegenden Typen umzugehen.

- Schließlich verallgemeinert das semistrukturierte Paradigma das Navigieren in Datenbeständen. Hiermit wird es möglich, neben Anfragen auf Daten, auch Anfragen über dem Schema zu formulieren. [BDHS96] stellen die Sprache *UnQL* vor, welche basierend auf einem Graphenmodell Anfragen sowohl über Schemainformationen als auch den eigentlichen Daten zulässt.

Im Umfeld semistrukturierter Daten finden sich verschiedene Fragestellungen, u. a. Gebiete wie Anfragesprachen [AQM+97, MS99, CCS00], Optimierung von Anfragen [GW97], Gewinnung von Strukturinformation [NAM97] und Sichtenverwaltung [Abi99], welches aktuelle Forschungsthemen sind.

2.4 Extensible Markup Language (XML)

Die *Extensible Markup Language* (XML) [W3C98a] ist ein im Jahr 1998 vom W3C [W3C] vorgeschlagener Standard zur Strukturierung und Beschreibung von Dokumenten im Web. Dieser Standard hat seitdem in rasanter Weise in die unterschiedlichsten Bereiche Einzug gefunden.

XML basiert auf der Idee des strukturierten Auszeichnens von Daten. XML liegt als Konzept die anfangs angesprochene Trennung von Inhalt, Struktur und Präsentation (Abbildung 2.7) zugrunde.

Bezüglich SGML kann man XML als eine vereinfachte Teilmenge sehen, wobei die Grundkonzepte nahezu unverändert sind. In Bezug auf HTML ist XML dagegen in dreierlei Hinsicht allgemeiner [Bos97]:

- Benutzer können eigene Tags im Hinblick auf ihre Anwendung definieren.

- Dokumentstrukturen können beliebig tief geschachtelt werden.

- Jedes XML-Dokument besitzt eine optionale Grammatikbeschreibung. Eine solche Grammatik kann von einer Anwendung zur strukturellen Verifikation benutzt werden.

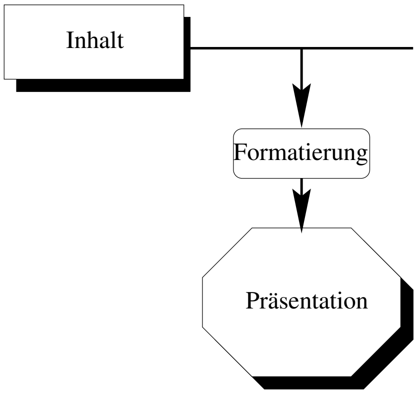


Abbildung 2.7: Sprachkonzept von XML

Daten werden in XML in Elemente eingebettet, welche durch Tags gekennzeichnet sind. Abbildung 2.8 zeigt ein Beispiel für ein XML-Dokument, welches als Grundlage für weitere Betrachtungen in dieser Arbeit dienen wird.

```
<bookstore>
  <novel id="a1" publisher="p2">
   <exemplar>
    <title> Ordinary Worlds </title>
    <author> Can Miller </author>
   </exemplar>
  </novel>
  <journal id="a2" publisher="p1"
     status="published">
    <exemplar>
    <title> Lecture Notes, Vol. 23 </title>
    </exemplar>
  </journal>
 <publisher id="p1">
    <name> Springer </name>
 </publisher>
 <publisher id="p2">
    <name> Warner </name>
 </publisher>
</bookstore>
```

Abbildung 2.8: Beispiel für ein XML-Dokument

XML-Daten sind selbstbeschreibend und besitzen offensichtliche Ähnlichkeit zu semistrukturierten Daten. Trotzdem bleibt zu beachten, dass XML ursprünglich lediglich als Markupsprache konzipiert wurde. Daher besitzt es kein zugehöriges Datenmodell. Auf den ersten Blick fallen dennoch Ähnlichkeiten zu konventionellen Datenmodellen auf:

Attribute: XML-Elemente können Attribute besitzen. Diese Attribute charakterisieren Elemente. In der Abbildung 2.8 gibt das Attribut „status" eines „journal" Auskunft darüber, ob es „published" oder „unpublished" ist.

Referenzen: Elemente können explizite Objekt-Identitäten erhalten (*oids*) und können auf *oids* anderer Elemente mithilfe reservierter Attribute verweisen. Betrachten wir Abbildung 2.8, so besitzt beispielsweise das Element „publisher" mit dem Namen „Springer" einen Identifikator welcher von einem anderen „journal" mit dem Namen „Lecture Notes" referenziert wird. Die Definition von XML stellt es der jeweiligen Anwendung frei, wie es Referenzen interpretiert. Sie fordert nur, dass alle referenzierten *oid*s auch definiert sein müssen.

Links: XML-Dokumente können *Links* auf Elemente im selben oder auch in anderen Dokumenten besitzen. *Links* können ähnlich wie *HTML anchors* über Servergrenzen hinaus verweisen.

Ordnung: Laut Definition ist ein XML-Dokument mit dessen Elementen geordnet. Dieses ist eine wichtige Eigenschaft, da herkömmliche Anfragesprachen i. d. R. auf Mengenbasis und damit ohne Berücksichtigung der Ordnung arbeiten [DFF+98].

Der Grammatikteil eines XML-Dokuments wird *Document Type Definition* (DTD) genannt. XML versetzt Autoren in die Lage, ihre eigenen Dokumenttypen zu definieren.

```
<!ELEMENT bookstore    (novel|journal|
                       publisher)*>
<!ELEMENT novel        exemplar>
<!ELEMENT journal      exemplar>
<!ELEMENT exemplar     (title, author?)>
<!ATTLIST novel        id ID
                       publisher IDREF>
<!ATTLIST journal      id ID
                       publisher IDREF
                       status="published|
                       unpublished">
<!ELEMENT publisher    (name)>
<!ATTLIST publisher    id ID>
<!ELEMENT title        (#PCDATA)>
<!ELEMENT author       (#PCDATA)>
<!ELEMENT name         (#PCDATA)>
```

Abbildung 2.9: Beispiel für eine XML-DTD

Eine XML-DTD spezifiziert mittels regulärer Ausdrücke die Art und Weise wie Elemente geschachtelt werden dürfen. Abbildung 2.9 zeigt eine DTD, die eine Struktur für das obige XML-Dokument festlegt.

Wie wir bereits in den vorangegangenen Beispielen gesehen haben, ist die Wahl der Modellierungsart eines realen Objekts, entweder als Element oder als Attribut, oftmals willkürlich. In vielen Fällen verwischt die Bedeutung des Elementinhalts und der Attributierung. Daher werden wir in der XML-Abstraktion im nächsten Kapitel auf eine Unterscheidung von Elementen und Attributen verzichten. Dennoch bleibt anzumerken, dass es wichtige Unterschiede gibt. So sind z. B. Elemente geordnet, während es Attribute nicht sind. Das zu entwickelnde Modell ließe sich aber in einfacher Weise um Attributierungen erweitern.

Namensräume

Namensräume (*Namespaces* [W3C99g]) dienen in XML dazu, Elementbezeichner bei deren Verwendung zu „qualifizieren". Konkret erlaubt das Konzept der Namensräume die Nutzung von gleichnamigen Elementen unterschiedlicher Herkunft, indem die jeweiligen Namensräume den verwendeten Elementen vorangestellt werden.

Ein Namensraum verbindet ein Element mit einem *Uniform Resource Identifier (URI)*. Dieser URI gibt nun Auskunft über die Bedeutung und die Benutzung des Elements. Das Element wird fortan als Präfix für andere Elemente verwendet. Wenn es in dieser Form angewendet wird, stammt das dem Präfix nachstehende Element aus dem zugewiesenen Namensraum.

Beim praktischen Einsatz werden Namensräume über ein Attribut `xmlns` definiert. Sie gelten jeweils lokal innerhalb des zugehörigen Elements. Im Folgenden Beispiel wird ein Namensraum `xlink` definiert. Dieser Namensraum wird über den URI `http://www.w3.org/1999/xlink/namespace` identifiziert. Das Element `simple` ist ein, aus diesem Namensraum stammendes Element. Die Definition des Namensraums innerhalb des Elements `simple` stellt dabei nur eine verkürzte Schreibweise dar.

```
<xlink:simple xmlns:xlink="http://www.w3.org/1999/xlink/namespace">
...
</xlink:simple>
```

Namensräume kommen beispielsweise bei allgemeinen XML-Standards wie XSL, XPointer oder XML Schema zum Einsatz. Durch die Verwendung von Namensräumen werden insbesondere mißverständliche Deutungen durch Anwender oder Applikation verhindert.

2.4.1 Allgemeine Modellierung von XML-Daten

Im Kontext von XML gibt es eine Reihe von Modellen, Sprachen und Konzepten, von denen wir die wichtigsten im Folgenden näher beschreiben werden.

DOM

XML-Daten sind in ihrer einfachsten Form reine Textdateien. Für die Verarbeitung des Inhalts dieser XML-Dateien ist jedoch ein universeller Zugriff wünschenswert. Da XML-Dokumente durch das Tagging eine implizite Baumstruktur besitzen, liegt es nahe, diese Struktur als logische Sicht zu verwenden.

Das *Document Object Model (DOM)* [W3C98c] ist eine plattform- und sprachunabhängige Schnittstelle, die es Programmen und Skriptsprachen erlaubt, dynamisch auf den Inhalt, die Struktur und das Layout eines XML-Dokuments zuzugreifen. Ein Dokument lässt sich auf der Basis der *DOM*-Sicht verarbeiten und manipulieren, wobei Verarbeitungsergebnisse wieder in das Dokument eingefügt werden können.

Zur Überführung eines XML-Dokuments in eine Baumstruktur dienen XML-Parser [Javc]. Die Informationen stehen so einer maschinellen Weiterverarbeitung zur Verfügung. Dabei werden alle Tags, deren Inhalte und die zwischen diesen herrschenden Beziehungen als Objekte definiert. Diese Objekte werden in eine Objekthierarchie abgebildet, auf die dann maschinell oder vom Benutzer zugegriffen werden kann (s. Abbildung 2.10).

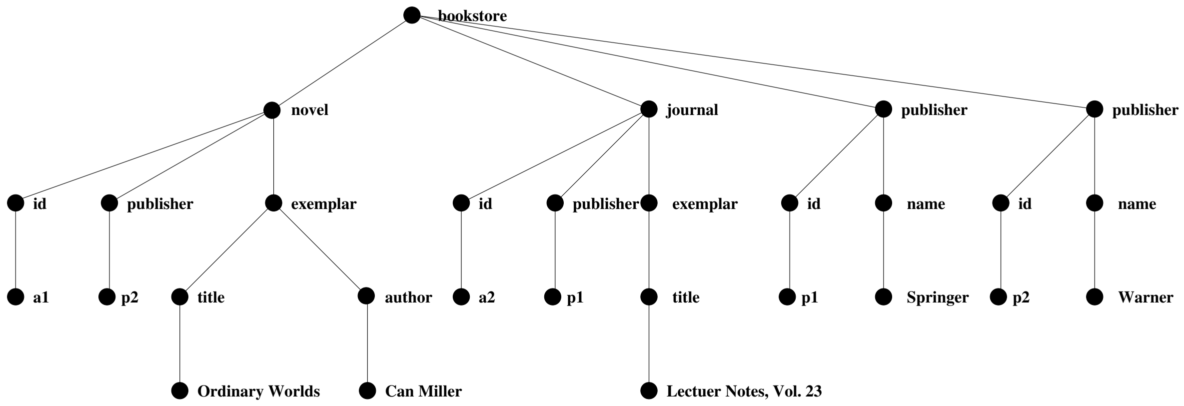


Abbildung 2.10: Baumdarstellung eines kompletten XML-Dokuments

Parser werden, je nachdem ob sie zur Analyse eines Dokuments eine DTD benutzen, in validierend und nicht-validierend unterschieden. Auch wird bei Parsern nach der Vorgehensweise differenziert. Während *document driven parser* ein Dokument vollständig durchlaufen und dabei die Struktur komplett (und speicheraufwendig) abbilden (in Form des *DOM*), schränken *event driven parser* ihre Aktivitäten unter Benutzung der *Simple API for XML (SAX)* nur auf jeweils benötigte Regionen ein. Hierbei werden externe Applikationen über das Auffinden vorgegebener Elemente informiert. Durch dieses Vorgehen ist es insbesondere nicht mehr notwendig, Dokumente komplett im Speicher zu halten.

Für den Zugriff auf das DOM gibt es Schnittstellendefinitionen für unterschiedliche Programmiersprachen, u. a. Java, C++ und auch in der Schnittstellenbeschreibungssprache *IDL*.

Auf der Basis dieser Schnittstellendefinitionen gibt es zahlreiche kommerzielle und frei verfügbare Implementierungen des DOM.

Das DOM ist eine sehr allgemeine Schnittstelle, wie Abbildung 2.11 zeigt. Bei der Veränderung der Baumhierarchie gibt es keinerlei schemaorientierte Verifikationen. So sind beispielsweise Einfügeoperationen uneingeschränkt möglich, solange sie die Baumstruktur nicht verletzen. DTD-orientierte Operationen, d. h. Manipulationen unter Berücksichtigung der grammatikalischen Struktur, wie sie in [Lin79] untersucht wurden, sind im DOM nicht möglich [Kem00].

```
public interface Node {
    String getNodeName ();
    void setNodeValue (String nodeValue) throws DOMException;
    String getNodeValue () throws DOMException;
    short getNodeType ();
    Node getParentNode ();
    NodeList getChildNodes ();
    Node getFirstChild ();
    Node getLastChild ();
    Node getPreviousSibling ();
    Node getNextSibling ();
    NamedNodeMap getAttributes ();
    Document getOwnerDocument ();
    Node insertBefore (Node newChild, Node refChild) throws DOMException;
    Node replaceChild (Node newChild, Node oldChild) throws DOMException;
    Node removeChild (Node oldChild) throws DOMException;
    boolean hasChildNodes ();
    Node cloneNode (boolean deep);
}
```

Abbildung 2.11: DOM-Schnittstellendefinition der Java-Klasse `Node`

Die Definition des DOM liegt mittlerweile in der zweiten Version vor (DOM Level 2). Gegenüber der ersten Version weist diese zusätzliche Funktionalitäten auf, die dem Anwendungsentwickler die Arbeit mit dieser Schnittstelle erleichtern helfen. So erlaubt DOM Level 2 Style Sheets und die Manipulation der Stilinformationen eines Objekts, definiert ein Event-Modell und unterstützt Namespaces. Die geplante dritte Version soll darüber hinaus die oben angesprochenen schemaorientierten Operationen unterstützen.

XPath

Der *XPath*-Standard [W3C99c] liefert die konzeptuelle Grundlage für die Traversierung von XML-Dokumenten. XPath entstand als Nebenprodukt bei der Entwicklung von XSL und XPointer, da diese Sprachen ähnliche Konstrukte zur Navigation in Dokumenten und zum Ansprechen einzelner Dokumentfragmente erforderten.

XPath-Ausdrücke beschreiben auf der Basis der oben angesprochenen Dokumentstruktur, wie einzelne Objekte eines Dokuments (Elemente, Attribute, Processing Instructions etc.) erreicht werden können. Die XPath-Syntax ist nicht strukturell XML-Element-basiert (wie z.B. im DOM), sondern kompakt und String-basiert. Dadurch wird eine Benutzung von XPath-Ausdrücken sowohl in XML-Attributen als auch in URIs [BLFM] möglich.

Ein XPath-Ausdruck ist vereinfacht gesagt ein Lokationspfad. Ein Lokationspfad spezifiziert in navigierender Form die Position eines Objekts in einer Struktur. Ein XML-Dokument wird dabei, ähnlich wie beim DOM, als ein logischer Baum angesehen, unabhängig von dessen physikalischer Repräsentation.

Zunächst soll das XPath-Datenmodell kurz erläutert werden. Betrachten wir das XML-Dokument in Abbildung 2.8. Im zugehörigen Baum eines XML-Dokuments gibt es für jedes Konstrukt einen Knoten. Als Beispiel soll wiederum die Baumdarstellung in Abbildung 2.10 dienen.

So liefert z. B. der Ausdruck

```
bookstore/novel[@position=1]/exemplar[@position=1]/author
```

den `author`, des ersten `exemplar` des zweiten `novel` unterhalb eines `bookstore`.

Bei der Betrachtung der Baumstruktur ist die Tatsache wichtig, dass die Knoten geordnet sind. Dadurch wird es möglich, einzelne Knoten direkt über deren ordonale Position anzusprechen.

Linking in XML

XML unterstützt ein wesentlich weiter gehendes Linking, als es von HTML bekannt ist. Das *World Wide Web Consortium (W3C)* [W3C] hat ein Linking-Modell für XML vorgeschlagen, das aus XML Linking Language (*XLink* [W3C00b]) und XML Pointer Language (*XPointer* [W3C00c]) besteht.

Während XLinks der Definition von Hyperlinks dient, wie sie auch in vereinfachter Form von HTML bekannt sind, zielt XPointer auf die Referenzierung von Dokumentteilen ab. XLink und XPointer sind daher nicht als alternative sondern als sich ergänzende Vorschläge anzusehen.

Das *XLink*-Modell unterstützt neben unidirektionalen Links (wie sie heute in HTML verwendet werden) bidirektionale Links, die in beide Richtungen referenzieren und erweiterte (*extended*) Links. Erweiterte Links besitzen die Fähigkeit, gleichzeitig auf mehrere Dokumente oder Dateien zu verweisen. Hinzu kommt noch die Möglichkeit, Links außerhalb des Dokuments, zu dem sie gehören, verwalten zu können. Die unterschiedlichen Linkarten sind in Abbildung 2.12 visualisiert.

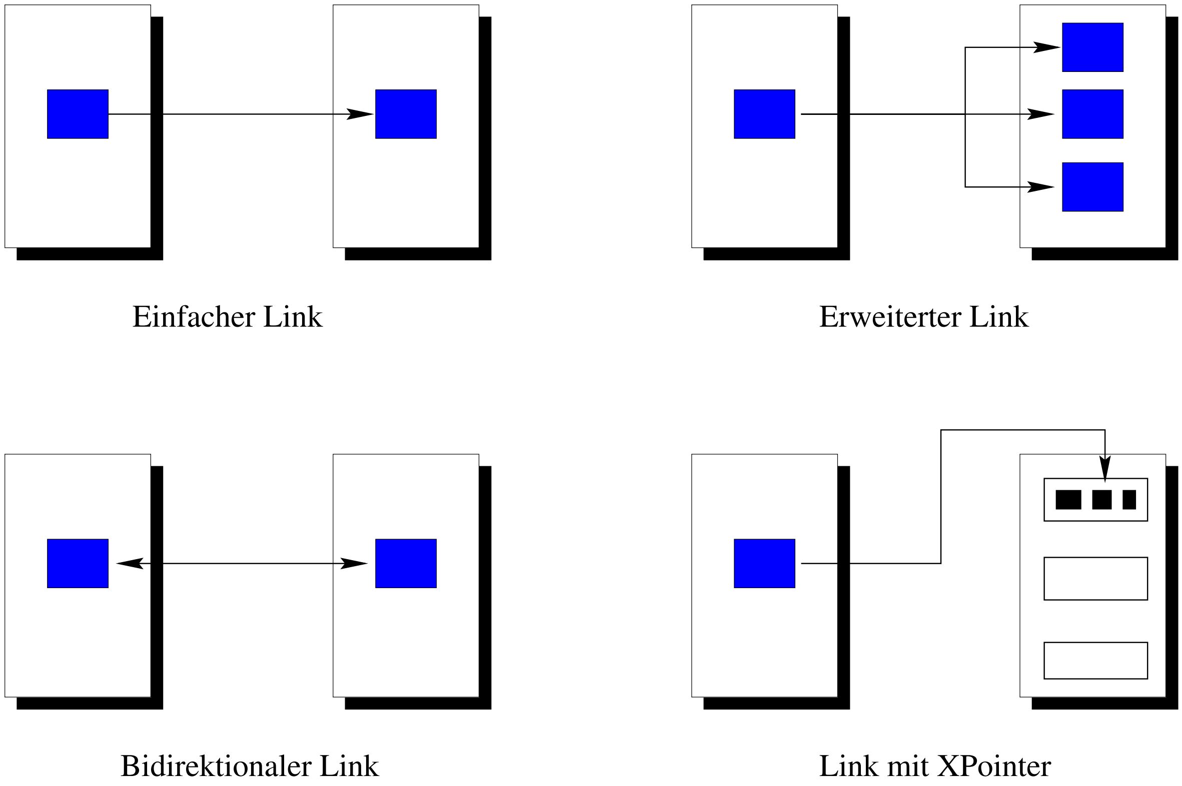


Abbildung 2.12: Veranschaulichung der unterschiedlichen Linkarten

Einfache Links (s. Abbildung 2.12) sind bereits von HTML her bekannt und werden über einen Anchor (Element `a`) definiert. So zeigt z. B.

```
<a href="~/doc.html#part1">link</a>
```

auf den mit `part1` bezeichneten Teil des Dokuments `doc.html`. Ein solcher Link wird in der Regel durch Anklicken innerhalb eines Browsers aktiviert.

Ein Beispiel für einen erweiterten XML-Link (s. Abbildung 2.12) kann wie folgt aussehen:

```
<xlink:extended  xmlns:xlink="http://www.w3.org/1999/xlink/namespace"
                 role="Liste" title="bookstore">
  <xlink:locator href="/path/doc.txt"
   role="Text-Dokument"
   title="Bookstore als Textdatei" />

  <xlink:locator href="/path/doc.xml"
```

```
   role="XML-Dokument"
   title="Bookstore als XML-Datei" />

  <xlink:locator from="/path/source.xml"
   to="/path/target.xml"
   title="Bookstore als XML-Datei" />

</xlink:extended>
```

In diesem konkreten Fall wird eine Liste von Links zu einem erweiterten Link zusammengefasst. Dieser erweiterte Link präsentiert unterschiedliche Formate eines Objekts.

Um die Semantik festzulegen, muss zunächst der Präfix `xlink` mit dem entsprechenden Namespace in Verbindung gebracht werden (Attribut `xmlns`). Über den Elementnamen (Element `extended`) wird festgelegt, dass es sich um einen erweiterten Link handelt.

Der erweitere Link besteht seinerseits aus mehreren Verweisen, die jeweils durch das Element `locator` gekennzeichnet sind. Während die ersten beiden Verweise ein einfaches Ziel besitzen (`href`), stellt letzterer einen sogenannten „Bogen“ dar, der über die Attribute `to` und `from` eine Verbindung zwischen Webressourcen herstellt. D. h., es wird eine Verknüpfung zwischen zwei Objekten definiert, die vorher nicht notwendigerweise in Beziehung standen. Hierbei kennzeichnet das Attribut `from` das Quellobjekt und das Attribut `to` das Zielobjekt.

Bei den Verweisen beschreibt das Attribut `role` die Rolle, also im gewissen Sinn die Semantik eines Links. Durch den `title` wird ein Name definiert, der beispielsweise für den Benutzer als Information dienen kann.

XPointer stellen ein weiteres Modell für das Linking in XML-Dokumenten dar. Mit XPointern ist es möglich, auf bestimmte Elemente einer XML-Instanz zu verweisen und zuzugreifen. Es kann direkt auf der Baumstruktur eines XML-Dokuments navigiert werden. Ein XML-Dokument kann unabhängig von seiner logischen Struktur aus beliebig vielen physischen Einheiten bestehen. Dadurch können (im Zusammenspiel von XLink und XPointer) in einem Netzwerk verteilte Daten zu einem einzigen XML-Dokument zusammengefügt werden. Als Sprachkonzept für die Definition von Pfaden in Dokumenten werden die oben beschriebenen XPaths benutzt.

Im folgenden Beispiel sehen wir einen XLink, der XPointer zur Lokalisierung von Dokumentteilen nutzt (vgl. Abbildung 2.12):

```
<xlink:simple xmlns:xlink="http://www.w3.org/1999/xlink/namespace"
              show="embeded"
              actuate="onload"
              href="/path/doc.xml#xpointer(id('novel'/exemplar[1]))"/>
```

Zunächst wird wiederum der Namespace zugewiesen (Attribut `xmlns`). Die Attribute `show` und `actuate` dienen als Information für die Client-Anwendung für die Darstellung des Links. Die Referenz (`href`) enthält einen XPointer-Anteil, der ausgehend vom Element `novel` das erste `exemplar`-Element anspricht. XPointer können neben einzelnen Dokumentteilen auch beliebige, nicht notwendigerweise zusammenhängende Fragmente ansprechen.

Die Lokalisierung von Dokumentteilen mittels XPointer kann sowohl durch absolute als auch durch relative Pfade geschehen. Während sich absolute Pfade auf konkrete Elemente und Attribute beziehen (wie im obigen Beispiel), können relative Pfade über allgemeine Knotenoperationen (ähnlich zum DOM) definiert werden.

2.4.2 XML-Anfrage- und Transformationssprachen

Für die Verarbeitung von Daten ist das Vorhandensein einer Anfragesprache essentiell. Eine XML-Anfragesprache sollte der strukturellen und inhaltsbasierten Suche dienen. Anfragesprachen herkömmlicher Datenmodelle sind aufgrund der XML-Eigenheiten ungeeignet. Das W3C hat im Jahr 1998 einen Workshop veranstaltet, der sich speziell diesem Thema widmete [QL998]. Im Rahmen dieses Workshops wurden verschiedene Vorschläge für XML-Anfragesprachen gemacht. Im Folgenden sollen die aus heutiger Sicht wichtigsten Vorschläge (XQL, XML-QL) und eine allgemeine XML-Transformationssprache (XSLT) näher beschrieben werden.

XSL(T)

Das W3C hat mit der *Extensible Style Sheet Language (XSL)* [W3C98b] eine eigene Style Sheet Sprache für XML verabschiedet. XSL unterstützt neben seinen eigenen Sprachkonstrukten auch Programmelemente und zwar entweder durch ECMAScript [Neta] (standardisiertes Java Script) oder durch von SGML bekannte DSSSL *Flow Object Classes*. Daneben treibt das W3C seit 1996 die Entwicklung von Cascading Style Sheets (*CSS*[LB96]) voran, einer Style Sheet Sprache, die die Darstellung von HTML-Dokumenten ermöglicht. Sowohl mit XSL als auch mit CSS können XML-Dokumente formatiert dargestellt werden. Beide Sprachen nutzen das gleiche zugrunde liegende Formatierungsmodell.

XSL besteht aus zwei Teilen,

1. einer Sprache zur Transformation von XML Dokumenten (XSLT), und
2. einem XML Vokabular zur Spezifikation der Formatierungssemantik (CSS-ähnlich).

Wir werden im Folgenden auf den Transformationsteil von XSL (*XSLT* [W3C99a]) eingehen. XSLT dient der Definition von Style Sheets in der XML Syntax. Ein XSL Style Sheet spezifiziert die Präsentation einer Klasse von XML-Dokumenten. Es gibt unter Benutzung des Formatierungsvokabulars vor, wie eine Instanz dieser Klasse in ein XML-Dokument überführt wird. Im Gegensatz zu anderen XML-Anfrage- und Transformationsvorschlägen war XSLT in seiner ersten Spezifikation nicht relational vollständig. Es besaß weder Joins noch allgemeine Variablen und die ausdrückbaren Transformationen waren eingeschränkt [ABS00].

Mittlerweile wurde dieser Mißstand verbessert und die XSL-Spezifikation entsprechend erweitert.

XSLT ist eine rekursive XML Transformationssprache, deren Programme als eine geordnete Menge von Schablonenregeln (*template rules*) angesehen werden kann, die einen Quellbaum in einen Zielbaum transformieren. Jede Schablonenregel hat ein zugehöriges Selektionsmuster (*selection pattern*) und enthält eine Konstruktionsschablone (*construction rule*). Eine Schablonenregel ermittelt Knoten, welche den Mustern genügen, und konstruiert eine Ausgabe gemäß der Konstruktionsschablone. Während ein Muster mit Elementen des Quellbaums verknüpft wird, wird eine Konstruktionsschablone zur Erzeugung von Teilen des Zielbaums instantiiert. Der Zielbaum ist dabei vom Ursprungsbaum getrennt. Die Struktur des Zielbaums kann sich komplett von der des Quellbaums unterscheiden. Während der Erzeugung des Zielbaums können Elemente des Quellbaums selektiert und umsortiert werden und Strukturen können beliebig hinzugefügt werden.

Die Transformation beginnt mit der Wurzel des Eingabebaums und entscheidet mithilfe der Selektionsmuster, wo die Transformation fortgesetzt wird. Eine XSL-Transformation ist ein wohlgeformtes XML-Dokument, welches dem XSLT-Namespace genügt.

```
<xsl:stylesheet>

<xsl:template match="bookstore">
  <OUTPUT>
    <UL>
      <xsl:apply-templates select="//exemplar/title"/>
    </UL>
    <UL>
      <xsl:apply-templates select="//exemplar/author"/>
    </UL>
  </OUTPUT>
</xsl:template>

<xsl:template match="title">
 <LI> ``<xsl:apply-templates/>'' </LI>
</xsl:template>

<xsl:template match="author">
 <LI> <xsl:apply-templates/> </LI>
</xsl:template>

</xsl:stylesheet>
```

Abbildung 2.13: Ein XSL-Programm

Das Selektionsmuster (`match`-Statement) bestimmt, welche Konstruktionsregel angewendet wird. Eine Konstruktionsregel wird von dem Tag `<xsl:template>` eingeschlossen. Es besteht aus einfachem Ausgabetext und einem Konstruktionsmuster (`select`-Statement). XSL benutzt *XPath*-Ausdrücke, um durch die Dokumentstruktur zu navigieren. Diese Pfade können sowohl abwärts als auch aufwärts durch den Baum führen. Neben Elementen können sie auch Attribute ansprechen.

Wir betrachten das XML-Dokument in Abbildung 2.8, welches eine Instanz der DTD in Abbildung 2.9 ist. Wir wenden das XSL-Programm in Abbildung 2.13 auf das Dokument an. Das Programm besteht aus drei Schablonen. Jede Schablone besteht aus einem Selektionsmuster und einer Konstruktionsregel. In diesem Programm werden die Schablonen auf `bookstore`, `title` und `author` angewendet. In der Schablone für `bookstore` werden die anderen beiden Schablonen mithilfe von Pfadausdrücken angesprochen.

Falls mehrere selektierbare Knoten zur Wahl stehen, werden diese in Präfixreihenfolge des zugehörigen Baums angesprochen. Wenn wir den Style Sheet auf das Dokument in Abbildung 2.8 anwenden, sieht die Ausgabe wie folgt aus:

```
<OUTPUT>
   <UL>
      <LI>``Ordinary Worlds'' </LI>
      <LI>``Lecture Notes, Vol. 23''  </LI>
  </UL>
   <UL>
      <LI> Can Miller </LI>
  </UL>
</OUTPUT>
```

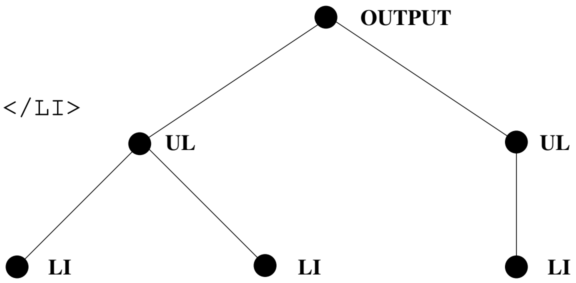


Abbildung 2.14: Eine XSL-Ausgabe und deren zugehöriger Baum

Die bisher benutzten Muster waren ausnahmslos linear. Linearität bedeutet, dass bei der Spezifikation von Elementen, nur deren Attribute und Vorfahrelemente (als der Weg von der Wurzel bis zum Element) herangezogen werden. Dagegen erlauben nichtlineare Muster die Berücksichtigung der Struktur der Elemente unterhalb der angesprochenen Elemente.

Nichtlineare Muster können mithilfe sogenannter Qualifizierer (*qualifiers*), welche von eckigen Klammern (`[...]`) eingeschlossen sind, definiert werden. Ein nichtlineares Muster prüft ausgehend vom Ergebniselement Knoten der unteren Ebenen auf bestimmte Eigenschaften. Z. B. spricht `novel[exemplar]` ein `novel`-Element an, welches ein `exemplar`-Unterelement besitzt. Dagegen bezeichnet `novel/exemplar` ein `exemplar`-Element in einem `novel`.

XSL-Transformationen müssen nicht terminieren. Wenn wir die zweite Schablone in Abbildung 2.13 durch

```
<xsl:template match="title">
      <xsl:apply-templates select="ancestor(bookstore)"/>
</xsl:template>
```

ersetzen, gelangt das Programm in eine nichtterminierende Schleife, sobald es auf ein `title`-Element trifft. Die Anweisung `ancestor(bookstore)` liefert hier den nächsten mit `bookstore` bezeichneten Vorgänger des aktuellen Knoten.

XQL

Obwohl sich zur Definition von Anfragen prinzipiell XSLT eignen würde, hat sich in letzter Zeit *XQL* [RLS98] mehr und mehr durchgesetzt. XQL ist für effiziente Anfragen an große Quellen konzipiert. Dennoch soll es keine „vollständige“ Anfragesprache sein - vielmehr deren „nützlichster Teil“. So unterstützt XQL keine Joins, wobei es in neuer Zeit Vorschläge für Erweiterungen in diese Richtung gab. Bei der Entwicklung standen vier Anwendungsfelder im Vordergrund:

- Anfrage auf einzelne Dokumente (z. B. in einem Browser oder Editor)
- Anfragen auf einer Menge von Dokumenten (z. B. Dokumenterstellung in einem XML-Repository)
- Ansprechen einzelner Objekte in einem Dokument oder darüber hinaus
- XSL Patterns

XQL benutzt XPath-Ausdrücke für die Lokation von Strukturteilen. Es fügt diesen Ausdrücken u.a. Boolesche Logik, Filter und Indizierung hinzu. Somit haben XQL-Ausdrücke Ähnlichkeit zu verallgemeinerten regulären Pfadausdrücken [CCM96]. XQL ist quellorientiert, was bedeutet, dass das Auffinden von Informationen im Vordergrund steht. Es stehen zunächst keine Möglichkeiten zur Formatierung der Anfrageergebnisse zur Verfügung.

Eine XQL-Anfrage ist immer im Kontext eines XML-Dokuments oder eines Teils davon zu sehen. So ist der Ergebnistyp einer Anfrage ohne die Kenntnis der Anfragequelle im allgemeinen nicht ableitbar.

Die Semantik eines XQL-Ausdrucks ist so definiert, dass mit jedem „/“ eine Ebene im XML-Dokument hinabgestiegen wird. Dabei befinden sich auf der nächsttieferen Ebene eines Elements sowohl die Kinderelemente als auch die Attribute (s. Abbildung 2.10). Ein Element wird dabei durch dessen Namen angesprochen, während ein Attribut durch das Zeichen „@“ lokalisiert wird. Sowohl für Elemente als auch für Attribute können Platzhalter verwendet werden. Einzelne Knoten können durch Vergleiche mit Konstanten selektiert werden. Schließlich ist es über Mengenoperationen möglich, Ergebnisse zu verknüpfen.

Betrachten wir die Baumdarstellung eines XML-Dokuments aus Abbildung 2.10, so liefert die Anfrage

```
//exemplar[title='Ordinary Worlds' $and$ author='Can Miller' ]
```

alle „exemplar“-Elemente beliebiger Tiefe, die ein Attribut „author“ mit dem Wert „Can Miller“ und einem „title“ mit dem Wert „Ordinary Worlds“ haben. Dagegen liefert

```
/bookstore/*/exemplar/@author
```

alle `author`-Attribute eines `exemplar`-Elements, welches sich wiederum zwei Ebenen unterhalb des `bookstore`-Wurzelelements befindet.

Zu XQL gibt es unterschiedliche Implementierungen, stellvertretend seien hier die Implementierungen im eXcelon [eC] und die des Projekts PDOM des IPSI/GMD [Tec] zu nennen.

XML-QL

XML-QL [DFF+98] wurde ebenso wie XQL auf dem W3C-Workshop für XML-Anfragesprachen vorgestellt [QL998]. Aus der Perspektive von XML-QL wird ein XML-Dokument als eine Datenbank angesehen. Die Anfrageoperationen sind im Vergleich zu XQL mehr zielorientiert, was sich in der Möglichkeit reichhaltiger Formatierungsmöglichkeiten des Anfrageergebnisses widerspiegelt.

```
WHERE <bookstore>
        <publisher>
        <name> $n </name>
       </publisher>
      </bookstore> in "doc.xml"
CONSTRUCT <new_name> $n </new_name>
```

Die Anfrage bindet die Variable n an jedes Vorkommen eines Namens. Entsprechend wird im Ausgabeteil für jeden Namen ein Element `new_name` erzeugt.

```
  <new_name> Springer </new_name>
  <new_name> Warner </new_name>
```

Wie man sieht, ist das Ergebnis nicht zwangsläufig ein korrektes XML-Dokument. In diesem Fall ließe sich das durch Hinzufügen eines entsprechenden Wurzelelements aber einfach erreichen.

Vergleich von XML-Anfragesprachen

In [DFF+99] werden Unterschiede und Gemeinsamkeiten der einzelnen Anfragesprachen herausgearbeitet. Neben den hier vorgestellten Sprachen wurde noch LOREL [AQM+97] in den Vergleich einbezogen. Als Vergleichskriterien sind u. a.

- Präzise Semantik
- Wiederformulierbarkeit von Anfragen für andere Datenmodelle
- Unterstützung der wichtigsten Anfrageoperationen

- Anfragesemantik eindeutig und orthogonal
- Ausnutzung von Schemainformationen
- Erhaltung der Ordnung
- XML-Konformität des Ergebnisses
- Einbettung von XML-Teilen

angewendet worden.

Das Ergebnis dieses Vergleichs ist in der folgenden Tabelle dargestellt:

	Präzise Semantik	Umschreibbarkeit	XML Ausgabe	Alle Anfrageoperationen	Kompakte Semantik	Schemaausnutzung	ordnungserhaltend	XML-Darstellung	XML Einbettung
XSLT	ja	nein	ja	nein	ja	nein	ja	ja	ja
XML-QL	ja	ja	ja	ja	ja	nein	ja	nein	ja
XQL	ja	nein	ja	nein	ja	nein	ja	nein	ja
LOREL	ja	ja	nein	ja	nein	nein	ja	nein	ja

2.4.3 Resource Description Framework (RDF)

Das *Resource Description Framework (RDF)* [W3C99b] ist ein Modell zur Beschreibung von Metadaten. Es ermöglicht den Austausch maschinenlesbarer Informationen zwischen Anwendungen über das Web. RDF unterstützt Möglichkeiten für die automatische Verarbeitung von Webressourcen. Dabei können RDF-Metadaten in unterschiedlichsten Anwendungsszenarien zum Einsatz kommen, u. a.

- bei der Suche nach Ressourcen, indem es bessere Suchmaschinenunterstützung liefert.
- bei der Beschreibung von Mengen von Seiten, welche ihrerseits eine logische Seite bilden, indem es deren Zusammenhang spezifiziert.
- für intelligente Softwareagenten, um Informationsaufteilung und -austausch zu unterstützen.
- zur Beschreibung der geistigen Eigentümerrechte von Webseiten.

RDF ist als Anwendung von XML konzipiert. Es spezialisiert das XML-Modell und die XML-Syntax für die Beschreibung von Ressourcen.

RDF Model
RDF definiert ein einfaches, obgleich mächtiges Modell zur Beschreibung von Ressourcen.

Die Grundidee von RDF ist einfach: Eine identifizierbare, adressierbare Ressource soll durch eine Auswahl von Eigenschaften (Größe, Name, Autor), jede mit ihrem zugehörigen Wert, beschrieben werden. Dabei kann eine Eigenschaft wiederum eine weitere Ressource sein. Dieses erlaubt dem Modell durch die Verbindung von Informationsquellen unbeschränkt zu wachsen.

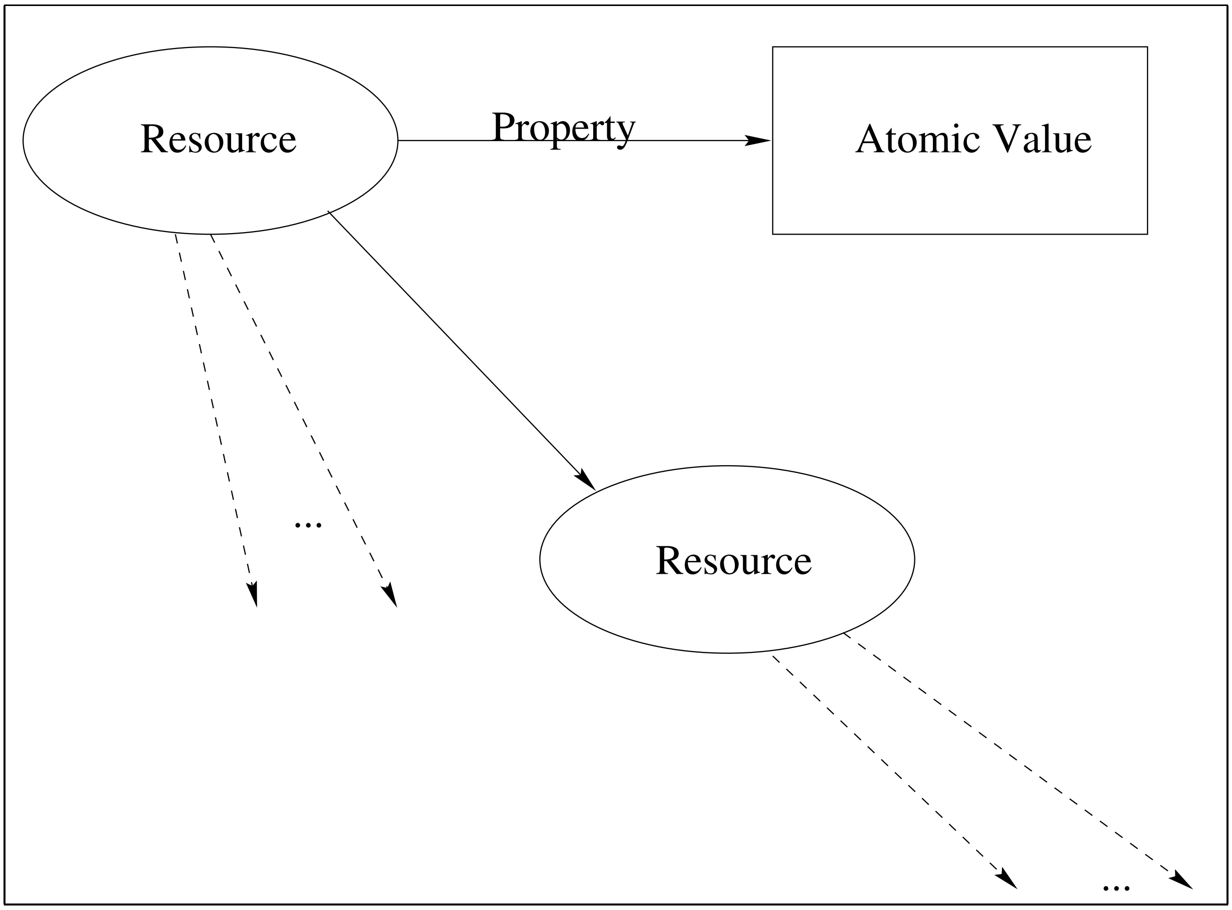


Abbildung 2.15: Eine generische RDF-Beschreibung

Wir werden die Benutzung von RDF anhand eines Beispiels erläutern. Zunächst wollen wir ausdrücken, dass „Ralf Behrens, behrens@ifis...“ der Autor der Arbeit ist, welche durch die URI *http://doc.txt* ansprechbar ist. Für diesen Fall würde es verschiedene natürlichsprachliche Möglichkeiten geben. Dagegen haben wir bzgl. des RDF-Modells eine mehr oder weniger eindeutige Darstellung (d. h. bis auf Wortumbenennungen), wie sie in Abbildung 2.16 zu sehen ist.

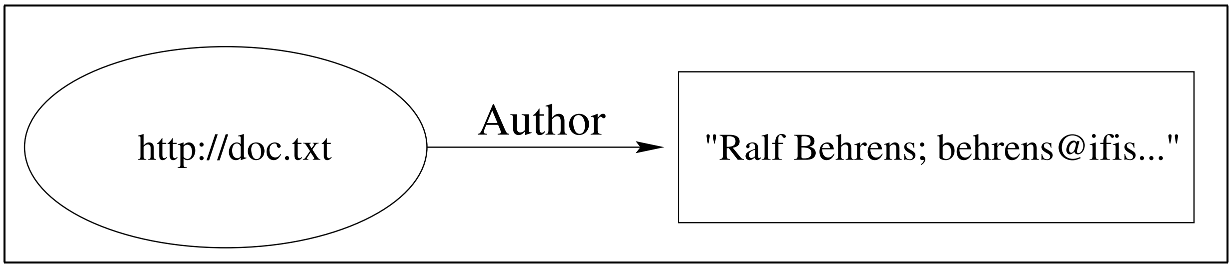


Abbildung 2.16: Konkrete RDF-Beschreibung

Im zweiten Fall geht es wiederum um die Modellierung der selben Situation, mit dem Unterschied, dass wir den Autor genauer beschreiben wollen. Dieses soll dadurch geschehen, dass wir seinen Vornamen, Nachnamen und seine eMail-Adresse explizit kennzeichnen. Wie Abbildung 2.17 zeigt, führen wir hierzu eine weitere Ressource ein.

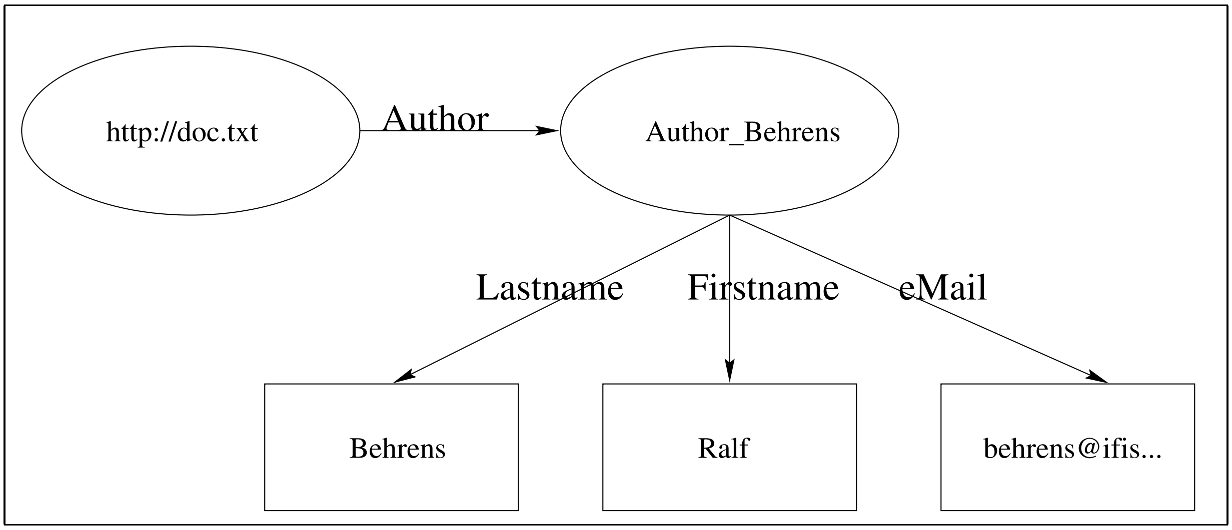


Abbildung 2.17: Verfeinerung der obigen RDF-Beschreibung

Wir können so die Beschreibung spezieller Eigenschaften beliebig erweitern. Beispielsweise wäre es möglich, auch für die eMail-Adresse eine eigene Ressource einzuführen. Wie wir sehen, erlaubt RDF die Erzeugung und Beschreibung von Ressourcen auf unterschiedlichen Granularitätsebenen.

Schließlich lässt RDF die „Beschreibung von Beschreibungen“ zu. Dieses kann in Fällen hilfreich sein, wo es um Sammlungen von Objekten geht. Beispielsweise können wir beschreiben, dass eine Person der Autor unterschiedlicher Dokumente ist.

RDF Syntax

Die RDF-Syntax zielt auf die Speicherung von im RDF-Modell ausgedrückten Informationen innerhalb textueller Dateien hin. Diese sollen unter Anwendungen austauschbar sein. Als syntaktische Grundlage wird XML verwendet. RDF gibt dabei eine formale Struktur vor, um die konsistente Repräsentation von Semantik zu gewährleisten.

RDF benutzt die bereits angesprochenen XML-Namespaces. Die syntaktische Darstellung des vorhergehenden Beispiels mittels der RDF-Syntax sieht wie folgt aus:

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/
                    22-rdf-syntax-ns#">
   <rdf:Description rdf:HREF = "http://doc.txt">
     <Author> Ralf Behrens, behrens@ifis... </Author>
   </rdf:Description>
</rdf:RDF>
```

Im folgenden, weitergehenden Beispiel (mit zusätzlichen deskriptiven Informationen bzgl. des Autors) werden ähnliche syntaktische Konstrukte benutzt.

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/
                    02/22-rdf-syntax-ns#">
  <rdf:Description href = "http://doc.txt">
     <Author href ="#Author_Behrens"/>
  </rdf:Description>

  <rdf:Description about="http://www.anywhere.org/document"
                   ID="Author_Behrens">
       <Firstname> Ralf </Firstname>
       <Lastname> Behrens </Lastname>
       <eMail> behrens@ifis.mu-luebeck.de </eMail>
  </rdf:Description>
</rdf:RDF>
```

RDF Schema

Ein RDF-Schema wird zur Definition von „*Vokabularen*“ benutzt. Ein Vokabular ist eine Menge von semantischen Eigenschaftstypen, welche von einer Gemeinschaft festgelegt werden. Ein RDF-Schema definiert sowohl die gültigen Eigenschaften innerhalb einer RDF-Beschreibung als auch Charakteristiken oder Beschränkungen der Eigenschaftstypen.

Da RDF für unterschiedliche Gemeinschaften die Möglichkeit der Definition von Bedeutungen liefert, ist es wichtig, diese Bedeutungsdefinition austauschbar zu machen. So kann z. B. die Eigenschaft „Autor“ eine engere oder weitergehende Bedeutung in Abhängigkeit der Bedürfnisse der jeweiligen Gemeinschaft haben. Daher ist es schwierig, wenn unterschiedliche Gemeinschaften identische Eigenschaftstypen zur Beschreibung andersartiger Dinge benutzen. Um diesem vorzubeugen, identifiziert RDF Eigenschaftstypen durch die Benutzung von XML Namespaces. Der XML Namespace-Mechanismus dient hier der Identifizierung der Semantik und der Konventionen, welche der Benutzung zugrunde liegen, indem es den Ersteller des Vokabulars identifiziert.

Eine menschen- und maschinenlesbare Beschreibung erhält man durch Dereferenzierung der Schema-URI. Wenn das Schema maschinell verarbeitbar ist, kann es für eine Anwendung möglich sein, die Semantik der Eigenschaftstypen zu „lernen". Der Begriff „lernen" ist so zu verstehen, als dass Namen mit Definition in Verbindung gebracht werden. Um ein bestimmtes RDF-Schema zu verstehen, müssen wir die Semantik jeder beschriebenen Eigenschaft verstehen. Um eine automatisierte Interpretation zu ermöglichen, werden RDF-Schemata selbst basierend auf dem RDF-Datenmodell strukturiert. Daher wird eine Anwendung in der Lage sein, die Beschreibung in Eigenschaftstypen und korrespondierende Werte zu überführen.

2.5 Fazit

Bei Betrachtung der unterschiedlichen Modellierungskonzepte stellt XML eine interessante Alternative zu herkömmlichen Lösungen dar.

XML präsentiert eine Sammlung von Standards und Ideen, deren Konzepte zwar allesamt nicht neu sind, aber in dieser Kombination eine allgemeine Grundlage schaffen können. Aufgrund der im Medienarchiv auftretenden, unterschiedlich stark strukturierten Informationen, erscheint die Verwendung von XML als sinnvoll. In naher Zukunft könnte XML als plattformübergreifendes Format proprietäre Formate ablösen. Datenbestände würden sich zwar durch ein zusätzliches Markup vergrößern; es ist aber denkbar, dass der entstehende Overhead durch speziell auf XML zugeschnittene Komprimierungsverfahren auf ein Mindestmaß reduziert werden kann [LS00]. Wie aus den vorangegangenen Erläuterungen ersichtlich ist, bietet XML gegenüber den herkömmlichen Datenmodellen verschiedene Vorteile, im Wesentlichen

- ein menschen- und maschinenlesbares Datenformat.
- ermöglicht es die Behandlung semistrukturierter Daten.
- unterstützt es multimediale Aspekte.
- gibt es eine Vielzahl vorhandener Anwendungen.

Gerade im Zusammenspiel mit der plattformunabhängigen Sprache Java kommt XML eine zentrale Bedeutung als universales Datenbeschreibungs- und Austauschformat zu.

Aus diesem Grund soll XML im nächsten Kapitel näher untersucht und analysiert werden. Wir werden Schwachstellen aufzeigen und eigene Erweiterungen vorstellen.

3

Analyse und Erweiterung von XML

„Die Metasprache XML wird für das Internet eine ähnliche Veränderung bringen, wie die Einführung des Telefons für die Gesellschaft.“

ZONA RESEARCH,
MARKTFORSCHUNGSINSTITUT

Im vorherigen Kapitel wurde XML bereits ansatzweise vorgestellt. Wir werden XML in diesem Kapitel tiefergehend analysieren. Diese Analyse soll dazu dienen, wesentliche Spezifika bzgl. Ausdrucksstärken von XML und den schon in SGML benutzten DTDs offenzulegen.

3.1 Überblick

XML (*Extensible Markup Language*) [W3C98a] ist eine Sprache zur Beschreibung von Dokumentsprachen. Bei der Abstraktion von XML wird klar, dass XML ein Hybrid aus erweitert kontextfreien Grammatiken und attributierten Grammatiken [Knu68] ist, wobei ihm jedoch die Möglichkeit von Berechnungen über Attributen fehlt. XML benutzt eine konkrete Syntax, um die logischen Elemente eines Dokuments zu beschreiben und es bedient sich einer abstrakten Syntax, um die Beziehungen zwischen logischen Komponenten auszudrücken.

Da XML-Dokumente sowohl Schema- als auch Instanzinformationen vereinigen, sollen die verschiedenen, korrespondierenden Begriffe zunächst gegenüber gestellt werden.

Schema-Ebene	**Instanz-Ebene**
logische Struktur	physische Struktur
DTD	Dokumentinstanz
Elementtyp-Deklaration	Elementinstanz
Inhaltsmodell	Inhalt
Attributtyp-Deklaration	Attributwert
Attributart	—
Entity	Entityreferenz
Element	
Dokument	

Im Folgenden werden die wichtigsten Bestandteile von XML kurz zusammengefasst. Dabei werden zum besseren Verständnis zu den deutschen Übersetzungen die englischen Ursprungsbezeichnungen genannt. Diese Übersicht dient als Grundlage für die spätere Abstraktion.

Dokument

Ein Dokument (*document*) besteht aus einem Element (*element*), welches weitere Elemente enthalten kann und einer optionalen *Document Type Definition* (DTD), die die logische Struktur eines Dokuments definiert. Jedes Element hat einen Namen, eine Liste von Attribut- und Wertepaaren und möglicherweise einen Inhalt (*content*) bestehend aus Elementen oder Text (*#PCDATA*). Elemente ohne Inhalt heißen leer (*EMPTY*). Ein Element kann eine ID besitzen, die durch ein Attribut mit dem Typ *ID* beschrieben wird. Entsprechend kann es auf ein oder mehrere Elemente über ein Attribut vom Typ *IDREF/IDREFS* verweisen.

Elemente

Elemente bilden die logischen Bestandteile der Struktur eines XML-Dokuments. Eine *Elementtyp-Deklaration* beschreibt für jedes Element durch ein Inhaltsmodell (*content model*) den möglichen Inhalt (*content*). Es wird zwischen zwei Arten von Element-Inhaltsmodellen unterschieden:

1. Leere Elemente haben das Inhaltsmodell *EMPTY* und bestehen aus einem Namen und einer Liste von Attribut- und Wertepaaren. Kein Attribut darf in einer Liste doppelt vorkommen. Nachfolgend werden die möglichen Typen der Attribute beschrieben.

2. Nichtleere Elemente bestehen aus einem Namen, einer Liste von Attribut- und Wertepaaren und einem Inhalt (*content*). Der Inhalt kann eine Folge von Elementen (*element content*), ein String (*character content*) oder beides (*mixed content*) sein.

Darüber hinaus gibt es noch das Inhaltsmodell *ANY*, das die Einsetzung beliebiger Elemente (die in der DTD definiert sind) erlaubt, syntaktisch gesehen jedoch nur eine Abkürzung ist.

Man beachte, dass ein Element, dessen Inhalt eine leere Liste von Elementen ist, nicht als *EMPTY*-Element gilt.

Wenn ein Element nicht leer ist, kann das Inhaltsmodell wie folgt aussehen:

1. *element content*: Die Reihenfolge und Anzahl der Kind-Elemente wird durch einen regulären Ausdruck über Elementnamen beschrieben.
2. *mixed content*: Eine Folge von Elementen und Character-Daten (*#PCDATA*). In diesem Fall lautet das einzige erlaubte Inhaltsmodell

$$(\#PCDATA|element_1|\dots|element_n)*$$

3. *character content*: Der Inhalt des Elements ist ein (geparster) String. Parsing bedeutet dabei, dass auftretende Platzhalter (sogenannte ENTITIES, s. u.) ersetzt werden.

Eine Elementtyp-Deklaration hat somit die Form

<!`ELEMENT` *Elementname Inhaltsmodell* >

wobei das Inhaltsmodell einer der drei oben genannten Alternativen entspricht.

Alle in der Dokumentinstanz benutzten Elemente müssen deklariert worden sein, selbst wenn sie nur als Kinder eines Elements mit dem Inhaltsmodell *ANY* vorkommen.

Als Beispiel für ein XML-Dokument soll wiederum das Buchladenbeispiel aus Abbildung 2.8 mit dessen DTD in Abbildung 2.9 dienen.

Document Type Definition (DTD)

Eine DTD definiert für jedes Element in der Dokumentinstanz eine Elementtyp- (s.o.) und optional eine Attributlisten-Deklaration (s.u.). Ein Dokument heißt gültig (*valid*), wenn es die folgenden Eigenschaften erfüllt:

- Kein Paar von Elementen hat die gleiche ID.
- Jede ID, die als Attribut auftaucht, ist die ID eines Elements im Dokument.
- Das Dokument enthält eine DTD und erfüllt diese.

Ein Dokument erfüllt seine DTD, wenn die Dokumentinstanz entsprechend der DTD aufgebaut ist. Ein Dokument, welches unabhängig von einer (etwaigen) DTD eine korrekte Schachtelung besitzt und die syntaktischen Regeln von XML einhält, wird als wohlgeformt (*well-formed*) bezeichnet. Diese Eigenschaft ist somit schwächer als die der Gültigkeit.

Eine *Attributlisten-Deklaration* bestimmt für jedes Element *name* eine Liste von Attributen mit den zugehörigen Typen.

Attributtypen und -arten

XML unterstützt verschiedene Attributtypen:

- *ID*: Der Wert eines ID-Attributs muss eindeutig innerhalb eines Dokuments sein. Jedes Element kann nur höchstens ein ID-Attribut haben, welches auch `id` heißen sollte.
- *IDREF*: Der Wert eines IDREF-Attributs muss der Wert des ID-Attributs eines Elements im selben Dokument sein. Man beachte, dass es nicht möglich ist, den Namen des referenzierten Elements anzugeben.
- *IDREFS*: Eine Liste von IDREF-Werten, die durch Leerzeichen getrennt sind.
- *ENTITY*: Der Wert ist ein Identifikator einer externen (nicht zwangsläufig XML-konformen) Ressource.
- *ENTITIES*: Eine Liste von ENTITY-Werten.
- *CDATA*: Ein einfacher String, der nicht geparst wird (*character data*).
- *enumerated types*: Aufzählungs-Typ, der eine Liste von möglichen Werten enthält.

Attribute haben darüber hinaus eine Attributart. Es gibt vier unterschiedliche Arten:

- *REQUIRED*: Der Wert dieses Attributs ist obligatorisch und muss bei der Benutzung des Elements zugewiesen werden.
- *IMPLIED*: Der Wert dieses Attributs ist optional und wird bei der Verarbeitung des Elements u. U. berechnet. Ein Beispiel für einen berechneten Wert ist die Größe des Dokuments im Browser-Fenster. Es wären aber auch Dinge wie Vererbungsvorschriften oder externe Funktionen denkbar.
- *FIXED*: Der Wert des Attributs ist fix und kann im Dokument nicht mehr verändert werden.
- „Standardwert“: Dem Attribut wird ein Vorgabewert zugewiesen, der im Dokument überschrieben werden kann.

Eine Attributlisten-Deklaration sieht in der XML-Syntax wie folgt aus:

`<!ATTLIST` *Elementname … Attributname Attributtyp Attributart* …>

Entities

XML bietet einen Mechanismus zur Definition von konstanten Namen und Makros. Dieser Mechanismus basiert auf der Deklaration unterschiedlicher ENTITIES. Ein *ENTITY* kann benutzt werden

- zur einfachen Ersetzung innerhalb einer DTD (*Parameter-Entity*) oder innerhalb einer Dokumentinstanz (*allgemeines ENTITY*).
- zur Referenzierung von externen Daten mittels URLs (*externes ENTITY*).

Ein *ENTITY* wird in der Dokumentinstanz mit einem vorangestellten & und in der DTD mit einem vorangestellten % verwendet.

ENTITIES dienen in den meisten Fällen als Platzhalter. So ist es hiermit möglich, vordefinierte Element-Inhalte mehrfach zu verwenden oder auch konstante Strings festzulegen, die bei der Verarbeitung automatisch ersetzt werden. Beispielsweise werden deutsche Umlaute, welche keine Standard-ASCII-Zeichen sind, als *ENTITIES* definiert, etwa „Ä“ als `Ä`.

3.2 Abstraktion von XML

Im Gegensatz zur ursprünglichen Intuition, XML als Dokumentmodell zu entwickeln, soll es hier als allgemeines Datenbeschreibungsmodell betrachtet werden. Daher wird nicht mehr von XML-Dokumenten, sondern allgemeiner von XML-Objekten die Rede sein. Dieses ist kein reiner Benennungsunterschied, sondern spiegelt auch eine universelle Anwendbarkeit wider.

Obwohl die Notwendigkeit eines formalen Modells für XML-Daten, welches von der konkreten Syntax abstrahiert, erkannt ist [Mai98], gibt es lediglich Ansätze, die einzelne Aspekte von XML anschneiden. Ein allgemeines Modell, welches XML umfassend behandelt, fehlt bisher. Ein solches Modell kann als Grundlage für theoretische Betrachtungen bzgl. Komplexität und Berechenbarkeit und als Basis für einen Anfrage-Kalkül oder eine Anfrage-Algebra dienen.

Ein Modell, das alle Eigenschaften von XML in einem monolithischen System berücksichtigt, wäre kaum noch überschaubar und algorithmisch schwer zu handhaben. Daher soll die Modellbildung Schritt für Schritt geschehen, um so die Eigenschaften auf den einzelnen Ebenen gezielt untersuchen zu können [BBLL99, Beh00a]. Wir werden Parallelen zu erweitert kontextfreien Grammatiken und regulären Sprachen aufzeigen und entsprechende Ergebnisse übertragen. Es werden wichtige Entscheidbarkeits- und Komplexitätsaussagen hergeleitet.

XML-Objekte können als Ableitungsbäume einer erweitert kontextfreien Grammatik gesehen werden. Erweitert kontextfreie Grammatiken (*eCFG*) sind kontextfreie Grammatiken (*CFG*) mit regulären Ausdrücken über Grammatiksymbolen auf den rechten Seiten der Regeln. Es ist bekannt, das eCFGs die gleiche Sprachklasse wie CFGs beschreiben. Wenn jedoch Grammatiken für die Modellierung von Dokumenten benutzt werden und man somit Ableitungsbäume betrachtet, werden wichtige Unterschiede ersichtlich.

Es gibt viele wichtige Dinge, welche aufgrund der Allgemeinheit kontextfreier Grammatiken nicht gelöst werden können. So sind kontextfreie Grammatiken z. B. nicht abgeschlossen unter Schnitt, was bedeutet, dass sich i. d. R. keine kontextfreie Grammatik für die Schnittmenge zweier kontextfreier Sprachen finden lässt.

DB	:	MitarbeiterListe	DB	:	Mitarbeiter$^+$
MitarbeiterListe	:	Mitarbeiter, MitarbeiterListe	Mitarbeiter	:	Vorname$^+$, Nachname
MitarbeiterListe	:	Mitarbeiter	Vorname	:	$(a + \ldots + z)^+$
Mitarbeiter	:	VornamenListe, Nachname	Nachname	:	$(a + \ldots + z)^+$
VornamenListe	:	Vorname, VornamenListe			
VornamenListe	:	Vorname			
Vorname	:	LetterListe			
Nachname	:	LetterListe			
LetterListe	:	Letter, LetterListe			
LetterListe	:	Letter			
Letter	:	$a + \ldots + z$			

Abbildung 3.1: CFG und eCFG

Vergleicht man die Grammatiken in Abbildung 3.1 mit ihren Ableitungsbäumen in Abbildung 3.2, so sieht man, dass für die Listenbeschreibung mit einer CFG zusätzliche Symbole ohne weitere Bedeutung eingeführt werden müssen. Eine Abbildung von erweitert kontextfreien Grammatiken auf kontextfreie Grammatiken ist zwar möglich, würde aber die Gestalt der Bäume verändern.

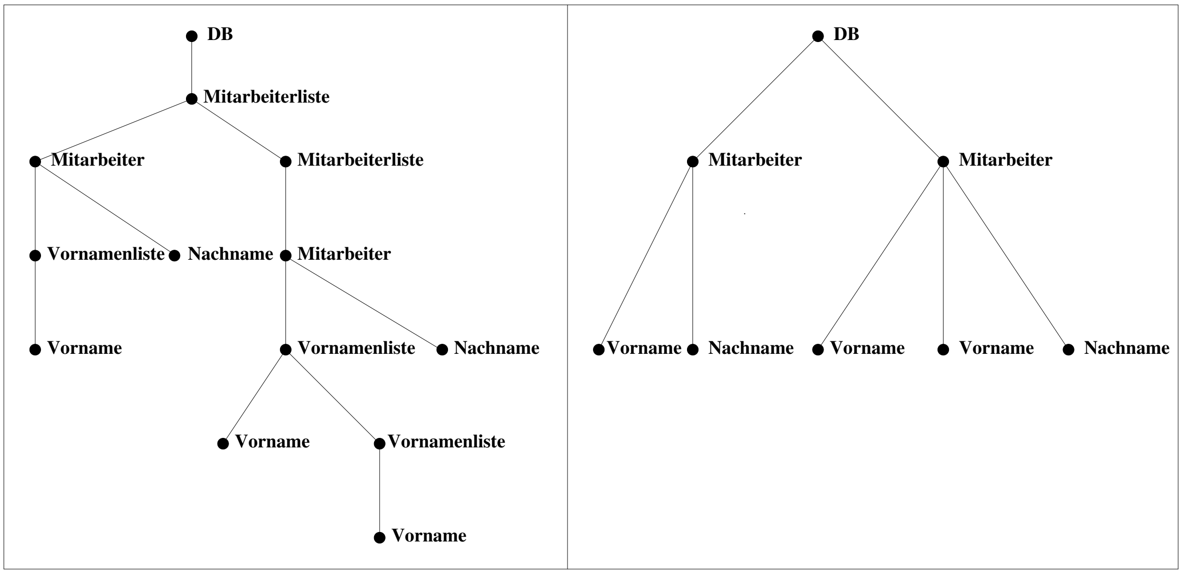


Abbildung 3.2: Ableitungsbäume zu den Grammatiken in Abbildung 3.1

Ein zu beachtender Unterschied zwischen CFG- und eCFG-Ableitungsbäumen ist die Tatsache, dass nur CFGs einen beschränkten Kinderknotengrad (*ranking*) besitzen. Daher können Knoten in eCFGs beliebig viele Kinderknoten besitzen. Während Bäume mit Rangreihenfolge eingehend untersucht wurden, haben Bäume ohne Rangreihenfolge erst durch SGML und XML neues Interesse erweckt [BKW98b]. Der Zusammenhang zwischen kontextfreien

Grammatiken, Ranking und Attributierung ist in Abbildung 3.3 dargestellt.

GRAMMATIK	attributiert	nicht attributiert
Ranking	Attr. Grammatik	CFG
kein Ranking	–	eCFG

Abbildung 3.3: Beziehung zwischen Grammatik und Attributierung

3.3 Präliminarien

Um ein einheitliches Begriffsgerüst zu schaffen, werden die verwendeten Notationen hier kurz erläutert.

Die Kenntnis der Grundbegriffe formaler Sprachen wird im Folgenden vorausgesetzt[HU94]. Hierzu gehören insbesondere *reguläre Ausdrücke* (*RA*), kontextfreie und erweitert kontextfreie Grammatiken und Sprachen (*CFG/eCFG, CFL*), *deterministisch kontextfreie Sprachen* (*DCFG*), deterministische und nichtdeterministische *endliche Automaten* (*DFA, NFA*), deterministische und nichtdeterministische *Kellerautomaten* (*DPDA,PDA*), Homomorphismen und Substitutionen.

Benutzt werden die Abgeschlossenheit regulärer und kontextfreier Sprachen unter Homomorphismen, sowie Schnitt und Ersetzung mit Sprachen der gleichen und anderer Klassen.

Mit Σ bezeichnen wir im weiteren Verlauf ein endliches Alphabet, mit ε das leere Wort, mit r_Σ einen regulären Ausdruck über einem Alphabet Σ und mit $L(r_\Sigma)$ die durch den regulären Ausdruck beschriebene Sprache. Sofern Σ aus dem Zusammenhang ersichtlich ist, kann es auch weggelassen werden.

Reguläre Ausdrücke über einem endlichen Alphabet Σ sind rekursiv definiert. Jeder reguläre Ausdruck r über Σ beschreibt eine reguläre Sprache $L(r) \subseteq \Sigma^*$:

Definition 3.1 (Reguläre Ausdrücke) *Gegeben sei ein endliches Alphabet Σ. Reguläre Ausdrücke werden induktiv definiert:*

1. *Einfache reguläre Ausdrücke*
   - *Jedes Zeichen $e \in \Sigma$ ist RA und beschreibt die Sprache $\{e\}$.*
   - *Das leere Wort ε ist RA und beschreibt die Sprache $\{\varepsilon\}$.*
   - *Die leere Menge $\emptyset$ ist RA und beschreibt die Sprache $\emptyset$.*
2. *Zusammengesetzte reguläre Ausdrücke*

   Sind r und r' RAs mit den Sprachen L bzw. L', dann gilt:

- (r, r') *ist die Konkatenation und beschreibt die Sprache* $\{ee' | e \in L, e' \in L'\}$.
- $(r + r')$ *ist die Vereinigung und beschreibt die Sprache* $L \cup L'$.
- (r^*) *ist die Kleenesche Hülle und beschreibt die Sprache* $\{e_1 \dots e_n | n \in N_0 \wedge e_1, \dots, e_n \in L\}$.
- $(r?)$ *ist gleichbedeutend mit* $r + \varepsilon$.
- (r^+) *ist gleichbedeutend mit* $r^* + r$.

Klammerungen können entfallen, falls der entstehende Ausdruck eindeutig bleibt. Zum Teil werden vereinfachend reguläre Ausdrücke und die durch sie beschriebenen regulären Sprachen synonym behandelt. Wir bezeichnen reguläre Sprachen über einem endlichen Alphabet auch als textuelle Sprachen. Die Menge aller regulären Sprachen wird mit *REG* bezeichnet.

Für die Modellierung von Dokumenten werden *Bäume* verwendet [Knu73]. Ist n ein Knoten in einem Baum t, so ist *label*(n) mit *label*$(n) \in \Sigma$ der Bezeichner des Knotens. Für eine Folge von Knoten $n_1, \dots, n_k$ liefert $name(n_1, \dots, n_k)$ die Folge der Knotenbezeichner. Der Teilbaum mit der Wurzel n ist $tree(n)$ und die Wurzel von t ist $root(t)$. Die Funktion $children(n)$ liefert die Menge aller Kinder von n in t und $leaves(t)$ die Menge der Blätter des Baums. Die Menge aller Knoten im Baum ist $nodes(t)$. Der Ausdruck $t[n, t']$ liefert einen Baum, der durch Ersetzung des Teilbaums $tree(n)$ durch t' in t entstanden ist. Eine Menge von Bäumen wird Baumsprache genannt.

3.4 Struktur-Modell

Wir werden im Folgenden lediglich die Struktur von XML-Objekten betrachten. Hierbei werden terminale Strings nicht berücksichtigt. Dieses hat den Hintergrund, dass Stringelemente auch als Attributwerte umgesetzt werden können und somit strukturell nicht von Bedeutung sind. Auch der in XML benutzte *mixed content* lässt sich durch Hinzufügen eines entsprechenden Elements ersetzen.

Eine XML-Instanz kann als ein knotenmarkierter, geordneter Baum gesehen werden. Jeder Knoten steht für ein XML-Element, wobei der Knoten mit dem Elementnamen markiert ist. Die Folge der Namen der Kinderknoten beschreibt den Elementinhalt (*content*) eines konkreten XML-Elements.

Definition 3.2 (xtree) *Wir beschreiben einen xtree über einem endlichen Alphabet* Σ *(Elementnamen) als einen endlichen Baum, in dem jeder Knoten mit einem Elementnamen aus* Σ *markiert ist und die Liste der Kinder eines Knotens geordnet ist.*

Da jede XML-Instanz eine verschachtelte Struktur besitzt, lässt sich ihr eindeutig ein *xtree* zuordnen (Abbildung 3.4). Wir werden im Folgenden eine verkürzte Version des XML-Dokuments aus Abbildung 2.8 verwenden.

```
<bookstore>
 <novel>
   <exemplar>
    <title>Winter</title>
    <author>Meyer</author>
   </exemplar>
 </novel>
 <journal>
   <exemplar>
    <title>Focus</title>
   </exemplar>
 </journal>
</bookstore>
```

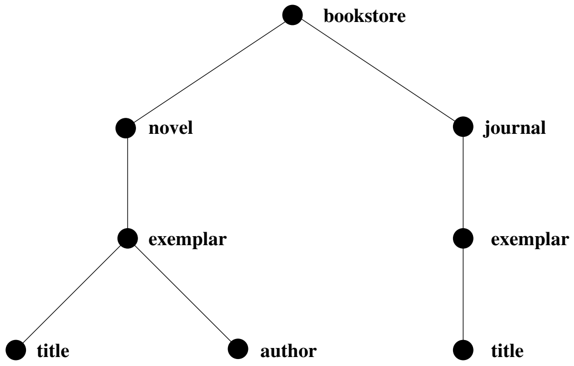


Abbildung 3.4: Beispiel für eine XML-Instanz und deren korrespondierenden *xtree*

Neben *xtree*s, die konkrete Instanzen repräsentieren, werden schematische Informationen benötigt, die diese Instanzen klassifizieren. Dieses geschieht durch Regeln und Regelmengen. Regeln (*xrule*s) werden verwendet, um für Elemente die möglichen Kinderfolgen innerhalb eines Baums zu definieren, während Regelmengen (*xscheme*s) mithilfe der *xrule*s ganze Baumstrukturen beschreiben.

Definition 3.3 (xrule) *Eine xrule über Σ ist ein Tupel $\langle e : type(e)\rangle$, wobei $e \in \Sigma$ und $type(e)$ ein regulärer Ausdruck über Σ ist. Während e als Regelkopf bezeichnet wird, ist $type(e)$ der Regelrumpf.*

Eine *xrule* ist das Gegenstück zu einer Elementtyp-Deklaration in einer XML-DTD. Betrachtet man den Regelrumpf $(type(e))$ separat, so beschreibt dieser die Sprache $L(type(e))$.

Beispiel 3.4 *Ein „exemplar“ wird, wie es im obigen xtree der Fall ist, durch „title“ und „author“ oder nur „title“ beschrieben:*

$$\boxed{\text{exemplar} \quad : \quad \text{title, author?}}$$

Bemerkung: Bei der Verwendung von regulären Ausdrücken entspricht das Inhaltsmodell *EMPTY* dem Regelrumpf mit der leeren Menge $\emptyset$ und *ANY* dem Regelrumpf $(e_1 + \ldots + e_n)^*$ mit $\Sigma = \{e_1, \ldots, e_n\}$.

XML-Inhaltsmodelle (*content models*) sind eindeutig. Es ist aber wichtig zu bemerken, dass sich der Begriff der „Eindeutigkeit“ in XML von dem der Formalen Sprachen unterscheidet. Daher werden wir zunächst diese beiden Konzepte definieren und differenzieren. Um die unterschiedlichen Positionen des gleichen Symbols in einem regulären Ausdruck zu kennzeichnen, markieren wir diese mit Indizes. So sind beispielsweise $(a_1 + b_1)^*a_2(a_3b_2)^*$ und $(a_4 + b_2)^*a_1(a_5b_1)^*$ Markierungen des Ausdrucks $(a + b)^*a(ab)^*$ [BKW98a]. Für jeden

Ausdruck r über Σ wird eine Markierung mit r' bezeichnet. Eindeutigkeit im Sinne von [BEGO71] bedeutet nun, dass jedes Kind eines Knoten n mit *label*$(n) = e$, welches von einer *xrule* $\langle e : type(e)\rangle$ erzeugt wurde, genau einem Symbol in $type(e)$ zugeordnet werden kann.

Behauptung 3.5 (Eindeutigkeit eines regulären Ausdrucks) *Für jede reguläre Sprache R können wir einen eindeutigen regulären Ausdruck r finden, so dass $L(r) = R$ [BEGO71].*

Dieses folgt daraus, dass sich für jede reguläre Sprache jeweils ein (bis auf Umbenennung von Zuständen) eindeutiger und minimaler endlicher Automat finden lässt.

Betrachten wir z. B. den Ausdruck $(a+b)^*aa^*$. Wenn wir unterschiedliche Positionen desselben Symbols mit Indizes markieren, erhalten wir $(a_1 + b_1)^*a_2a_3^*$. Jetzt stehen drei indizierte Worte, nämlich $a_1a_2a_3$, $a_1a_1a_3$, und $a_2a_3a_3$ für dasselbe Wort aaa. Somit ist $(a+b)^*aa^*$ mehrdeutig. Es gibt jedoch einen eindeutigen regulären Ausdruck $(a + b)^*a$, welcher die gleiche Sprache wie $(a + b)^*aa^*$ beschreibt.

Im Gegensatz zu [BEGO71], beschreibt der Begriff der „Eindeutigkeit" in XML ein „one-symbol-lookahead-Eindeutigkeit". In [BKW98a] wird diese Eindeutigkeit als *„1-Eindeutigkeit"* bezeichnet.

Definition 3.6 (1-Eindeutigkeit) *Ein regulärer Ausdruck r is 1-eindeutig, genau dann wenn für alle Wörter u, v, w über dem Alphabet Σ und alle Symbole x, y in Σ, die Bedingungen $uxv, uyw \in L(r')$ und $x \neq y$, $x^\natural \neq y^\natural$ implizieren (dabei bezeichnet ein mit $^\natural$ annotiertes Wort dessen unindiziertes Gegenstück). Eine reguläre Sprache ist „1-eindeutig" wenn sie von einem 1-eindeutigen regulären Ausdruck beschrieben werden kann.*

Dieses bedeutet mit anderen Worten, dass es für jedes Wort w, welches durch einen 1-eindeutigen regulären Ausdruck r beschrieben wird, genau ein indiziertes Wort w' gibt, welches Symbol für Symbol ohne weitere Vorausschau erzeugt werden kann. Betrachten wir den Ausdruck $(a + b)^*a$ welcher zu $(a_1 + b_1)^*a_2$ indiziert wird. Im Wort baa können wir nach dem Symbol b an der Position b_1 nicht entscheiden, ob wir das darauffolgende a mit der Position a_1 oder a_2 in Verbindung bringen sollen, ohne über das Symbol a im Wort hinauszuschauen. Obwohl $(a+b)^*a$ eindeutig ist, ist es somit nicht 1-eindeutig. Dagegen ist $b^*(a(b^*a))^*$ ein 1-eindeutiger Ausdruck, welcher die gleiche Sprache wie $(a+b)^*a$ beschreibt.

In [BKW98a] wird gezeigt, dass nicht jeder eindeutige reguläre Ausdruck in einen 1-eindeutigen überführt werden kann. Dennoch werden wir im Folgenden die Eindeutigkeit im Sinne von [BEGO71] verwenden, um den Bezug zu den regulären Ausdrücken nicht zu verlieren.

Eine XML-Instanz besteht aus verschachtelten Objekten (den Elementen). Eine XML-DTD beschreibt die Struktur dieser Objekte, indem sie für jeden Objekttyp (den Elementnamen) die mögliche Folge der Typen seiner Unterobjekte vorgibt. Eine DTD wird als *xtree scheme* (*xscheme*) modelliert. Ein *xscheme* fasst Regeln zu Regelmengen zusammen.

Definition 3.7 (xscheme) *Ein xscheme d ist ein 3-Tupel $\langle \Sigma, s, \mathcal{R}\rangle$, wobei*

- Σ *ein endliches Alphabet ist.*
- $s \in \Sigma$ *das Startelement ist.*
- $\mathcal{R}$ *eine Menge von xrules über* Σ *der Form* $\langle e : type(e) \rangle$ *ist.*

Im weiteren Verlauf wird ein *xscheme* durch Auflistung seiner *xrule*s dargestellt. Das Startelement ist der Regelkopf der ersten *xrule*, und das Alphabet Σ ist die Menge aller vorkommenden Elementnamen. Der Bequemlichkeit halber führen wir, für den Fall, dass Elementnamen e in einem Regelrumpf aber nicht in einem Regelkopf vorkommen, *xrule*s der Form $\langle e : \epsilon \rangle$ nicht auf, nehmen aber immer deren Vorhandensein an.

bookstore	:	novel, journal
novel	:	exemplar*
journal	:	exemplar*
exemplar	:	title, author?

Abbildung 3.5: Beispiel für ein *xscheme*

Bisher beschreiben *xrule*s und *xscheme*s nur textuelle Sprachen. Der Bezug zu *xtree*s wird durch den Begriff der *„Gültigkeit"* hergestellt. Analog zu DTDs ist ein *xtree* gültig ($valid$) in einem *xscheme*, wenn er den einzelnen *xrule*s genügt.

Definition 3.8 (Gültigkeit eines xtree) *Ein xtree* t *ist genau dann in einem xscheme* $d = \langle \Sigma, s, \mathcal{R} \rangle$ *gültig (in Zeichen* $\models_t d$*), wenn*

- *label*$(root(t))$=s.
- *für jeden Knoten* n *in* t *mit Kindern* $n_1, \ldots, n_k$ *das Wort label*$(n_1) \ldots$*label*(n_k) *in* $L(type(label(n)))$ *ist, wobei* $\langle label(n) : type(label(n)) \rangle \in \mathcal{R}$ *gilt.*

Ein *xscheme* d beschreibt somit eine (evtl. unendliche) Menge von *xtree*s. $G(d)$ bezeichnet die Menge aller in d gültigen *xtree*s, so dass $t \in G(d) \Leftrightarrow \models_t d$. Mit dieser Definition ist z. B. der *xtree* in Abbildung 3.4 gültig im *xscheme* in Abbildung 3.5.

Bemerkung: Ein *xtree* ist im Gegensatz zu einer XML-Instanz durch seinen hierarchischen Aufbau immer wohlgeformt (*wellformed*).

Im Folgenden wird angenommen, dass ein *xscheme* $d = \langle \Sigma, s, \mathcal{R} \rangle$ *minimal* ist, d. h. anschaulich, dass es keine wirkungslosen *xrule*s gibt ($\neg \exists \langle e : type(e) \rangle \in \mathcal{R} : G(\langle \Sigma, s, \mathcal{R} \rangle) = G(\langle \Sigma, s, \mathcal{R} - \langle e : type(e) \rangle \rangle)$).

In Analogie zu den formalen Sprachen liegen *xscheme*s und kontextfreie Grammatiken bzw. *xtree*s und Ableitungsbäume auf der gleichen Sprachebene. Zwischen Grammatiken und *xscheme*s besteht dennoch ein wichtiger Unterschied. Wörter, die durch Grammatiken beschrieben werden, lassen sich als Ableitungsbäume darstellen, wobei ein Wort mehrere Ableitungsbäume haben kann. Eine XML-Instanz, welche ebenfalls ein Wort einer Grammatik ist, besitzt dagegen immer einen eindeutigen Ableitungsbaum (*xtree*).

Einem *xtree* kann somit in eineindeutiger Weise wieder ein Wort zugeordnet werden:

Definition 3.9 (Textdarstellung eines xtree) *Die textuelle Darstellung zu einem xtree t über* Σ *wird als xword bezeichnet. Der Übergang wird durch die Funktion* $text : G(d) \mapsto (\Sigma \cup \{<,>\})^*$ *beschrieben, mittels*

1. $text(t) = label(n) < text(n_1) \ldots text(n_k) >$, *wenn* $root(t) = n$ *und* n *hat die Kinder* $n_1, \ldots, n_k$.
2. $text(t) = label(n)$, *wenn* $root(t) = n$ *ein Blatt ist.*

Dabei sind $<,>$ spezielle Sonderzeichen mit $<,> \notin \Sigma$. Die Funktion $text()$ liefert eine normalisierte Präfix-Darstellung der XML-Instanz mit eindeutigen Trennzeichen. Die verwendeten Trennzeichen abstrahieren von der konkreten Tag-Syntax. Damit lassen sich auch Objekte anderer Beschreibungssprachen wie LaTeX, SGML usw. in ähnlicher Weise umsetzen. Voraussetzung ist ein explizites Tagging der Dokumente, so dass Formate wie *MS Word* und *Postscript* in ihrer nativen Form ungeeignet sind.

Beispiel 3.10 *Die Textdarstellung zum xtree t der XML-Instanz in Abbildung 3.4 ist in Abbildung 3.6 zu sehen.*

text(t)=bookstore<novel<exemplar< title author>>journal<exemplar <title>>>

Abbildung 3.6: Textdarstellung eines xtrees

Es lässt sich einfach erkennen, dass für ein *xscheme* d die Menge $L(d) = \{text(t) | t \in G(d)\}$ eine kontextfreie Sprache ist. $L(d)$ wird auch als die von d beschriebene textuelle Sprache bezeichnet, während $G(d)$ die Baumsprache ist. Da auch jedem *xword* wieder ein *xtree* zugeordnet werden kann, gibt es eine bijektive Beziehung zwischen den *xword*s eines *xscheme* d und deren *xtree*s. Die Umkehrfunktion zu $text()$ ist $parse : L(d) \mapsto G(d)$ und ordnet jedem *xword* einen *xtree* zu. Baumsprache und textuelle Sprache besitzen durch diese Isomorphie die gleiche Ausdruckskraft.

Unterschiedliche *xscheme*s können die gleichen Strukturen beschreiben. Daher wird zwischen identischen und ähnlichen *xscheme*s unterschieden.

Definition 3.11 (Äquivalenz/Ähnlichkeit) *Zwei xschemes* d_1 *und* d_2 *sind genau dann* <u>*äquivalent*</u>, *wenn* $G(d_1) = G(d_2)$. *Zwei xschemes* d_1 *und* d_2 *sind genau dann* <u>*ähnlich*</u>, *wenn es* *eine bijektive Abbildung* $h : \Sigma_1 \mapsto \Sigma_2$ *zwischen Elementnamen gibt, so dass für die kanonische Erweiterung auf xtrees* $h(G(d_1)) = G(d_2)$ *gilt.*

Obwohl die Äquivalenz zweier Grammatiken bzgl. der beschriebenen Sprache unentscheidbar ist, lässt sich einfach feststellen, ob zwei Grammatiken die gleichen Mengen von Ableitungsbäumen generieren. Dieses gilt trivialerweise genau dann, wenn die Grammatiken bis auf wirkungslose Regeln identisch sind. Die *Ähnlichkeit* entspricht gerade der starken Äquivalenz von Grammatiken [Bro95].

(i)	person	:	name, address*, phone, phone?, birthday
(ii)	person	:	name, (address, address) + address + address*, phone?,phone,birthday
(iii)	person	:	firstname, location*, call_number, call_number?, date_of_birth

Abbildung 3.7: Beispiel für äquivalente/ähnliche *xschemes*

Wie in Abbildung 3.7 leicht zu sehen ist, sind die aus jeweils einer *xrule* bestehenden *xschemes* (i) und (ii) äquivalent, obwohl ihre Regelrümpfe unterschiedlich sind. Ein Grund dafür ist, dass das zweite *xscheme* keinen eindeutigen regulären Ausdruck enthält. Die *xrules* (i) und (iii) sind ähnlich, da sie sich durch reine Umbenennungen aufeinander abbilden lassen. Aber auch (ii) und (iii) sind ähnlich, da die erzeugten $xtrees$ ebenfalls bis auf Umbenennungen identisch sind.

Wie bereits angesprochen können wir o. B. d. A. annehmen, dass die Regelrümpfe in einem *xscheme* eindeutig sind.

Der Begriff der „Korrektheit" drückt aus, dass sich alle *xtrees* einer Menge mit einem *xscheme* beschreiben lassen.

Definition 3.12 (Korrektheit) *Gegeben sei eine Menge* M *von xtrees. Ein xscheme* d *ist genau dann korrekt bzgl.* M*, wenn* $\forall t \in M : \ \models_t d$.

Ist eine Menge von *xtrees* korrekt bzgl. eines *xscheme*, so stellt sich die Frage, wie genau sie von diesem *xscheme* beschrieben wird. Hierzu schaffen wir einen Begriff der „Genauigkeit" mit unterschiedlichen Ausführungen:

Definition 3.13 (Genauigkeit) *Gegeben sei eine Menge* M *von xtrees. Ein korrektes xscheme* d *ist*

1. *genau bzgl.* M*, wenn* $G(d) = M$.
2. *ein genauestes, wenn es kein anderes korrektes xscheme* d' *gibt, mit* $G(d') \subset G(d)$.
3. *genauer als ein xscheme* d'*, wenn* $G(d) \subset G(d')$.

Intuitiv ist ein *xscheme* genauer als ein anderes, wenn die beschriebenen Mengen von *xtree*s in Teilmengenbeziehung stehen. Dieses ist ähnlich zum Enthaltensein bei kontextfreien Sprachen. Die Frage, ob ein *xscheme* genauer als ein anderes ist, ist entscheidbar. Sie beschränkt sich auf die Frage des Enthaltenseins der durch die korrespondierenden Regelrümpfe beschriebenen Sprachen. Die Frage des Enthaltenseins ist bei regulären Ausdrücken *PSPACE*-hart [GJ79, HU94]. Die Komplexität bzgl. *xscheme*s werden wir hier nicht weiter betrachten. Wie wir später zeigen werden, ist das genaueste *xscheme*, sofern es existiert, eindeutig bestimmt.

Dass die Begriffe *genau* und *am genauesten* nicht zwangsläufig deckungsgleich sind, zeigen die folgenden Beispiele:

Beispiel 3.14 *Wir schränken die durch das xscheme in Abbildung 3.5 beschriebene Menge von xtrees so ein, dass nur xtrees erlaubt sind, die für „novel“ jeweils „title“ und „author“ und für „journal“ lediglich „title“ beinhalten. Diese Menge ist nicht durch ein xscheme darstellbar. Das xscheme in Abbildung 3.5 ist zwar das genaueste, es ist aber nicht genau. Andererseits lässt sich aber auch kein genaueres xscheme finden, da sich die Einschränkung bzgl. der abhängigen Verwendung von „title“ und „author“ nicht mit einem xscheme ausdrücken lässt.*

Existiert ein genaues *xscheme*, so ist dieses trivialerweise das genaueste. Für den Fall, dass es kein genaues *xscheme* gibt, muss es aber auch kein genauestes *xscheme* zu einer Menge M geben.

Beispiel 3.15 *Eine Menge M soll aus einfachen xtrees mit einem „novel“ und Kindern bestehen, wobei alle Kinder mit „exemplar“ benannt sind. Die Anzahl der Kinder entspricht genau den Primzahlen. Für eine solche Menge lässt sich kein genaues und auch kein genauestes xscheme finden.*

(i)	novel	:	exemplar, exemplar + (exemplar, exemplar, exemplar, exemplar*)
(ii)	novel	:	exemplar, exemplar + (exemplar, exemplar, exemplar) + (exemplar, exemplar, exemplar, exemplar, exemplar, exemplar*)

Abbildung 3.8: *xscheme*s mit unterschiedlicher Genauigkeit

So ist (ii) in Abbildung 3.8 zwar genauer als (i), da es nur eine Teilmenge an xtrees zulässt, man kann aber jeweils noch ein genaueres xscheme finden.

Eine solche Einschränkung von *xscheme*s durch endliche Aufzählungen ist nur innerhalb von regulären Ausdrücken möglich. Es lassen sich keine Vorfahrsensitivitäten (s. u.) spezifizieren. Wollte man beispielsweise das *xscheme* in Abbildung 3.5 so einschränken, dass die Gesamtzahl der Exemplare eine Primzahl ist, so könnte man nicht jeweils ein genaueres *xscheme* finden.

Mit diesen Definitionen lässt sich nun ein wichtiges Problem behandeln. Hat man eine Menge von *xtree*s gegeben, so stellt sich die Frage, ob es ein gemeinsames *xscheme* gibt, ob dieses genau ist oder welches das genaueste *xscheme* ist.

Definition 3.16 *Eine Menge M von xtrees über einem endlichen Alphabet Σ_{all} kann die folgenden, voneinander unabhängigen Eigenschaften haben:*

1. *Ist die Wurzel aller xtrees mit dem gleichen Elementnamen bezeichnet, so heißt M konsistent.*

   $(\forall t, t' \in M : label(root(t)) = label(root(t')))$

2. *Für jedes $e \in \Sigma_{all}$ sei $L_{all}(e)$ die Sprache, die alle Wörter $label(n_1) \ldots label(n_k)$ enthält, für die es einen Knoten n mit $label(n) = e$ gibt und der die Kinder $n_1, \ldots, n_k$ hat. Ist $L_{all}(e)$ regulär ($\forall e \in \Sigma_{all}$), dann heißt M regulär.*

   $(\forall e \in \Sigma_{all} : \{e_1 \ldots e_n | \exists t \in M, n \in t : label(n) = e \wedge label(children(n)) = e_1 \ldots e_n\}$ *ist regulär)*

3. *Ist jeder Baum, der durch Ersetzung eines Teilbaums durch einen anderen Teilbaum entstanden ist, wobei die Teilbäume gleichbenannte Wurzeln haben, auch in M, so heißt M lokal.*

   $(\forall t, t' \in M, n \in nodes(t), n' \in nodes(t') : label(n) = label(n') \rightarrow t'[n/tree(n')] \in M)$

Ist eine Menge nicht lokal, so treten Elementnamen innerhalb von *xtree*s in Abhängigkeit von Vorfahren auf. Eine solche Abhängigkeit wird auch als „*Vorfahrsensitivität*" bezeichnet. Die Begriffe der Definition 3.16 sollen durch ein Beispiel veranschaulicht werden.

Beispiel 3.17 *Wir betrachten die drei Mengen von xtrees in Abbildung 3.9.*

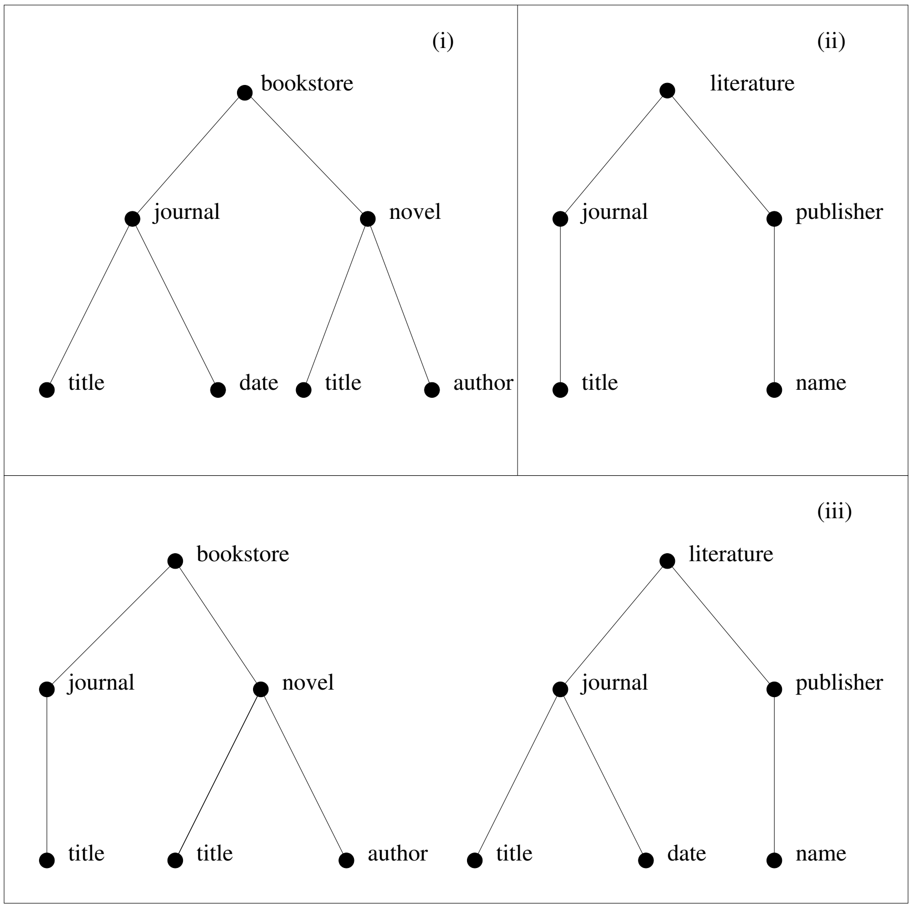


Abbildung 3.9: Ein Beispiel mit Mengen von xtrees

- (iii) *ist weder konsistent, da alle Wurzeln unterschiedlich bezeichnet sind, noch lokal, da die xtrees aus* (i) *und* (ii) *fehlen. Wir haben eine Vorfahrabhängigkeit, da z. B. ein einzelner Kinderknoten „title" lediglich unterhalb eines „journal" existiert, welcher seinerseits unterhalb eines Elements „bookstore" vorkommt.*

- (iii) *ist regulär, da für alle Knoten die Folge der Kinderbezeichner durch einen regulären Ausdruck beschreibbar ist. Im vorliegenden Fall ist dieses trivial, da die Anzahl der xtrees endlich ist.*

- (i) *und* (ii) *sind für sich selbst gesehen lokal.*

- *Die Vereinigung von* (i) *und* (iii) *is nicht lokal, da immer noch Vorfahrsensitivitäten auftreten.*

- *Die Vereinigung von* (i), (ii) *und* (iii) *ist lokal, da die Ersetzung von Teilbäumen nicht aus der Menge herausführt. Sie ist ebenso regulär aber nicht konsistent.*

Das folgende Beispiel zeigt, dass die Bedingungen der Regularität und der Lokalität einer Menge unabhängig voneinander sind.

Beispiel 3.18 *Für das xscheme in Abbildung 3.5 seien nur solche xtrees erlaubt, in denen die Anzahl der „exemplar"-Knoten unterhalb eines „novel"-Knoten prim sind. Diese Menge von xtrees ist offensichtlich nicht durch ein xscheme genau ausdrückbar. Sie ist nicht regulär, aber lokal. Dagegen ist eine Menge, die nur aus dem xtree in Abbildung 3.4 besteht nicht lokal, aber regulär.*

Satz 3.19 *Zu einer Menge M von xtrees gibt es gerade dann ein korrektes xscheme, wenn sie konsistent ist.*

Beweis. Da Σ_{all} endlich ist, kann es lediglich endlich viele Regeln geben. Daher ist das *xscheme* $d = \langle \Sigma_{all}, s, \mathcal{R} \rangle$, mit $\mathcal{R} = \{\langle e : (e_1 + \ldots + e_n)^* \rangle | e \in \Sigma_{all} = \{e_1, \ldots, e_n\}\}$ und $s = root(t)(\forall t \in M)$ immer korrekt. □

Im Folgenden benötigen wir eine Beziehung zwischen nicht-lokalen Mengen und *xscheme*s:

Satz 3.20 *Ist eine Menge von xtrees nicht lokal (d. h. sie hat Vorfahrsensitivitäten), dann ist sie nicht durch ein xscheme beschreibbar.*

Beweis. Eine Menge M mit Vorfahrsensitivitäten habe zwei Bäume s, t, so dass sich ein Baum t' (bzw. s') wie in Abbildung 3.10 erzeugen lässt, der nicht in M ist. Wären nun s und t durch ein *xscheme* beschreibbar, so hätte dieses *xscheme* eine Regel $\langle e : type(e) \rangle$, die sowohl den Teilbaum in s als auch den in t zulassen würde. Damit wären aber auch $t', s' \notin M$ durch das *xscheme* beschreibbar, also ist die reine Menge M nicht durch das *xscheme* beschreibbar.

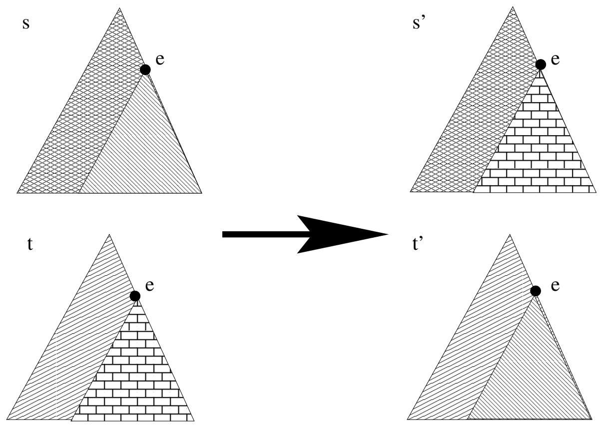


Abbildung 3.10: Lokale Implikation von Bäumen

□

Satz 3.21 *Zu einer Menge M von xtrees gibt es gerade dann ein genaues xscheme, wenn sie konsistent, regulär und lokal ist.*

Beweis. Das einzig mögliche genaue *xscheme* für eine konsistente, reguläre und lokale Menge M ist das *xscheme* d, für das $L(type(e)) = L_{all}(e)(\forall e \in \Sigma_{all})$ gilt, wobei das Startelement der Bezeichner aller Wurzeln ist. Konsistenz und Regularität sind trivialerweise gegeben.

$\Rightarrow$:
Da Vorfahrsensitivitäten gemäß dem Umkehrschluß zu Satz 3.20 nicht ausgedrückt werden können, ist M lokal.

$\Leftarrow$:
Annahme: d ist nicht genau. Dann gibt es einem Baum t mit (i) $t \in M \wedge t \notin G(d)$ oder (ii) $t \notin M \wedge t \in G(d)$.

(i) Da t die korrekte Wurzel hat und die Regeln $\langle e : L_{all}(e)\rangle$ erfüllt, wäre es in $G(d)$, Widerspruch!

(ii) Wäre $t \notin M$, so hätte es eine anders benannte Wurzel und würde eine der regulären Sprachen nicht-regulär machen oder zu einer Vorfahrsensitivität führen und wäre somit nicht in $G(d)$, Widerspruch! □

Wenn eine Menge M von *xtree*s kein genaues *xscheme* besitzt, ist es wichtig, ein *xscheme* zu finden, das diesem möglichst nah kommt.

Satz 3.22 *Zu einer Menge M von xtrees gibt es gerade dann ein genauestes xscheme, wenn sie konsistent und regulär ist.*

Beweis.

$\Rightarrow$:
Da ein *xscheme* für M existiert, ist gewährleistet, dass es nur ein Startelement gibt und die Sprachen $L_{all}(e)$ regulär sind, somit sind Konsistenz und Regularität gegeben.

$\Leftarrow$:
Da M konsistent und regulär ist, können wir das *xscheme* $L(type(e)) = L_{all}(e)(\forall e \in \Sigma_{all})$ mit entsprechendem Startelement erzeugen. Gebe es ein genaueres *xscheme*, so müsste dieses eine *xrule* $\langle e : type'(e)\rangle$ besitzen, mit $type'(e) \subset type(e)$ (da *xscheme*s ausschließlich über *xrule*s definiert werden). Dieses steht jedoch im Widerspruch zum erstgenannten *xscheme*.
□

Im folgenden Satz nehmen wir an, dass es einem Formalismus zur Beschreibung einer Menge M von *xtree*s gibt, der benötigte Entscheidbarkeitsaussagen zulässt.

Satz 3.23
(a) Die Frage, ob ein xscheme das genaueste für eine Menge M von xtrees ist, ist entscheidbar.

(b) Die Frage, ob ein xscheme genau für eine Menge M von xtrees ist, ist entscheidbar.

Beweis.
(a) Dieses folgt direkt aus dem Beweis zu Satz 3.22. Es muss lediglich untersucht werden, ob für alle Elementnamen e gilt, dass $L(type(e)) = L_{all}(e)$, da sich wegen der Regularität der $L_{all}(e)$ jeweils ein regulärer Ausdruck bestimmen lässt.

(b) Analog zu (a), unter Verwendung von Satz 3.21. □

Wir werden jetzt die Abschlusseigenschaften von *xscheme*s betrachten. Der Abschluss einer Sprachklasse bzgl. einer Operation gewährleistet, dass die Anwendung der Operation die Sprachklasse nicht verlässt. Im Folgenden werden wir untersuchen, ob *xscheme*s eine Boolesche Algebra bilden.

Satz 3.24 (Abschluss von xschemes) *xschemes sind unter Schnitt, nicht aber unter Vereinigung, Differenzbildung und Komplement (bzgl. Σ) abgeschlossen.*

Beweis.

- *Schnitt*:
  Wir betrachten die zugehörigen Mengen von Bäumen. Wenn beide Mengen identische Wurzelbezeichner haben, ist das auch für den Schnitt der Fall, ansonsten ist der Schnitt leer, also ebenfalls konsistent. Die regulären Vater-Kind-Beziehungen bleiben aufgrund der regulären Abgeschlossenheit bestehen.

  Zur Beibehaltung der Lokalität betrachten wir wiederum Abbildung 3.10. Seien s und t zwei Bäume mit einem mit e bezeichneten Knoten in M_1 (alle Fälle mit mehreren Bäumen lassen sich auf diesen Fall projizieren). Aufgrund der Lokalität müssen auch die durch Teilbaumersetzung entstandenen Bäume s', t' in der Menge vorhanden sein, also sei $M_1 = \{s, t, s', t'\}$.

Die mit dieser Menge zu schneidende Menge M_2 kann folgende Teilmengen besitzen:

M_2	$M_1 \cap M_2$	lokal?
s	s	ja
t	t	ja
s'	s'	ja
t'	t'	ja
s, s'	s, s'	ja
t, t'	t, t'	ja
s, t, s', t'	s, t, s', t'	ja

Alle anderen Kombinationen sind wegen der Lokalität nicht möglich, daher ist der Schnitt abgeschlossen.

- *Differenzbildung:*
  Betrachten wir die Differenz der *xscheme*s in Abbildung 3.11. Das *xscheme* (ii) beschreibt genau einen *xtree*. Ziehen wir diesen von den *xtree*s in (i) ab, so erhalten wir ein nicht-lokale Menge, denn dort existiert das Gegenstück zum abgezogenen *xtree* mit „title“ und „author“.

(i)	bookstore	:	novel + journal
	novel	:	exemplar*
	journal	:	exemplar*
	exemplar	:	title, author?
(ii)	bookstore	:	novel
	novel	:	exemplar
	exemplar	:	title

Abbildung 3.11: Differenzbildung von *xscheme*s

- *Vereinigung*:
  Betrachten wir die *xscheme*s und den *xtree* in Abbildung 3.12. Bildet man die Vereinigung der *xtree*s der beiden *xscheme*s (i) und (ii), so wäre (iii) das genaueste *xscheme*. Es lässt sich aber kein genaues *xscheme* finden, d. h. die Vereinigung ist nicht abgeschlossen. So ist der abgebildete *xtree* zwar in (iii) aber nicht in (i) oder (ii) gültig.

(i)	bookstore	:	novel
	novel	:	exemplar*
	exemplar	:	title, author
(ii)	bookstore	:	journal
	journal	:	exemplar*
	exemplar	:	title
(iii)	bookstore	:	novel + journal
	novel	:	exemplar*
	journal	:	exemplar*
	exemplar	:	(title, author) + title

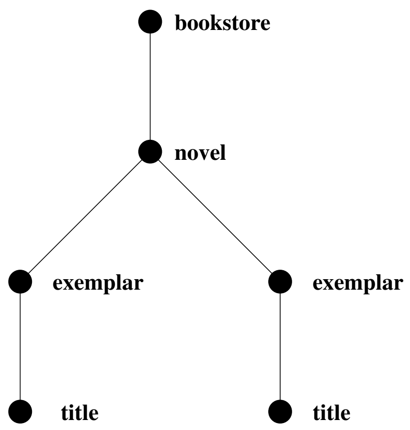


Abbildung 3.12: Inkonsistente Vereinigung zweier *xscheme*s

Komplement:
Die Komplementbildung ist i. a. nicht möglich, da ansonsten die Vereinigung durch Komplement und Schnitt ausdrückbar wäre. □

Eine direkte Konsequenz von Satz 3.24 ist die Tatsache, dass eine Menge keine unterschiedlichen genauesten *xscheme*s (d. h. *xscheme*s die unterschiedliche Sprachen beschreiben) besitzen kann. Eine Menge M hat also entweder ein genauestes *xscheme*, oder man kann für jedes korrekte *xscheme* ein genaueres finden (s. Bsp. 3.15). Diese Tatsache berechtigt, von *dem* genauesten *xscheme* zu sprechen.

Wie dieser Abschnitt gezeigt hat, besitzen *xscheme*s eine Reihe von Nachteilen. Im folgenden Abschnitt werden *xscheme*s dahingehend erweitert, dass bestimmte Abschlusseigenschaften erfüllt werden und dennoch wichtige Entscheidbarkeitsaussagen erhalten bleiben. Außerdem wird die Ausdruckskraft im Hinblick auf Anfragen und Restrukturierungen erweitert.

3.5 Behandlung von Daten unterschiedlicher Granularität

Wie oben gesagt können *xscheme*s (d. h. DTDs) als das Schema eines Dokuments betrachtet werden. Sie sind aber hinsichtlich der Struktur, die sie einem Dokument auferlegen, deutlich flexibler. Auf der einen Seite können sie Daten ähnlich stark strukturieren, wie das beispielsweise ein relationales Schema tut. Andererseits kann es möglich sein, dass jedes Element jedes andere Element enthält. Summa summarum definiert ein *xscheme* eine Struktur, die weniger restriktiv ist und mehr Variationsmöglichkeiten der Daten liefert, als Schemata herkömmlicher Datenmodelle [Suc98].

Diese Flexibilität kann bei der Beschreibung realer Daten ausgenutzt werden. Die Behandlung von Daten auf unterschiedlicher Ebene erlaubt eine genaue Berücksichtigung entsprechender Strukturinformationen.

3.5.1 Verfeinerung von regulären Ausdrücken

Für die Strukturdefinition in *xscheme*s werden *xrule*s verwendet. Mit diesen *xrule*s können auf der Basis regulärer Ausdrücke Strukturen für jeweils eine Ebene definiert werden. Für reguläre Ausdrücke sollen Verfeinerungen in dem Sinne definiert werden, dass ein regulärer Ausdruck eine Teilmenge eines anderen beschreibt.

(i)	person	:	name, firstname$^+$, phone?, address*
(ii)	person	:	name, firstname, firstname, address?

Abbildung 3.13: Beispiel für die Verfeinerung einer *xrule*

Die *xrule* (ii) in Abbildung 3.13 ist eine Verfeinerung von (i), da sie variable Kardinalitäten festlegt (vorname,telefon) bzw. einschränkt (adresse). Es ist leicht ersichtlich, dass jedes bzgl. des Regelrumpfs in (ii) korrekte Wort auch bzgl. des Regelrumpfs in (i) korrekt ist.

Zunächst ist jeder reguläre Ausdruck Verfeinerung von sich selbst. Die Tabelle in Abbildung 3.14 zeigt weitere Verfeinerungsregeln.

	allg. Ausdruck	verfeinerter Ausdruck
(i)	a^*	$a?$
(ii)		$a^* + a^*$
(iii)		ε
(iv)		$a, \ldots, a$, wobei die Anzahl der a's 0 sein kann.
(v)	$a?$	a
(vi)		ε
(vii)	$a + b$	a
(viii)		b

Abbildung 3.14: Verfeinerungsmöglichkeiten für reguläre Ausdrücke, wobei a, b RAs sind

Definition 3.25 (Verfeinerung von xrules) *Eine xrule $\langle e' : type(e')\rangle$ ist genau dann eine Verfeinerung einer xrule $\langle e : type(e)\rangle$, wenn $e' = e$ und $L(type(e')) \subseteq L(type(e))$.*

Damit ist ein *xscheme* bzgl. der *xrule*s feiner als ein anderes, wenn für Paare von korrespondierenden *xrule*s gilt, dass diese Verfeinerungen sind.

3.5.2 Verfeinerung der Struktur

Identische Informationen können auf verschiedenen Ebenen strukturiert werden. Es folgen zwei *xscheme*s, die identische Informationen unterschiedlich stark untergliedern:

Beispiel 3.26 *Ein Dokument ist in verschiedene Teile untergliedert, welche wiederum eine Struktur besitzen können. Dieses kann auf unterschiedlichen Stufen geschehen (s. Abbildung 3.15).*

(i)	document	:	intro, contents, chapter*, bib
(ii)	document	:	intro, contents, chapter*, bib
	chapter	:	section*
	section	:	paragraph*
	bib	:	references*

Abbildung 3.15: Beispiel für die strukturelle Verfeinerung eines xschemes

Das zweite *xscheme* ist in dem Sinne eine Verfeinerung des ersten, als dass es identische Informationen durch zusätzliche Elemente beschreibt.

Der Spezialisierungsbegriff wird somit für den zweiten Fall der strukturellen Verfeinerung erweitert:

Definition 3.27 (Verfeinerung der Struktur) *Ein xscheme d ist genau dann eine strukturelle Verfeinerung eines xschemes d', wenn $\mathcal{R}' \subset \mathcal{R}$ gilt. Dabei gilt insbesondere, dass d' minimal sein soll.*

3.5.3 Spezialisierung der Ausprägung

Mit einem *xscheme* lässt sich eine Menge von *xtree*s nur soweit spezifizieren, als dass für jeden Knoten ein regulärer Ausdruck die zulässigen Kinderknoten beschreibt. Es fehlt die Möglichkeit, abhängig von vorangegangenen Knoten, die weitere Struktur einzugrenzen. Gerade die Bedingung der Lokalität schränkt die Ausdruckskraft stark ein [PV99] . Es ist ein Mechanismus notwendig, der die regulären Ausdrücke mit einem „Gedächtnis“ ausstattet (ähnlich dem Keller eines PDAs), um so z. B. Vorfahrsensitivitäten behandeln zu können.

Beispiel 3.28 *Die Menge aus Bsp. 3.14 ist nicht durch ein xscheme beschreibbar. Auch ist eine Menge, die nur aus dem xtree in Abbildung 3.4 besteht, nicht durch ein xscheme beschreibbar. Beide Mengen sind nicht lokal nach Def. 3.16.*

Die beiden Mengen in Bsp. 3.14 sind also in dem Sinne Spezialisierungen ihrer *xscheme*s, als dass sie lediglich Teilmengen der von den *xscheme*s beschriebenen Mengen von *xtree*s zulassen. Es gilt somit die *genauer*-Beziehung aus Def. 3.13.

Beispiel 3.29 *Wir betrachten ein xscheme, das spezielle Mitarbeiter-XHTML[1]-Seiten beschreibt. Diese Seiten beginnen (vereinfacht) mit einem Bild, einer Liste mit (zwei) Forschungsinteressen und einer Liste mit (einer) Publikation, gefolgt von einer optionalen Adresse.*

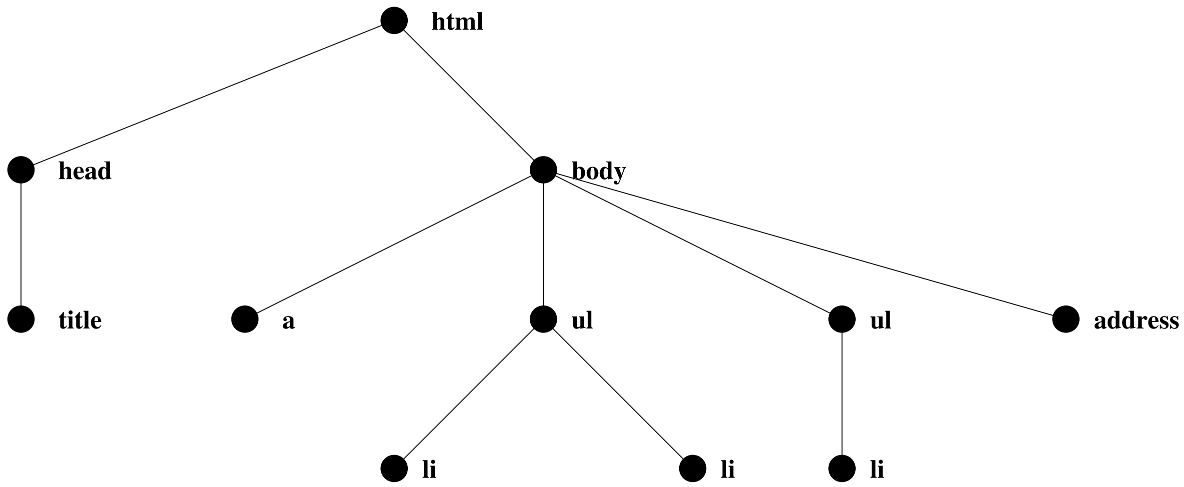


Abbildung 3.16: Ein xtree für eine Mitarbeiter-Seite

Während (i) *ein vereinfachtes XHTML-xscheme ist, soll* (ii) *ein auf obige Anforderung spezialisiertes XHTML-Mitarbeiter-xscheme beschreiben und* (iii) *ein eigenständiges Mitarbeiter-xscheme sein.*

(i)	html	:	head, body
	head	:	title
	body	:	(a + ul + address)*
	ul	:	li*
(ii)	html	:	head, body
	head	:	title
	body	:	a, ul, ul, address?
	ul	:	li + (li, li)
(iii)	staff_html	:	head, body
	head	:	title
	body	:	a, topic_ul, public_ul, address?
	topic_ul	:	li, li
	public_ul	:	li

Die allgemeine XHTML-Syntax (i) lässt sich mittels eines *xscheme* nicht soweit einschränken, dass sie nur Mitarbeiter-Seiten beschreibt. So sind z. B. Dokumente mit einem Forschungsinteresse und zwei Publikationen möglich. Andererseits ist eine Mitarbeiter-Seite nicht mehr XHTML-konform.

[1] XHTML ist die XML-konforme Definition von HTML [W3C00a]

Beispiel 3.30 *In XHTML können Listen wieder Listen enthalten. Es soll festgelegt werden, dass die Schachtelung dieser Listen nur bis zu einer bestimmten Tiefe gehen darf. Ähnlich wie oben kann diese Schachtelung nur über zusätzliche Elemente oder ohne Einschränkung der Rekursionstiefe ausgedrückt werden.*

Wir werden jetzt einen geeigneten Spezialisierungs-Mechanismus einführen, der die beschriebenen Probleme löst:

Definition 3.31 (s-xscheme) *Ein spezialisiertes xscheme (s-xscheme) d_S ist ein 5-Tupel $\langle \Sigma, \Sigma', S, \mathcal{R}, simplify \rangle$, wobei*

- Σ, Σ' *endliche Alphabete sind.*
- $S \subseteq \Sigma'$ *eine Menge von Startelementen ist.*
- $\forall s \in S : \langle \Sigma', s, \mathcal{R} \rangle$ *ein xscheme ist, welches mit d^s benannt wird.*
- $simplify : \Sigma' \mapsto \Sigma$ *eine Abbildung von Elementnamen auf Elementnamen ist.*

Die Abbildungsvorschrift $simplify()$ impliziert einen Homomorphismus über *xtree*s und damit auch über *xword*s. Der implizierte Homomorphismus wird ebenfalls mit $simplify()$ bezeichnet. Dieser Homomorphismus kann in kanonischer Weise auf Mengen erweitert werden.

Definition 3.32 (Gültigkeit bei Spezialisierungen) *Sei $d_S = \langle \Sigma, \Sigma', S, \mathcal{R}, simplify \rangle$ ein s-xscheme. Ein xtree t über Σ ist genau dann gültig in d_S (in Zeichen $\models_t d_S$), wenn es ein $s \in S$ gibt, so dass $t \in simplify(G(d^s))$.*

Hierbei bezeichnet d^s ein *xscheme* mit dem Startelement s und dem Alphabet Σ'. Anschaulich ist die Menge der in einem *s-xscheme* gültigen *xtree*s das homomorphe Abbild der Vereinigung der zugehörigen Mengen der *xschemes*. Ein *s-xscheme* wird also durch die Angabe der *xscheme*s und eines Homomorphismus über Elementnamen eindeutig beschrieben.

Beispiel 3.33 *Wir betrachten die xschemes (ii) und (iii) für Mitarbeiter-Seiten in Bsp. 3.29. Für das spezialisierte xscheme $d_S = \langle \Sigma, \Sigma', S, \mathcal{R}, simplify \rangle$ gelte*

Σ={*html, head, body, title, a, ul, li, address*}

Σ'={*staff_html, head, body, title, a, topic_ul, public_ul, li, address*}

$\mathcal{R} =$

staff_html	:	head, body
head	:	title
body	:	a, topic_ul, public_ul, address?
topic_ul	:	li li
public_ul	:	li

$$simplify(e) = \begin{cases} \text{html} & wenn\ e = \text{staff\_html} \\ \text{ul} & wenn\ e = \text{topic\_ul} \\ \text{ul} & wenn\ e = \text{public\_ul} \\ e & sonst \end{cases}$$

S={staff_html}

Beispiel 3.34 *Auch die endliche Rekursion der XHTML-Listenelemente lässt sich mithilfe eines s-xscheme beschreiben. Für das spezialisierte xscheme $d_S = \langle \Sigma, \Sigma', S, \mathcal{R}, simplify \rangle$ gelte*

$$\mathcal{R} = \boxed{\begin{array}{lcl} \text{html} & : & \text{head, body} \\ \text{head} & : & \text{title} \\ \text{body} & : & \text{a, ul , address?} \\ \text{ul} & : & \text{ul1 ?} \\ \text{ul1} & : & \text{ul2 ?} \\ \text{ul2} & : & \text{ul3 ?} \end{array}}$$

$$simplify(e) = \begin{cases} \text{ul} & wenn\ e = \text{ul1} \\ \text{ul} & wenn\ e = \text{ul2} \\ \text{ul} & wenn\ e = \text{ul3} \\ e & sonst \end{cases}$$

S ={html}

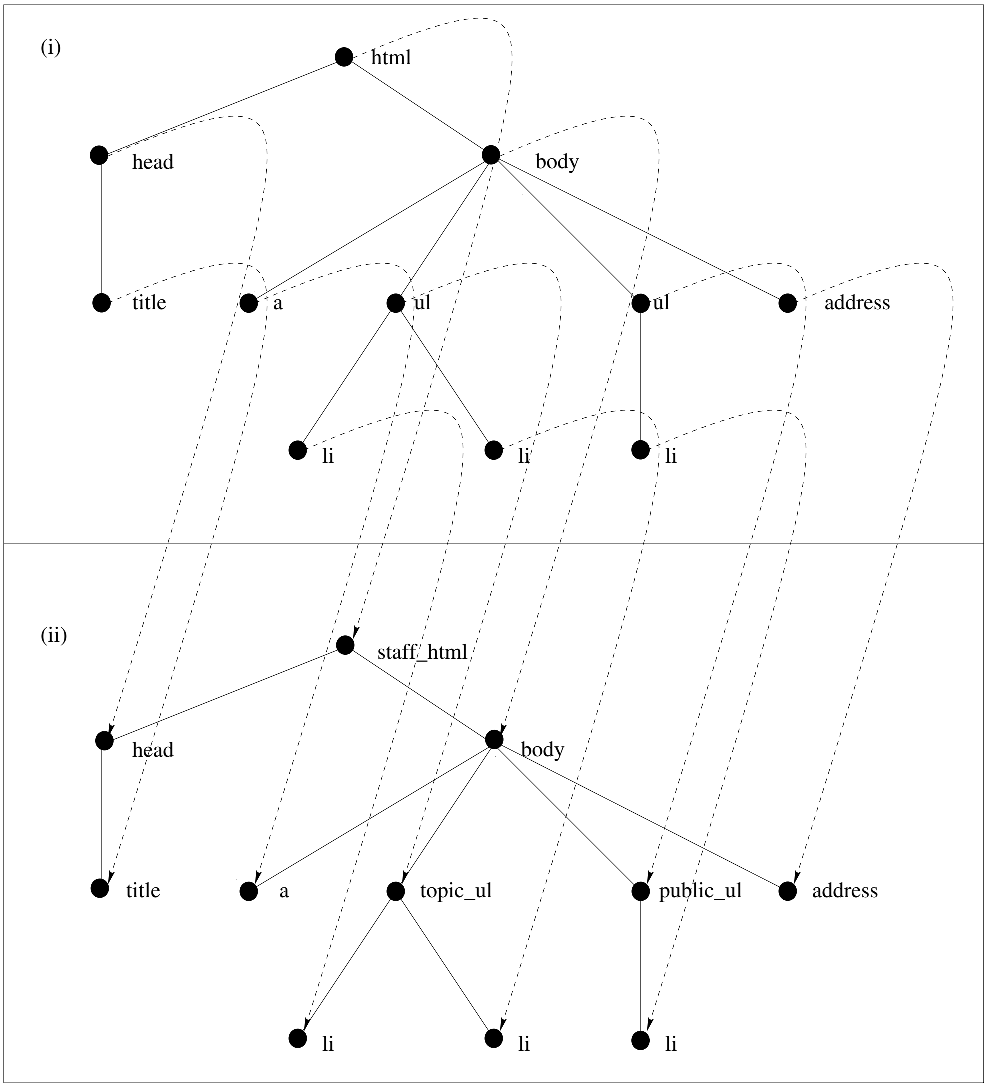
und den sich ergebenden Alphabeten Σ und Σ'.

Hierbei können Listen wiederum Listen enthalten. Dieses ist bis zu einer maximalen Tiefe von drei, unter Verwendung der gleichen Listenelemente möglich.

Alle weiteren Definitionen bzgl. *xscheme*s wie Korrektheit, Genauigkeit, Ähnlichkeit und Äquivalenz übertragen sich entsprechend auf *s-xscheme*s.

Ist ein *xtree* über Σ gültig in einem *s-xscheme* $d_S = \langle \Sigma, \Sigma', S, \mathcal{R}, simplify \rangle$, so gibt es mindestens einen zugehörigen *xtree* über Σ', der in einem *xscheme* $\langle \Sigma', s, \mathcal{R} \rangle$ mit $s \in S$ gültig ist.

Beispiel 3.35 *Zum obigen s-xscheme entsprechen sich die Knoten der xtrees über Σ (i) und Σ' (ii) wie folgt:*



Es kann zu einem *xtree* über Σ aber auch mehrere zugehörige *xtree*s über Σ′ geben, wie das folgende Beispiel zeigt:

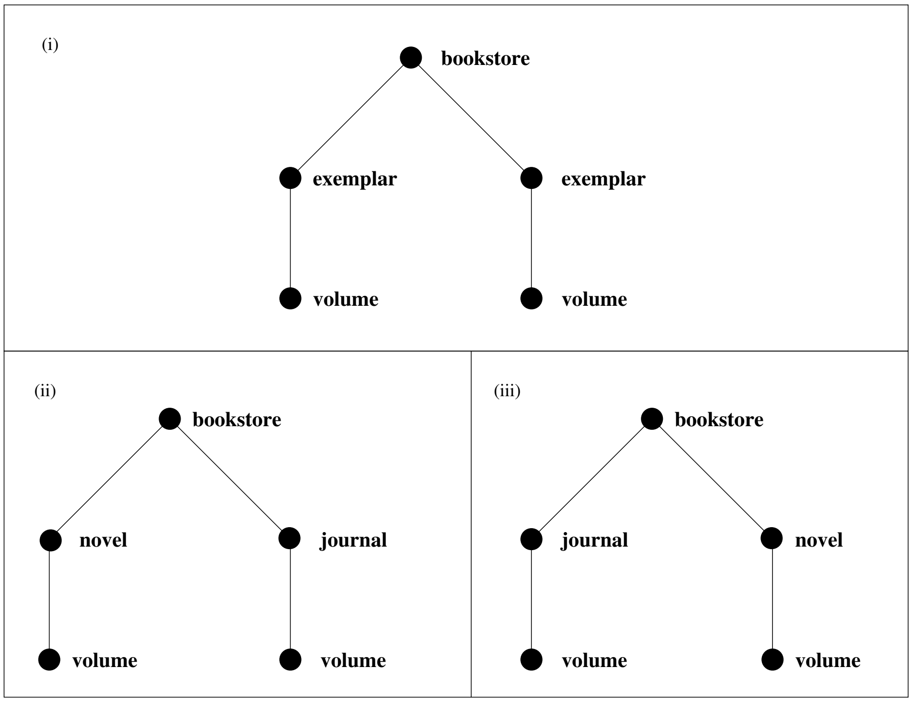
Beispiel 3.36 *Wir betrachten das s-xscheme* $d_S = \langle \Sigma, \Sigma', S, \mathcal{R}, simplify \rangle$ *mit*

$$\mathcal{R} = \boxed{\begin{array}{lcl} \text{bookstore} & : & (\text{novel} + \text{journal})^* \\ \text{novel} & : & \text{volume} \\ \text{journal} & : & \text{volume} \end{array}}$$

$$simplify(e) = \begin{cases} \text{exemplar} & wenn\ e = \text{novel} \\ \text{exemplar} & wenn\ e = \text{journal} \\ \quad e & sonst \end{cases}$$

und den sich daraus ergebenden Alphabeten Σ und Σ′.

Dann werden die beiden *xtree*s (ii) und (iii) auf den gleichen *xtree* (i) abgebildet:



Satz 3.37 (Wortproblem für s-xschemes) *Die Frage, ob ein xtree gültig in einem s-xscheme ist, ist entscheidbar.*

Beweis. Ein *xtree* t ist gerade dann in einem *s-xscheme* $d_S = \langle \Sigma, \Sigma', S, \mathcal{R}, \mathit{simplify} \rangle$ gültig, wenn es ein $s \in S$ gibt, so dass $\mathit{text}(t) \in \mathit{simplify}(L(d^s))$ gilt. Da $L(d^s)$ eine kontextfreie Sprache ist, ist durch die homomorphe Gestalt von $\mathit{simplify}()$ auch $\mathit{simplify}(L(d^s))$ kontextfrei. Ebenso ist die Vereinigung von kontextfreien Sprachen abgeschlossen. Somit kann man sich auf das Wortproblem für kontextfreie Sprachen beschränken. □

Die Spezialisierung der Ausprägung stellt sich als wichtigster Spezialisierungsmechanismus heraus, da sie die Ausdruckskraft bei Anfrage- und Integrationsoperationen stark erhöht. Das Vorhandensein von *s-xscheme*s erlaubt so die gezielte Erweiterung von Strukturinformationen.

3.6 Charakterisierung mit Automaten

Wie wir im Folgenden sehen werden, entspricht die Ausdruckskraft der *xscheme*s bzw. *s-xscheme*s genau den (*lokalen*) *regulären Baumsprachen* und *Baumautomaten* von [BKW98b]. Bei diesen regulären Baumsprachen handelt es sich um eine generalisierte Form der herkömmlichen Baumautomaten [GS96, CDG$^+$97]. Im Folgenden werden die wesentlichen Konzepte dieses Baumautomaten aufgegriffen und um Beispiele angereichert. Da die hier verwendeten Automaten teilweise eingeschränkte Varianten der Automaten in [BKW98b] sind, werden einige Beweise in vereinfachter Form wiederholt.

Die Hauptschwierigkeit bei der Entwicklung eines geeigneten Automatenmodells für *xtree*s besteht darin, dass die Anzahl der Kinderknoten im Gegensatz zu herkömmlichen Baumautomaten *unranked*, d. h. unbeschränkt ist. Daher müssen Überführungsfunktionen für eine unbeschränkte Anzahl von Kinderknoten definiert werden [Mur95]. Dieses geschieht, indem zunächst jedem Blattknoten ein Zustand zugewiesen wird. Ein Knoten erhält einen Zustand in Abhängigkeit davon, ob die Folge der Zustände seiner Kinderknoten in einer regulären Sprache enthalten ist. Auf diese Weise kann eine „unendliche" Überführungsfunktion in endlicher Form dargestellt werden.

Definition 3.38 (NATA, Def. 1.2 in [BKW98b]) *Ein nichtdeterministischer aufsteigender Baumautomat $\mathcal{M}$ (<u>NATA</u>) ist ein 4-Tupel $\langle Q, \Sigma, \delta, F \rangle$, wobei*

1. *Q eine endliche Menge von Zuständen ist.*
2. *Σ ein Alphabet von Elementnamen ist.*
3. *$\delta \subseteq \Sigma \times Q^* \times Q$ eine Relation (Zustandsüberführungsrelation) ist, so dass für jedes $q \in Q$ und $e \in \Sigma$ die Sprache $\{q_1 q_2 \ldots q_n | n \geq 0, \delta(e, q_1 q_2 \ldots q_n, q)\}$ regulär ist.*
4. *$F \subseteq Q$ eine Menge von akzeptierenden Zuständen ist.*

Bemerkung: Die Sprache $\{q_1 q_2 \ldots q_n | n \geq 0, \delta(e, q_1 q_2 \ldots q_n, q)\}$ werden wir im Folgenden durch $L(e, q)$ kennzeichnen.

Die Regularität der Überführungsfunktion gewährleistet, dass jeweils nur Worte einer regulären Sprache erkannt werden. Durch die Möglichkeit für einen Elementnamen mehrere Zustände zu definieren, ist es möglich, Vorfahrsensitivitäten auszudrücken.

Definition 3.39 (DATA, Def. 1.2 in [BKW98b]) *Ein deterministischer aufsteigender Baumautomat (<u>DATA</u>) ist ein NATA, wobei zusätzlich $\delta(e, q_1 q_2 \ldots q_n, q) \wedge \delta(e, q_1 q_2 \ldots q_n, q') \rightarrow q = q'$ (Determinismus der Zustandsüberführungen) gilt.*

Die deterministische Variante des Baumautomaten kann, im Gegensatz zur nichtdeterministischen, bei einem gegebenen Elementnamen und einer Folge von Zuständen, nur in einen einzigen Zustand wechseln. Wir sprechen daher auch von der „Zustandsüberführungsfunktion".

Definition 3.40 (LATA) *Ein lokaler aufsteigender Baumautomat (<u>LATA</u>) ist ein NATA, wobei zusätzlich $\delta(e, p_1 p_2 \ldots p_n, q) \wedge \delta(e, q_1 q_2 \ldots q_m, q') \rightarrow q = q'$ (Lokalität) und $|F| = 1$ (Konsistenz) gilt.*

Bei den lokalen Automaten gibt es für einen Elementnamen (unabhängig von den Zuständen seiner Kinder) jeweils nur einen Zustand. Damit ist ein lokaler Automat insbesondere auch deterministisch.

Berechnungen eines Automaten auf einem *xtree* werden induktiv über den Aufbau des *xtree*s spezifiziert:

Definition 3.41 (Berechnung eines Automaten) *Die Berechnung $M(t)$ eines Automaten $\mathcal{M}$ auf einen xtree t ist definiert als*

1. $M(t) = \{q|\varepsilon \in L(e,q)\}$, *wenn t aus einem Knoten besteht, der mit e bezeichnet ist.*
2. $M(t) = \{q|\exists q_1 \in M(t_1), \ldots, q_n \in M(t_n) \wedge q_1 \ldots q_n \in L(e,q)\}$, *wenn t die Unterbäume $t_1, \ldots, t_n$ hat.*

Die Zustände in $\{q|\varepsilon \in L(e,q)\}$ werden auch Startzustände genannt.

Ein *xtree* wird gerade dann von einem Automaten erkannt, wenn die Berechnung einen akzeptierenden Zustand liefert:

Definition 3.42 (Akzeptierter xtree) *Ein xtree t wird gerade dann von einem Automaten $\mathcal{M}$ akzeptiert, wenn $M(t) \cap F \neq \emptyset$.*

Die Menge aller von $\mathcal{M}$ akzeptierten *xtree*s wird mit $G(\mathcal{M})$ bezeichnet. Eine Menge M von *xtree*s ist genau dann erkennbar, wenn es einen Automaten $\mathcal{M}$ gibt, mit $G(\mathcal{M}) = M$.

Satz 3.43 *[Äquivalenz von DATA und NATA, Satz B in [BKW98b]] Ein Menge von xtrees wird genau dann von einem DATA erkannt, wenn sie von einem NATA erkannt wird.*

Beweis.
$\Rightarrow$:
Klar, da ein *DATA* auch ein *NATA* ist.

$\Leftarrow$:
Sei $\mathcal{M} = \langle Q, \Sigma, \delta, F\rangle$ ein NATA. Ein *DATA* $\mathcal{M}' = \langle Q_d, \Sigma, \delta_d, F_d\rangle$ sei wie folgt definiert: Die Menge der Zustände Q_d besteht aus allen Teilmengen der Zustandsmenge Q, d. h. $Q_d = 2^Q$. Dabei sind in F_d die Zustände aus Q_d, die einen akzeptierenden Zustand in $\mathcal{M}$ enthalten. Sei p ein Zustand in Q_d mit $p = \{q_1, \ldots, q_n\}$, wobei $q_1, \ldots, q_n \in Q$ gilt. Für die Zustandsüberführungen gilt dann $\delta_d = \{(e, p_1 \ldots p_n, p)|p = \{q \in Q|\exists q_1 \in p_1, \ldots, q_n \in p_n, (e, q_1 \ldots q_n, q) \in \delta\}\}$. Somit ist $L(e,p)$ mit $p \in Q_d$ regulär. □

I. a. ist es nicht erforderlich, die gesamte Potenzmenge der Zustände zu generieren. Es folgt ein Algorithmus, der nur erreichbare Zustände erzeugt [CDG+97]:

Algorithmus 1 *[Generierung eines DATA aus einem NATA]*

function *simplify_spec*(*NATA* $\mathcal{M} = \langle Q, \Sigma, \delta, F \rangle$) **returns** *DATA*
  / Ein Zustand q des äquivalenten DATA ist in 2^Q */*
  Sei $Q_d = \emptyset$ und $\delta_d = \emptyset$
  repeat
    $Q_d \leftarrow Q_d \cup \{q\}; \delta_d \leftarrow \delta_d \cup \{(e, q_1 \dots q_n, q)\}$
    where
      $e \in \Sigma, q_1, \dots, q_n \in Q_d,$
      $q = \{p \in Q | \exists p_1 \in q_1, \dots, p_n \in q_n : \delta(e, p_1 \dots p_n, p)\}$
  until *keine Regel mehr zu δ_d hinzugefügt werden kann*
  $F_d = \{q \in Q_d | s \cap F \neq \emptyset\}$
  return $\mathcal{M}_d = \langle Q_d, \Sigma, \delta_d, F_d \rangle$
end

Die Arbeitsweise der definierten Automaten soll in einem Beispiel veranschaulicht werden:

Beispiel 3.44 *Wir betrachten das xscheme in Abbildung 3.5 und den xtree in Abbildung 3.4. Der DATA $\mathcal{M} = \langle Q, \Sigma, \delta, F \rangle$ wird wie folgt definiert:*

δ enthält die Tupel

(title,ε,q_1), (author,ε,q_2), (exemplar,r_1,q_3), (novel,r_2,q_4), (journal,r_3,q_5), (bookstore,r_4,q_6),

wobei

r_1 in $L((q_1, q_2) + q_1)$, r_2, r_3 in $L(q_3{}^*)$ und r_4 in $L(q_4, q_5)$ sind.

Der akzeptierende Zustand lautet $F = \{q_6\}$, wobei sich die Mengen Q und Σ ergeben. Die Übergänge des Automaten sind in Abbildung 3.17 zu sehen.

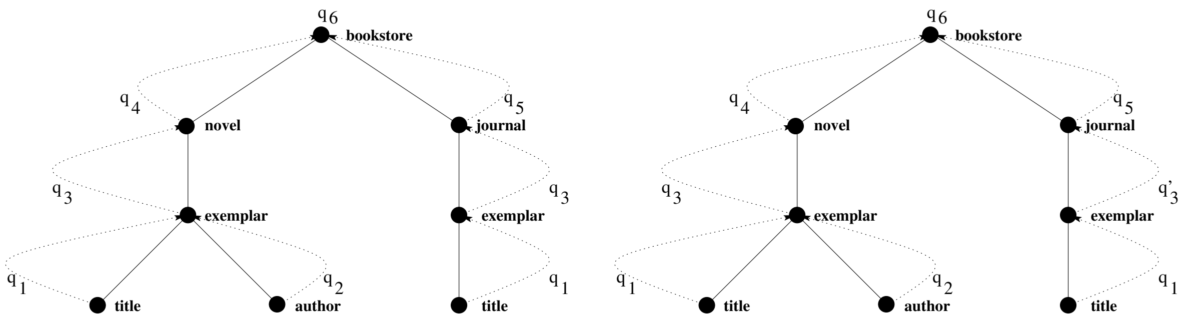


Abbildung 3.17: Übergänge eines Baumautomatens bei einem xscheme bzw. s-xscheme

Jetzt schränken wir die Mengen von xtrees so ein, dass ein „exemplar" unterhalb eines „novel" immer *einen „title“ und einen „author“ besitzt, während ein „exemplar“ unterhalb eines „journal“ analog zum Beispiel 3.14 lediglich einen „title“ hat.*

Mit diesen Beschränkungen kann ein „exemplar“ in Abhängigkeit seines Vorkommens im xtree verschiedene Zustände (q_3, q_3') *haben. Die Überführungsfunktion ändert sich wie folgt:*

δ *besteht aus den Tupeln*

(exemplar,r_1*,*q_3*),*

(exemplar,r_1'*,*q_3'*),*

(novel,r_2*,*q_4*),*

(journal,r_3*,*q_5*),*

und bookstore, title und author wie im vorhergehenden Fall. Für die regulären Ausdrücke gilt

r_1 *ist in* $L(q_1, q_2)$,

r_1' *ist in* $L(q_1)$,

r_2 *ist in* $L(q_3{}^*)$,

r_3 *ist in* $L(q_3'{}^*)$,

r_4 *ist in* $L(q_4, q_5)$.

Die Übergänge des auf dem xtree arbeitenden Automaten sind im zweiten Baum in Abbildung 3.17 zu sehen.

Im Folgenden benötigen wir eine Eigenschaft der definierten Baumautomaten. Wir können stets einen Automaten finden, so dass zwei Elementnamen nie den gleichen Zustand erhalten.

Lemma 3.45 *Für jeden DATA* $\mathcal{M} = \langle Q, \Sigma, \delta, F\rangle$ *existiert ein DATA* $\mathcal{M}' = \langle Q', \Sigma, \delta', F'\rangle$*, so dass*

$$\left.\begin{array}{l}(e, q_1 \ldots q_n, q) \in \delta',\\ (e', q_1' \ldots q_m', q') \in \delta',\\ (e \neq e')\end{array}\right\} \Rightarrow q \neq q'$$

Beweis. Wir definieren

$Q' = Q \times \Sigma$
$F' = \{(q, e) \in Q | q \in F\}$
$\delta' = \{(e, (q_1 e_1) \dots (q_n e_n), (q, e)) | e_1, \dots, e_n \in \Sigma, (e, q_1 \dots q_n, q) \in \delta\}$

Offensichtlich erfüllt $M' = \langle Q', \Sigma, \delta', F' \rangle$ die Forderung des Lemmas. □

Durch die Äquivalenz des deterministischen und nichtdeterministischen Baumautomaten können wir nun zeigen, dass diese die gleichen Baumsprachen beschreiben wie *s-xschemes*.

Satz 3.46 *[s-xscheme und DATA/NATA, Satz D in [BKW98b]] Für eine Menge M von xtrees gibt es genau dann ein genaues s-xscheme d_S, wenn es einen DATA bzw. NATA $\mathcal{M}$ gibt, der M erkennt.*

Beweis.
$\Rightarrow$:
Wir nehmen an, dass M von einem *s-xscheme* $d_S = \langle \Sigma, \Sigma', S, \mathcal{R}, \mathit{simplify} \rangle$ erzeugt wird. Sei $\mathcal{M} = \langle Q, \Sigma, \delta, F \rangle$ ein *NATA*. Die Relation $\delta : \Sigma \times Q^* \times Q$ ist definiert als $\delta(e, q_1 q_2 \dots q_n, q)$ für ein $\langle q : r \rangle \in \mathcal{R}, n \geq 0, q_1 q_2 \dots q_n \in L(r)$ mit $e = \mathit{simplify}(q)$. Dann gilt $G(\mathcal{M}) = G(d_S)$.

$\Leftarrow$:
Die Menge M werde von einem *DATA* $\mathcal{M} = \langle Q, \Sigma, \delta, F \rangle$ erkannt. Sei $d_S = \langle \Sigma, \Sigma', S, \mathcal{R}, \mathit{simplify} \rangle$ ein *s-xscheme*. Es gelte $\Sigma' = Q$ und $S = F$. Nach Lemma 3.45 können wir annehmen, dass keine zwei Elementnamen den gleichen Zustand erhalten. Daher definieren wir $\mathit{simplify}(q) = e \Leftrightarrow \exists q_1, q_2, \dots, q_n \in Q : (e, q1 \dots q_n, q) \in \delta$. Für jedes $q \in Q$ und $e \in \Sigma$ sei $r_{q,e}$ ein regulärer Ausdruck über Q, so dass $L(r_{q,e}) = L(e, q)$. Die Menge der *xrules* $\mathcal{R}$ ist definiert als $\bigcup_{q \in Q, e \in \Sigma} \{\langle q : r_{q,e} \rangle | \exists q_1, q_2, \dots, q_n \in Q : (e, q_1 q_2 \dots q_n, q) \in \delta\}$. Somit gilt $G(d_S) = G(\mathcal{M})$. □

Satz 3.47 *Eine Menge von xtrees ist regulär, wenn zu ihr ein genaues s-xscheme existiert.*

Beweis.

Wie wir im Satz 3.46 gesehen haben, besitzen *s-xschemes* und DATAs die gleiche Ausdrucksstärke. Die Aussage dieses Satzes folgt somit direkt aus Lemma 5.3 in [BKW98b]. □

Für die Umkehrung von Satz 3.47 muss eine Menge (neben der Regularität) einen endlichen Anfangsindex besitzen. Wir werden hierauf nicht näher eingehen; für Definition und Beweis sei ebenfalls auf [BKW98b] verwiesen.

Satz 3.48 (xscheme und LATA) *Für jedes xscheme d gibt es einen LATA $\mathcal{M}$, mit $G(d) = G(\mathcal{M})$.*

Beweis. Bei einem *LATA* gibt es eine eindeutige Zuordnung eines Elementnamen zu einem Zustand. Dadurch ließe sich die Überführungsfunktion auch in eine Menge von Tupeln $(e, q_1, \ldots, q_m)$ transformieren, welche genau den Sprachen der *xrule*s eines *xscheme* entsprechen. □

Satz 3.49 (Abschluss von s-xschemes, Satz H in [BKW98b]) *s-xschemes sind unter Schnitt, Vereinigung, Differenzbildung und Komplement (bzgl. Σ) abgeschlossen.*

Satz 3.50 (Leerheitsproblem für s-xschemes) *Die Frage, ob ein s-xscheme eine leere Sprache beschreibt, ist entscheidbar.*

Beweis. Betrachten wir einen *DATA*, so muss geprüft werden, ob die Überführungsfunktion einen transitiven Übergang von Startzuständen zu einem akzeptierenden Zustand zulässt. Dieses geschieht so, dass zunächst alle Zustände inkl. der Startzustände markiert werden, die sich von diesen mit einem beliebigen $e \in \Sigma$ erreichen lassen. Dieses wird solange wiederholt, bis entweder ein akzeptierender Zustand markiert ist, oder kein weiterer Zustand erreichbar ist. Da es nur endliche viele Zustände gibt, ist dieses immer möglich. □

Bemerkung: Da ein *xscheme* auch als *s-xscheme* aufgefasst werden kann, gilt dort die gleiche Aussage.

Satz 3.51 (Enthaltensein für s-xschemes) *Die Frage, ob ein s-xscheme genauer als ein anderes ist, ist entscheidbar.*

Beweis. Da *s-xscheme*s die Gesetze einer Booleschen Algebra erfüllen, ist für zwei *s-xscheme*s d_S und d'_S auch $G(d_S) - G(d'_S)$ durch ein *s-xscheme* beschreibbar. Die Frage des Enthaltenseins reduziert sich somit auf die Frage, ob $G(d_S) - G(d'_S) = \emptyset$. □

Durch die Entscheidbarkeit des Enthaltenseins, ist auch die Frage der Äquivalenz entscheidbar, da sich diese auf das gegenseitige Enthaltensein beschränkt.

3.7 Parsen von XML-Dokumenten

Werden *s-xscheme*s zur Definition von Schemata verwendet, so stellt sich schnell die Frage nach geeigneten Parsingmöglichkeiten [Beh00c]. Ein Parser sollte ähnlich einem herkömmlichen XML-Parser (z. B. [Javc]) arbeiten. Wir werden im Folgenden parsingrelevante Eigenschaften von *s-xscheme*s näher spezifizieren.

Zunächst werden wir eine konkrete Syntax für erweiterte DTDs benutzen, auf die im nächsten Abschnitt noch näher eingegangen wird.

```
<!ELEMENT bookstore    novel, journal>
<!ELEMENT novel        n_exemplar*>
<!ELEMENT journal      j_exemplar*>
<!ELEMENT n_exemplar   (title, author)>
<!ELEMENT j_exemplar   title>
<!ELEMENT title        (#PCDATA)>
<!ELEMENT author       (#PCDATA)>
<!MAPPING exemplar      n_exemplar, j_exemplar>
```

Abbildung 3.18: Beispiel für eine konkrete, erweiterte DTD

Wie wir in Abbildung 3.18 sehen können, bindet das Statement `MAPPING` verschiedene Ursprungsnamen an einen Elementnamen. Diese Ursprungsnamen werden *Ursprungskandidaten* genannt, welche sich vom nun folgenden Begriff des *Ursprungselements* unterscheidet:

Definition 3.52 (Ursprungselement) *Sei t ein xtree welcher in einem s-xscheme d_s gültig ist, t' ein anderer xtree, so dass $t' \in simplify^{-1}(t)$. Seien n, n' zwei korrespondierende Knoten in t bzw. t' und e, e' seien die zu n bzw. n' korrespondierenden Elemente. Das Element e' wird als Ursprungselement von e bzgl. n bezeichnet.*

Es ist wichtig zu bemerken, dass es in Abhängigkeit vom *s-xscheme* u. U. mehrere Ursprungselemente geben kann. Im Gegensatz zum Ursprungselement ist *der Ursprungskandidat* eines Elements e jedes Element e' für das $simplify(e') = e$ gilt. Daraus folgt, dass ein Ursprungselement ebenfalls ein Ursprungskandidat ist. Dieses gilt jedoch nicht umgekehrt.

Das Parsen eines *xtree* bzgl. eines *s-xscheme* kann auf folgende (ineffiziente) Weise geschehen: Für einen *xtree* erzeugen wir alle Ursprungsbäume, d. h. die Urbilder mit den Ursprungselementen in Bezug auf die homomorphe Abbildung. Jetzt können wir jeden dieser *xtree*s mit einem herkömmlichen Parser testen, was uns zum korrekten Ursprungsbaum führt. Wie bereits angedeutet, würde dieses Vorgehen jedoch zu exponentieller Zeitkomplexität führen und wäre daher nicht akzeptabel. Aus diesem Grund werden wir im Folgenden ein effizienteres Verfahren vorstellen.

Definition 3.53 (Sicheres *s-xscheme***)** *Ein s-xscheme d_s wird als genau dann als sicher bezeichnet, wenn jeder xtree t in $L(d_s)$ die Bedingung $|h^{-1}(t) \cap h^{-1}(L(d_s))| = 1$ erfüllt.*

Sicherheit gewährleistet, dass es immer genau ein Ursprungselement für einen Knoten eines *xtree* gibt.

Beispiel 3.54

```
<!ELEMENT bookstore    (novel + journal)*>
<!ELEMENT novel        volume>
<!ELEMENT journal      volume>
<!ELEMENT volume       (#PCDATA)>
<!MAPPING exemplar     novel, journal>
```

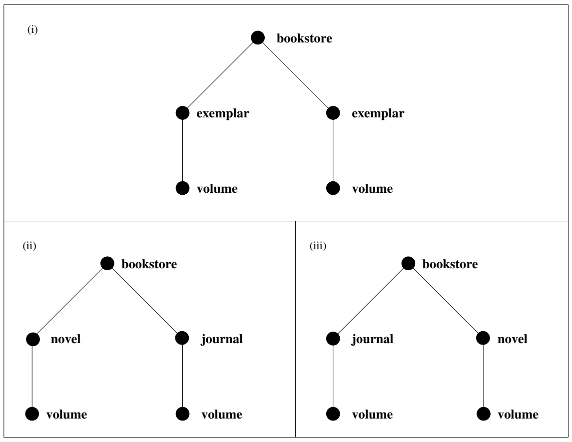


Abbildung 3.19: Unsichere spezialisierte DTD

Wir betrachten nun die unsichere spezialisierte DTD in Abbildung 3.19. Hier erzeugen wir einen xtree mit unterschiedlichen Ursprungsbäumen. Unter Benutzung der homomorphen Funktion werden die xtrees (ii) und (iii) auf denselben xtree (i) abgebildet.

Wir können zeigen, dass die Eigenschaft *sicher* die Ausdruckskraft von *s-xscheme*s nicht einschränkt.

Satz 3.55 (Existenz eines sicheren *s-xscheme***)** *Für jedes s-xscheme d_s gibt es ein sicheres s-xscheme $d_{s'}$, so dass $L(d_s) = L(d_{s'})$ gilt.*

Beweis. Wie im vorhergehenden Abschnitt gezeigt wurde, können wir für jede Sprache, die durch ein *s-xscheme* beschrieben wird, einen regulären Baumautomaten definieren, der diese Sprache akzeptiert. Jeder nichtdeterministische Baumautomat kann durch Erzeugung der Potenzmenge der Zustände in einen deterministischen Automaten umgeformt werden (bzw. der erreichbaren Zustände, s. Algorithmus 1). Dieses Vorgehen lässt sich entsprechend auf ein *s-xscheme* übertragen.

Ausgehend von einem unsicheren *s-xscheme* erzeugen wir ein neues *s-xscheme*, welches die Potenzmenge der Elementbezeichner enthält. Die Startelemente sind solche, die ein Startelement des unsicheren *s-xscheme*s enthalten. Die zugehörigen Regeln werden analog Satz 3.43 erzeugt.

□

Demzufolge setzen wir von nun an voraus, dass ein *s-xscheme* sicher ist, was die Eindeutigkeit des Ursprungsbaumes eines *xtree* innerhalb von $simplify^{-1}(L(d_s))$ impliziert.

Beispiel 3.56 *Die spezialisierte DTD von Beispiel 3.54 hat folgende sichere Variante:*

```
<!ELEMENT bookstore      novel_journal*>
<!ELEMENT novel_journal  volume>
<!ELEMENT volume         (#PCDATA)>
<!MAPPING exemplar       novel_journal>
```

Jedes XML-Dokument kann bezüglich seiner Struktur in linearer Zeit und mit linearem Platz geparst werden. Dieses ist durch die Tatsache begründet, dass jede Kindersequenz, welche zu einer 1-eindeutigen regulären Sprache gehört, in linearer Zeit und mit linearem Platz geparst werden kann. Betrachten wir einen *xtree* so müssen wir für jeden Knoten testen, ob seine Kindersequenz in der korrespondierenden *xrule* gültig ist. Dieses muss k-mal gemacht werden (in Abhängigkeit von der Anzahl der inneren Knoten), wobei für jeden Test t_i Kinderknoten herangezogen werden, so dass $\Sigma_{i=1}^{k} t_i = n$. Der Grund warum dieser Weg des Parsens für *s-xscheme*s nicht funktioniert, ist die mögliche Abhängigkeit nachfolgender Knoten von vorhergehenden. Wenn ein *xtree* top-down geparst wird, wissen wir im allgemeinen nicht, wie das Ursprungselement eines Knoten lautet (d. h. ohne mehrere Knoten nach vorne zu schauen).

Definition 3.57 (1-Eindeutigkeit eines *s-xscheme***)** *Ein s-xscheme heißt <u>1-eindeutig</u>, wenn jeder (vereinfachte) reguläre Ausdruck $simplify(r_e)$ einer xrule 1-eindeutig ist.*

Ein vereinfachter regulärer Ausdruck ist dabei der Ausdruck, der durch Umbenennung der Symbole entsprechend der homomorphen Abbildung entsteht.

Wenn wir die Eigenschaft der 1-Eindeutigkeit genau betrachten, stellen wir fest, dass sie die Ausdruckskraft von *s-xscheme*s einschränkt. In Fällen, in denen die Operatoren „$*, ?$“ verwendet werden, können die Ausdrücke nicht immer in eine 1-eindeutige Form gebracht werden.

Beispiel 3.58 *Betrachten wir folgende erweitere (Teil-)DTD*

```
<!ELEMENT body  (a,topic_ul?,public_ul,address?)>
<!MAPPING ul     topic_ul,pulic_ul>
```

Nach dem Parsen des Elements `a` *sind die Elementnamen* `topic_ul` *und* `public_ul` *mögliche Ersetzungen für das Element* `ul`*. Die korrekte Ersetzung kann erst zu einem späteren Zeitpunkt erkannt werden.*

Dennoch versetzt uns die Bedingung der 1-Eindeutigkeit in die Lage, Dokumente ohne jeglichen look-ahead gegen ein *s-xscheme* zu parsen. Dieses gibt uns die Möglichkeit, einen herkömmlichen XML Parser mit geringem Aufwand so zu erweitern, so dass er unsere XML-Erweiterungen verarbeiten kann.

Algorithmus[*s-xscheme*-Parsieren eines *xtree*]

function *parse*(1-eind. sicheres *s-xscheme* d_s, *xtree* t) **returns** Ursprungsbaum *xtree*
begin
  Beginne bei der Wurzel n von t
    Sei n mit e bezeichnet und habe Kinder bezeichnet mit der Sequenz $e_1 \ldots e_k$
    Suche eine *xrule* $\langle e : r \rangle$, so dass $e_1 \ldots e_k \in L(simplify(r))$
    if keine *xrule* existiert **quit**
    Benenne jeden Knoten mit seinem Ursprungselement
    for each Teilbaum t_i (welcher kein Blatt ist) der als Wurzel einen der Knoten hat
      do *parse*(d_s, t_i)
  return t
end

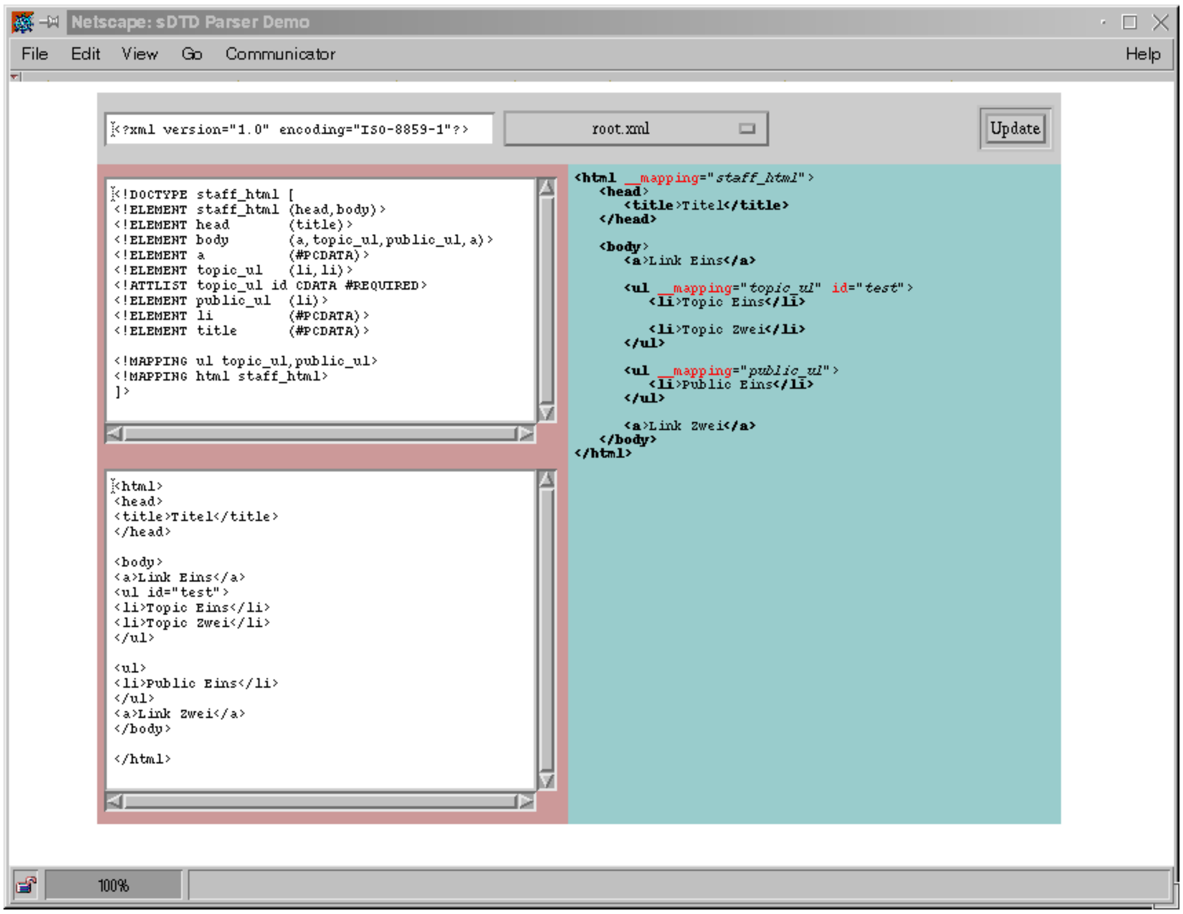


Abbildung 3.20: Screenshot des Webinterface zum Java-Parser für *s-xschemes*

In [Kre00] wurde ein Parser implementiert, der diese Untersuchungen verifiziert. Der Parser basiert auf Javasoft's *JAXP*-Parser [Javc] und ist komplett in Java implementiert. Abbildung 3.20 zeigt das Webinterface des Parsers.

3.8 Konkrete Umsetzung und Anwendung von *s-xscheme*s

Die theoretischen Grundlagen der vorangegangenen Abschnitte werden nun in eine konkrete Syntax überführt. Die homomorphe Abbildung wird durch ein `MAPPING`-Statement beschrieben, welches einen Elementnamen an seine möglichen Ursprungsnamen bindet (s. Abbildung 3.21). Dabei ist zu beachten, dass dieses Statement nicht Teil der standardisierten DTD-Definition ist, wie sie vom W3C vorgeschlagen wurde. Auf der linken Seite der Anweisung steht das homomorphe Abbild, während auf der rechten Seite die Urbilder abgebildet sind. Betrachten wir das *s-xscheme* von Beispiel 3.33 welches Mitarbeiter-XHTML beschreibt. Die konkrete Syntax hierfür ist in Abbildung 3.21 zu sehen:

```
<!ELEMENT staff_html   head, body>
<!ELEMENT head         title>
<!ELEMENT body         a, topic_ul, public_ul, address?>
<!ELEMENT topic_ul     li, li>
<!ELEMENT public_ul    li>
<!MAPPING ul           topic_ul, public_ul>
<!MAPPING html         staff_html>
```

Abbildung 3.21: Eine spezialisierte DTD für Mitarbeiter-XHTML-Seiten

Sobald ein zu diesem Schema konformes XML-Dokument geparst wurde, können wir Transformationen anwenden, die die Verfügbarkeit zusätzlicher, von der erweiterten DTD gelieferten Elementinformationen ausnutzen.

In Abbildung 3.22 ist ein XSLT-Programm [W3C98b, W3C99d] zu sehen, welches auf das Ursprungsbild des XML-Dokuments angewendet wird. Diese Transformation verarbeitet Dokumente, welche `staff_html`-Seiten entsprechen, und sammelt alle `topic_ul`-Elemente, die sie in eine neue Liste mit dem Tag `ul` einfügt. Dieses wäre nicht ohne ein *s-xscheme* möglich, da dieses zusätzliche Informationen liefert, die der Differenzierung eines herkömmlichen `ul`-Elements von einem `topic_ul`-Element dienen.

```
<xsl:stylesheet>
 <xsl:template match="staff_html/body">
  <OUTPUT>
    <H1> Research Topic List of :
         <xsl:apply-templates select="//address"/> </H1>
    <UL> <xsl:apply-templates select="//topic_ul/li"/> </UL>
  </OUTPUT>
 </xsl:template>
 <xsl:template match="address">
  <xsl:apply-templates/>
 </xsl:template>
 <xsl:template match="li">
  <LI> ``<xsl:apply-templates/>''</LI>
 </xsl:template>
</xsl:stylesheet>
```

Abbildung 3.22: Eine XSLT-Transformation von Mitarbeiter-XHTML-Seiten

Abbildung 3.23 zeigt das Ergebnis der Anwendung des obigen XSL-Programms auf das abgebildete XML-Dokument. Dabei werden die Informationen genutzt, die die Anwendung des *s-xscheme* liefert.

```
<html>
 <head> <title> R. Behrens
 </title> </head>
 <body>
  <a href="Picture.gif"></a>
  <ul>
    <li> XML </li>
    <li> Web </li>
  </ul>
  <ul>
    <li> Grammar Model for XML </li>
  </ul>
  <address> R. Behrens,
   IfIs </address>
 </body>
</html>
```

```
<OUTPUT>
    <H1> Research Topic List of :
         R. Behrens, IfIs  </H1>
    <UL>
     <LI>  ``XML'' </LI>
     <LI>  ``Web'' </LI>
    </UL>
</OUTPUT>
```

Abbildung 3.23: Eingabe und Ausgabe eines XSLT-Programms mit erweiterten Elementnamen

3.9 Vergleich zu anderen Ansätzen

DTDs dienten als erster Standard zur Beschreibung von XML-Schemainformationen. DTDs beschreiben ein Schema jedoch in einer sehr allgemeinen Form. Neben den in diesem Kapitel angesprochenen Nachteilen besitzen *DTD*s weitere Unzulänglichkeiten:

- DTDs definieren grundsätzlich eine Ordnung auf Elementen.
- Mit DTDs kann keine Ordnung auf Attributen festgelegt werden.
- DTDs lassen keine Typisierung von Attributen oder Referenzen zu.

Darüber hinaus zeigt [Sah00], dass DTDs in vielen Fällen als Dokumentations- anstatt als Strukturierungswerkzeug benutzt werden. So leiden DTDs unter ähnlichen Schwächen wie die ersten Programmiersprachen, da Modularität (Zusammensetzbarkeit einer DTD aus Teilen anderer DTDs) und Typsicherheit(Referenzen und Attribute können nicht getypt werden) nicht unterstützt werden. Diese (und andere) Nachteile führten zu weiteren Schemamechanismen, die hier kurz angesprochen werden.

Seit der Verabschiedung der ersten XML-Version gab es Diskussionen und Ansätze zur exakteren Schemabeschreibung. Der erste Vorschlag war *XML-Data* [W3C98e], welche die Spezifikation von Schemainformationen innerhalb der XML-Syntax, angereichert um Datentypen präsentiert.

Das *Schema for Object-Oriented XML (SOX)* [W3C99h] versucht, das objektorientierte Konzept näher mit XML zusammenzubringen. Ein anderer Vorschlag ist die *Document Definition Markup Language (DDML)*, welche in der ersten Version auch als *XSchema* bezeichnet wurde. Diese Sprache ist *XML-Data* sehr ähnlich.

Wir werden im Folgenden auf die aus der Sicht dieser Arbeit wichtigsten Ansätze *DCD* und *XML Schema* eingehen.

Document Content Description (DCD)

Die *Document Content Description (DCD)* [W3C98d] dient der Definition von Regeln für Struktur und Inhalt von XML-Dokumenten. Sie benutzt eine Teilmenge von *XML-Data*. Das besondere an DCD ist die Tatsache, dass das Resource Description Framework (RDF) zur Schemadefinition herangezogen wird. Genauer gesagt ist DCD ein spezielles RDF-Vokabular. Ein DCD-Schema ist dabei selbst ein wohlgeformtes XML-Dokument.

Die Grundideen von DCDs sind die folgenden:

- Die Semantik von DCD soll eine Obermenge der XML-DTDs sein.
- Das DCD-Datenmodell und die DCD-Syntax soll RDF-konform sein.
- Die mit DCD spezifizierten Strukturen sollen von „Authoring-Tools“ und anderen Applikationen nutzbar sein.
- DCD soll an allen geeigneten Stellen Mechanismen anderer W3C Arbeitsgruppen nutzen.

- DCDs sollen menschenlesbar und leicht verständlich sein.

Der DCD-Standard unterstützt die Grunddatentypen. Ein verkürztes Beispiel für eine, die obige DTD beschreibende DCD sieht wie folgt aus:

```
<DCD>
<ElementDef Type="bookstore" model="Elements" Content="Closed">
<Description> Ein einfacher Buchhaendler </Description>
 <Group RDF:Order="Alt">
  <Element>novel</Element>
  <Element>journal</Element>
  <Element>publisher</Element>
 </Group>
...
</DCD>
```

In diesem Beispiel wird ein Elementtyp `bookstore` definiert, mit der durch `Description` festgelegten Semantik. Als Inhalt sind für diesen Elementtypen die Elemente `novel`, `journal` und `publisher` alternativ (`Alt`) möglich.

Anzumerken bleibt, dass der DCD-Vorschlag über den Status einer „Note" nicht hinausgekommen ist.

XML Schema

Im Jahr 1999 schlug das W3C *XML Schema* vor. XML Schema ist ausdrucksstärker als DTDs. Während DTDs ihre eigene, Grammatik-ähnliche Syntax benutzen, werden XML Schemas selbst in der Form eines XML-Dokuments definiert. Damit ist es möglich, diese Schemabeschreibungen mit herkömmlichen XML-Werkzeugen zu verarbeiten. Darüber hinaus ist XML Schema selbst durch ein XML Schema definiert.

Die Spezifikation der XML Schemas wurde in zwei Teile untergliedert; dem Strukturteil [W3C99e] und der Beschreibung der Datentypen [W3C99f].

Im Strukturteil der XML Schema-Spezifikation [W3C99e] wird eine Menge von XML-Tags definiert. Diese Tags dienen der Einschränkung der Struktur, deren Bedeutung, Benutzung und der geltenden Beziehungen. Dennoch sind mit XML Schemas nicht alle denkbaren Strukturbeschränkungen ausdrückbar.

Durch Benutzung eines Dokumentationsvokabulars kann für ein Schema dessen Bedeutung, Benutzung und Funktion spezifiziert werden. Das Vokabular von XML Schema besteht aus ca. 30 Elementen und Attributen.

Im Datentypenteil der XML Schema-Spezifikation [W3C99f] wird die Typisierung von XML-Elementen und -Attributen behandelt. Dieses schien für Zwecke wie Dokumentverarbeitung und Datenaustausch unerlässlich. So wurden die wichtigsten Typen herkömmlicher

Programmiersprachen eingebracht, aus denen beliebige benutzerdefinierte Typen zusammengesetzt werden können.

Zunächst beschreibt ein Schema den möglichen Inhalt von Elementen und Attributen. Eine einfache Definition sieht dabei wie folgt aus:

```
<elementType name="bookstore">
  <mixed />
</elementType>
```

Hier wird der Elementtyp von `bookstore` definiert. Die Bezeichnung Elementtyp anstatt Element wird benutzt, um zwischen dem Typ eines Elements und dem Element selbst zu differenzieren. In diesem Fall ist als Inhalt der Typ `mixed` angegeben, was bedeutet, dass eine Mischung von Elementen und Strings (PCDATA) erlaubt ist.

```
<elementType name="exemplar">
  <sequence>
   <elementTypeRef name="title" minOccur="1" maxOccur="1"/>
   <elementTypeRef name="author" minOccur="0" maxOccur="1"/>
  </sequence>
</elementType>
```

Im zweiten Beispiel wird für den Elementtyp `exemplar` der Inhalt als Folge (`sequence`) von `title` und `author` definiert, wobei das Vorkommen des Autors optional ist (`minOccur=0`). Wichtig zu bemerken ist, dass die referenzierten Elementtypen ebenfalls definiert sein müssen.

Im Vergleich zu DTDs liefert XML Schema Basistypen, wie *Binary*, *Date* usw. bereits mit. Daneben unterscheidet sich *XML Schema* in mehreren Punkten von DCDs. Zunächst einmal wird bei der Nutzung von XML Schemas die Verwendung des zugehörigen Namespaces vorausgesetzt (die jedoch der Einfachheit halber in unseren Beispielen fehlen). Darüber hinaus gibt es bei XML Schema Vererbung und komplexe Typen, welche im DCD nicht vorkommen.

Es ist interessant, dass mittels des Vererbungsmechanismus von XML Schemas die Beschreibung lokaler Elemente, wie sie auch bei *s-xscheme*s vorkommen, unterstützt wird:

```
<elementType name="journal.exemplar">
  <archetypeRef name="exemplar"/>
</elementType>
```

Hiermit wird `exemplar` als *Archetype* des (bereits definierten) Elements `journal.exemplar` bestimmt. In der XML-Instanz wird der Elementname `exemplar` ohne weitere Verfeinerung benutzt.

Derzeit gibt es noch wenige Werkzeuge, die XML Schema unterstützen. Es ist jedoch wahrscheinlich, dass sich dieses in nächster Zeit ändern wird. Auch sind viele Effizienzfragen noch ungeklärt.

3.10 Fazit

DTDs waren der erste (da von SGML übernommen) und bis heute verbreitetste Ansatz zur Definition von Schemainformationen für XML-Daten. Die in diesem Kapitel diskutierten Unzulänglichkeiten führten zu weiteren Vorschlägen, die auf jeweils eigene Weise versuchen, einzelne Nachteile von DTDs zu kompensieren.

Unser Ansatz nimmt in dem Sinne eine Sonderstellung ein, als dass er die ursprüngliche DTD-Syntax weitgehend beibehält und sie um zusätzliche Komponenten erweitert. So kann gewährleistet werden, dass herkömmliche DTD-basierte XML-Parser mit weitgehend geringem Aufwand angepasst werden können. Dabei wird durch den vorgestellten Ansatz der Bezug zu formalsprachlichen Grundlagen, insbesondere die Notation in grammatikähnlicher Form, beibehalten.

Während Verarbeitungswerkzeuge für die erweiterten Schemaansätze nur prototypisch und vereinzelt vorliegen, ist ein geeigneter Parser für *s-xscheme*s bereits vollständig implementiert [Kre00].

4

Das MONTANA Datenmodell (MDM)

„Die Grenzen meiner Sprache bedeuten die Grenzen meiner Welt.“

LUDWIG WITTGENSTEIN

In diesem Kapitel wird das MONTANA-Datenmodell (MDM) vorgestellt, welches als zentrales Beschreibungswerkzeug innerhalb des Medienarchivs verwendet wird.

Generell besteht ein Datenmodell aus einer Menge von Abstraktionskonzepten, mit deren Hilfe Daten und Methoden eines realen Weltausschnitts auf einer abstrakten Ebene beschrieben werden können. Mithilfe der Abstraktionskonzepte werden Objekte und Beziehungen (meist graphisch) modelliert.

Insbesondere für die Anforderungen des Medienarchivs ist es wichtig, ein angemessenes Modell zu entwickeln, welches die einheitliche Beschreibung der multimedialen Eigenheiten und Zusammenhänge von Objekten erlaubt.

Der Entwicklung des MONTANA Datenmodells (MDM) lagen verschiedene Überlegungen und Erfahrungen aus vorangegangenen Prototypen und Arbeiten zugrunde [BL00, LL98]. Die Notwendigkeit eines Modells zur Klassifikation und Strukturierung von Objekten soll anhand eines universitären Beispielszenarios veranschaulicht werden:

1. Medizinisches Bildmaterial soll zu Lehrzwecken innerhalb des Medienarchivs abgelegt werden.

2. Zu diesem Bildmaterial sollen Thematik, Farbintensität und Patienteninformationen gespeichert werden.

3. Das Bildmaterial ist z. B. im `jpg`– oder `gif`–Format oder speziellen medizinischen Formaten vorhanden. Dem Material liegt dabei eine innere Struktur zugrunde.

4. Zwischen den Bildteilen herrschen Beziehungen, wie „ist-in“ oder „ist-Teil-von“.

5. Das Bildmaterial enthält eine große Anzahl von inhärenten Informationen, die mit entsprechenden Techniken automatisch extrahiert werden können.

6. Zum Bildmaterial sind dem Benutzer oder der Institution zusätzliche Informationen bekannt, die zwecks späterer Verarbeitung oder Recherche mit im Archiv abgelegt werden sollen.

7. Das Bildmaterial soll in unterschiedlichen Kontexten eingeordnet werden; beispielsweise soll eine Klassifikation auf Basis des in der Medizin häufig verwendeten MESH-Thesaurus [mesb] möglich sein[1].

Dieses Beispiel führt zu den nachstehend aufgeführten Folgerungen:

- Beim Importieren in das Archiv sollen Objekte einer Analyse unterzogen werden, welche den Objekten innewohnende Informationen ermittelt. Diese Informationen sollen vom Benutzer veränder- und erweiterbar sein. Die anfallenden Informationen sollen auf adäquate Weise dargestellt werden, so dass sie zum einen einfach manipuliert und zum anderen Anfrageoperationen zugänglich gemacht werden können.

- Objekte und Objektklassen besitzen interne Strukturen. Die diesen Objekten inhärente Struktur soll in das MDM abgebildet werden.

- Objekte können eine Vielzahl zusätzlicher Informationen besitzen, die gemeinsam verwaltet werden müssen. Diese Informationen können sowohl automatisch als auch manuell erstellt werden. Idealerweise soll die Beschreibung dieser Informationen von deren Herkunft unabhängig sein.

- Logische Beziehungen zwischen Objekten sollen genauso wie physikalische Verknüpfungen ausdrückbar sein. Logische und physikalische Verknüpfungen können sich sowohl auf ganze Objekte als auch auf Objektteile beziehen.

- Die im Archiv herrschenden Hierarchien, d. h. baumartigen Strukturen, angereichert um (evtl.) zyklische Links, müssen darstellbar sein. Die Darstellung soll dabei konsistent zur Beschreibung der globalen und strukturellen Zusammenhänge sein.

4.1 Datenmodell

Bei der Konzeption des Modells wurde ein weitgehend minimaler Ansatz gewählt. Das Modell ähnelt dem *Resource Description Framework (RDF)* des W3C [W3C99b]. Im Gegensatz zum RDF werden die schematischen Informationen mittels DTDs (bzw. der im letzten Kapitel angesprochenen Erweiterung) definiert.

Beschreibungen werden über einen *Subjekt-Prädikat-Objekt-Ansatz* realisiert. Das Subjekt stellt eine Ressource dar, welche über ein Prädikat mit einem Objekt in Beziehung steht.

[1]Man beachte, dass die Namensgleichheit des MESH-Projekts und des MESH-Thesaurus zufällig ist.

Das Objekt kann wiederum eine Ressource sein oder aber auch beliebig tief strukturiert sein. Durch diese Beschreibung erhält man schnell ein netzwerkartiges Geflecht der Objekte mit ihren Metainformationen und semantischen Beziehungen. In Abbildung 4.1 ist der grundsätzliche Zusammenhang zwischen Ressourcen und deren Beschreibung dargestellt.

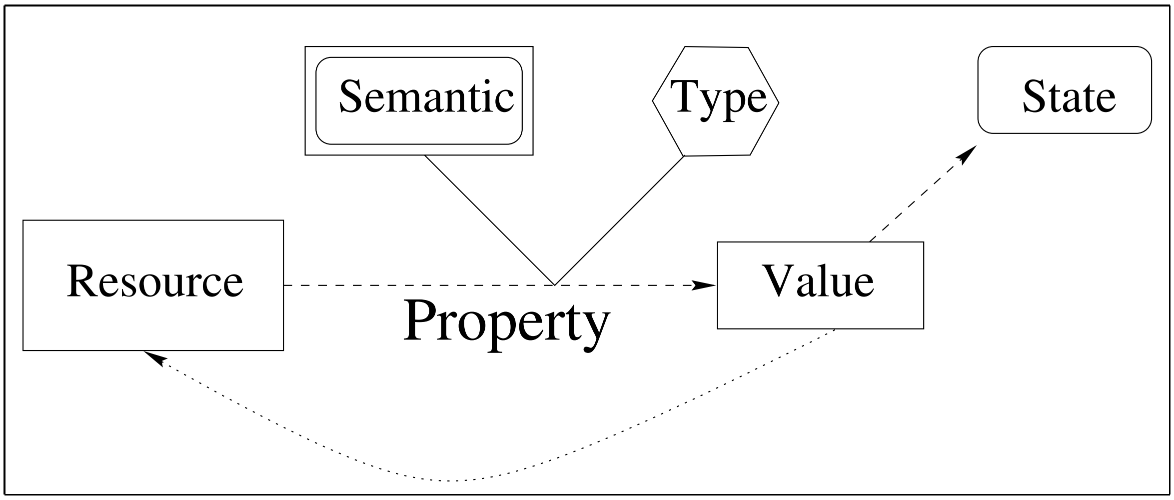
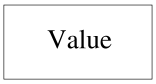


Abbildung 4.1: Beschreibung von Objekten im MDM

Wir werden hier die graphischen Elemente der Abbildung 4.1 näher erläutern:

Eine „Resource“ stellt ein Objekt oder auch ein Teilobjekt dar.

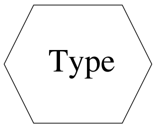


Ein „Value“ ist ein Wert, welcher selbst wieder eine Ressource sein kann. Ein Wert kann flach oder auch beliebig tief strukturiert sein.

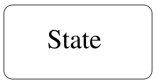
Property

Eine „Property“, d. h. eine Eigenschaft, stellt den Bezug zwischen einem Objekt und einem Wert her. Ein Objekt kann dabei beliebig viele Eigenschaften besitzen.

Die „Semantic“ spiegelt die Bedeutung einer Eigenschaft wider. So hat z. B. die Eigenschaft „Author“ in Abbildung 4.2 die Semantik, den Autor eine Ressource zu beschreiben.



Der „Type“ kennzeichnet den zulässigen Wertebereich einer Eigenschaft. Dabei können sowohl aus Programmiersprachen bekannte Typen (`String`, `Integer`, `Boolean`) als auch strukturierte Typen (z. B. XML-DTDs) zum Einsatz kommen.



Der „State“ bezeichnet den Status einer Eigenschaft. Die Bedeutung der verschiedenen Stati, die orthogonal zueinander sind, ist wie folgt definiert:

- **required:** Die Belegung dieser Eigenschaft ist zwingend, der Wert darf also insbesondere nicht leer sein.
- **implied:** Die Eigenschaft wird zunächst vom System vergeben. Diese kann beispielsweise eine automatisch berechnete Eigenschaft, wie etwa die Objektgröße sein.
- **fixed:** Die Eigenschaft ist unveränderlich. Z. B. kann der Besitzer einer Ressource so zwingend festgelegt werden.
- **system:** Die Eigenschaft wird vom System benutzt und ist für den Benutzer nicht sichtbar. Beispiele hierfür sind interne Identifikatoren.

Skalare Verbindungen, die bei deren Instanziierung durch genau eine Beziehung ersetzt werden.

Mehrwertige Verbindung, die bei deren Instanziierung durch beliebig viele Beziehungen ersetzt werden kann.

Implizite Sein-Beziehung, welche kennzeichnet, dass ein Wert wieder eine Ressource darstellt.

Je nach Verwendung werden in den folgenden Beispielen zum Teil einzelne Elemente weggelassen, um die Graphiken überschaubar zu halten.

Innerhalb des Modells können mit identischen Mitteln das Schema und auch die eigentliche Objektinformation definiert werden. Während Abbildung 4.2 die möglichen Eigenschaften eines Objekttyps festlegt, zeigt Abbildung 4.3 die konkrete Ausprägung einer Objektbeschreibung mit zwei Eigenschaften, wobei der Wert der zweiten Eigenschaft strukturiert ist.

Objekte (Verzeichnisse, Dateien, Verknüpfungen) haben verschiedene Eigenschaften und Eigenschaftsattribute. Eine Eigenschaft kennzeichnet ein bestimmtes Merkmal eines Objekts. Eine solche Eigenschaft besitzt ihrerseits weitere Annotierungen.

Als Beispiel für eine Beschreibung soll das Modell in Abbildung 4.2 dienen. Hier wird eine „GIF-Resource“ als Objekt mit zwei Eigenschaften („author“, „content“) definiert. Diese Eigenschaften haben jeweils einen Typ und eine Semantik. Der Wert des Autors muss für eine „GIF-Resource“ belegt sein („required“). Außerdem kann er nicht verändert werden („fixed“).

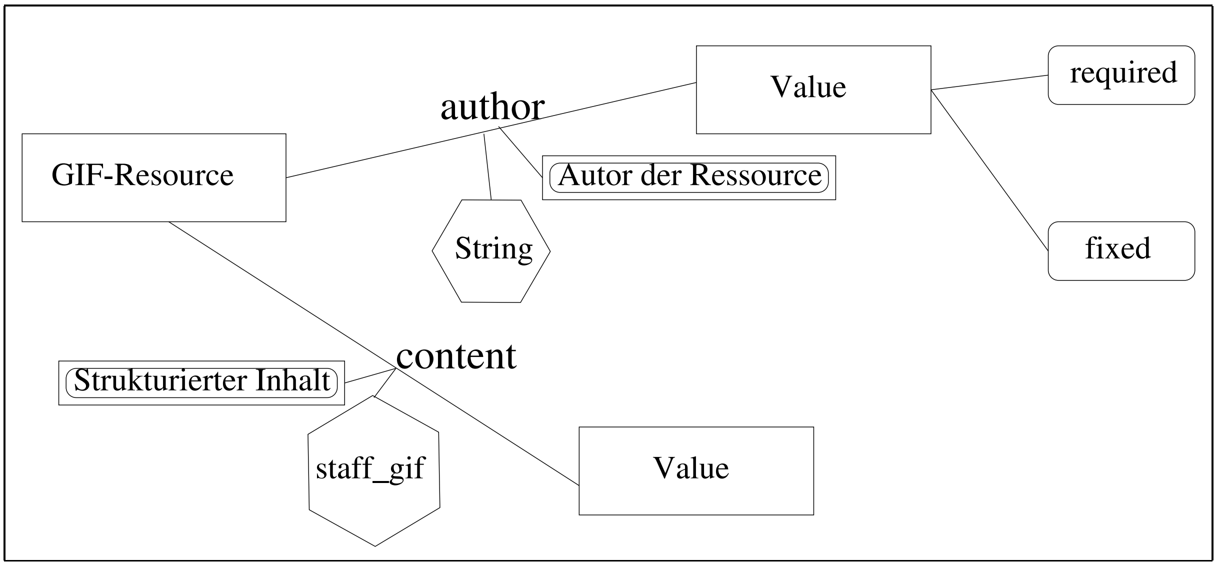


Abbildung 4.2: Schematische Darstellung einer GIF-Resource

Eine konkrete Instanz zum vorangegangenen Schema ist in Abbildung 4.3 dargestellt. Hier werden den Eigenschaften explizit Werte zugewiesen. Während die Eigenschaft „author" einen einfachen Wert mit dem Typ „String" besitzt, weist die Eigenschaft „content" eine interne Struktur auf (welche durch ein Dreieck dargestellt ist).

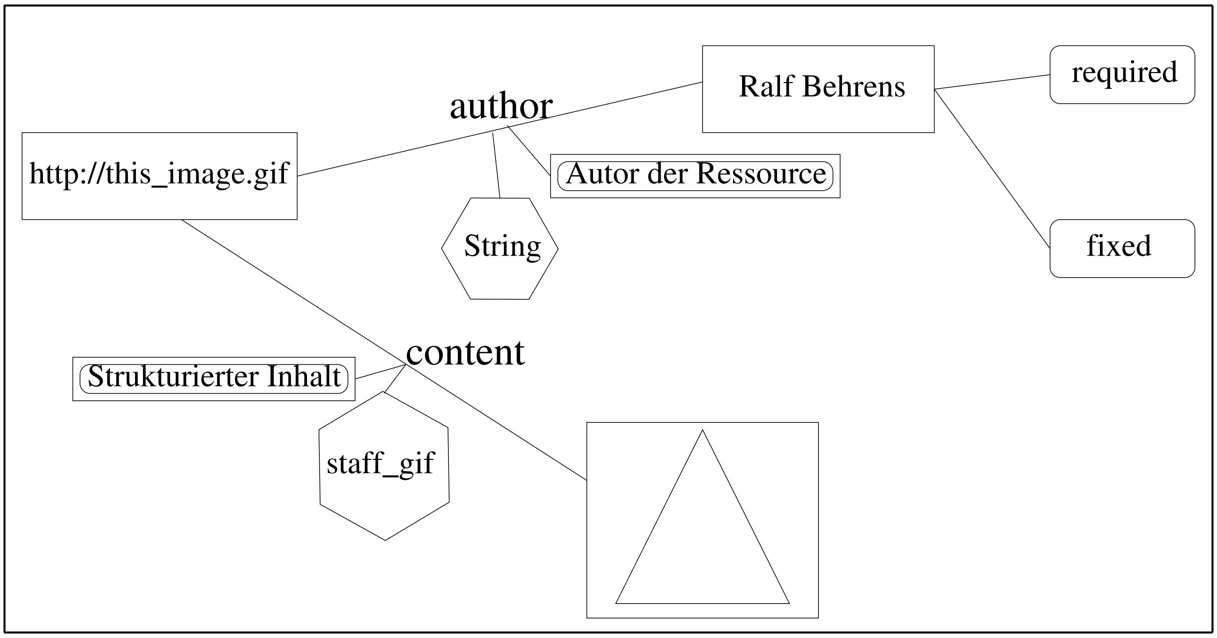


Abbildung 4.3: Konkrete Beschreibung eines Objekts

Die Klassifikation von Objekten geschieht über hierarchische Strukturen. Dabei werden neben Objektbeziehungen auch die tatsächlichen Objekte durch Strukturen beschrieben. Objekte können ihrerseits auf andere Objekte der Struktur oder auch in diese hinein verweisen. In Abbildung 4.4 ist die Beschreibung von Objektbeziehungen dargestellt. Ein innerhalb des MDM modelliertes Szenario ist aufgrund der Vielschichtigkeit der Beschreibungen als mehrdimensionales Gebilde interpretierbar. Logische Verknüpfungen im Modell werden über „Links" realisiert, die ebenfalls Objekte sind.

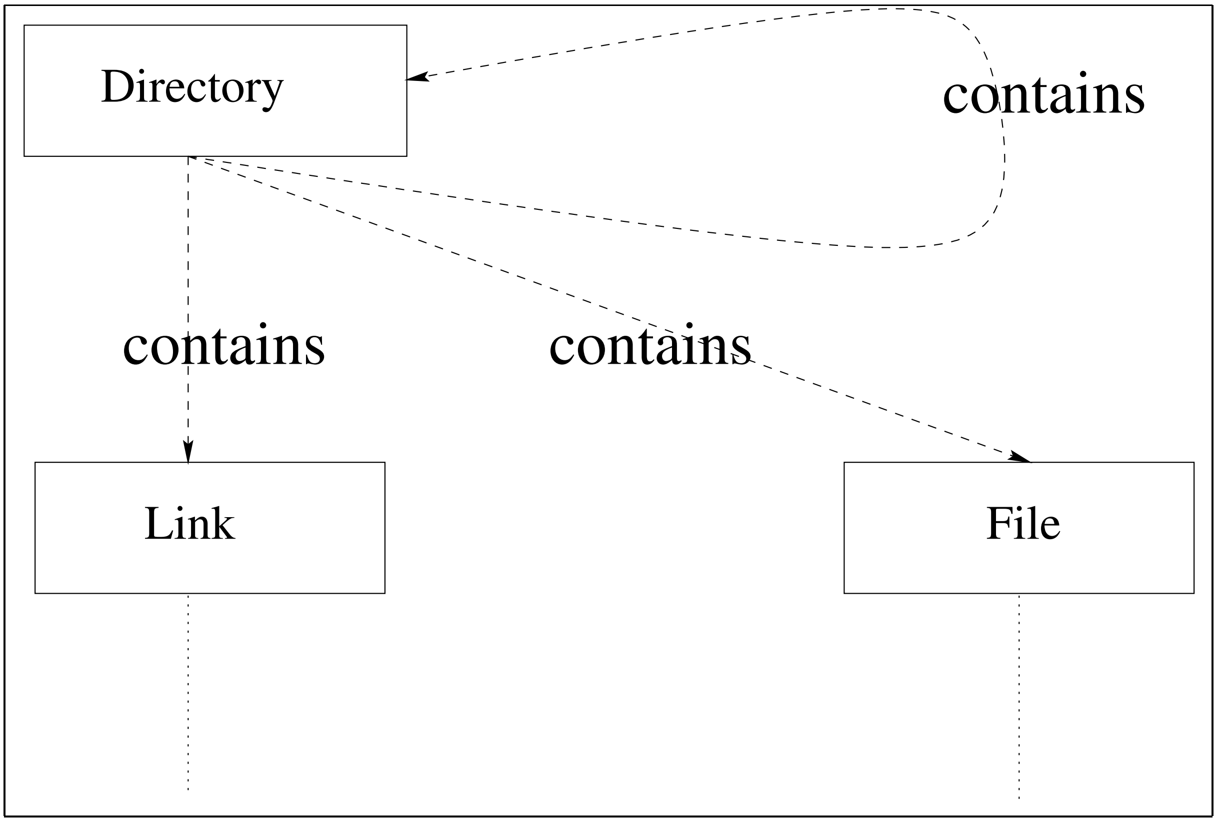


Abbildung 4.4: Schematische Beschreibung einer hierarchischen Klassifikation

4.2 Syntax des Modells

Die MDM-Syntax wird zur Speicherung von Instanzen des MDM in maschinenlesbarer Form benutzt. XML dient dabei als syntaktische Grundlage. Das MDM legt XML Restriktionen zur konsistenten Darstellung von Strukturen und Semantik auf. Da sich herkömmliche DTDs (wie im vorangegangenen Kapitel gezeigt) für die schematische Beschreibung als unzureichend erweisen, werden hierzu die in dieser Arbeit eingeführten *s-xscheme*s verwendet. Die mehrdimensionale Beschreibung mittels des MDM wird in eine flache XML-Struktur abgebildet. DTDs werden auf modulare Weise zusammengesetzt, um je nach Medientyp spezielle Strukturinformationen verfügbar und auch erweiterbar zu machen.

Anzumerken ist hierbei, dass die vorgestellte Umsetzung nach XML nur eine mögliche Variante darstellt. Je nach Einsatzgebiet und unter Berücksichtigung von Perfomancegesichtspunkten sind auch andere Alternativen denkbar bzw. können diese geeigneter sein.

In Abbildung 4.5 ist die XML-konforme Definition der auftretenden hierarchischen Strukturen (vgl. Abbildung 4.4) mittels einer DTD vereinfacht dargestellt. Objekte im Archiv können Verzeichnisse (`DIRECTORY`), Dateien (`FILE`) oder Verknüpfungen (`LINK`) sein. Der Objektbezeichner wird über das Attribut `name` festgelegt und ist sinnvollerweise innerhalb der Hierarchiestufe eindeutig.

Verknüpfungen werden über eine bidirektionale Bindung mit einem anderen Objekt assoziiert. So verweist sowohl eine Verknüpfung auf ihr Ziel (mittels `ref`) als auch das zugehörige Objekt auf seine Verknüpfungen (mittels `ref` innerhalb von `LINKSOURCE`). Durch diese bidirektionale Bindung wird das Auftreten von „toten Links“, d. h. von Verknüpfungen ohne zugehörige Objekte, verhindert.

```
<!ELEMENT REPOSITORY WORKSPACE*>

<!ELEMENT WORKSPACE  %objects;

<!ENTITY  % objects
          "(DIRECTORY | FILE | LINK)*" >

<!ELEMENT DIRECTORY  DESCRIPTION, LINKSOURCE*,
                     %objects;>
<!ATTLIST DIRECTORY  mode #PCDATA
                     owner #PCDATA
                     name #PCDATA>

<!ELEMENT FILE       DESCRIPTION, LINKSOURCE*>
<!ATTLIST FILE       mode #PCDATA
                     owner #PCDATA
                     name #PCDATA>

<!ELEMENT LINK       DESCRIPTION, LINKSOURCE*>
<!ATTLIST LINK       ref #PCDATA
                     name #PCDATA>

<!ELEMENT LINKSOURCE EMPTY>
<!ATTLIST LINKSOURCE ref #PCDATA>
```

Abbildung 4.5: Strukturbeschreibung der MDM-Hierarchien

Anhand einer Beispielausprägung der Strukturdefinition in Abbildung 4.5 soll der Übergang von der graphischen Repräsentation zu tatsächlichen XML-Daten veranschaulicht werden.

Die Abbildung 4.6 stellt zwei Objekte dar, die sich unterhalb der Hierarchie „workspace" befinden. Während das erste Objekt („this_image.gif") eine konkrete Datei ist, stellt das zweite Objekt („link_image.gif") eine Verknüpfung mit dem ersten Objekt dar. Der Zusammenhang zwischen diesen beiden Objekten ist über die Eigenschaften „linktarget" und „linksource" spezifiziert.

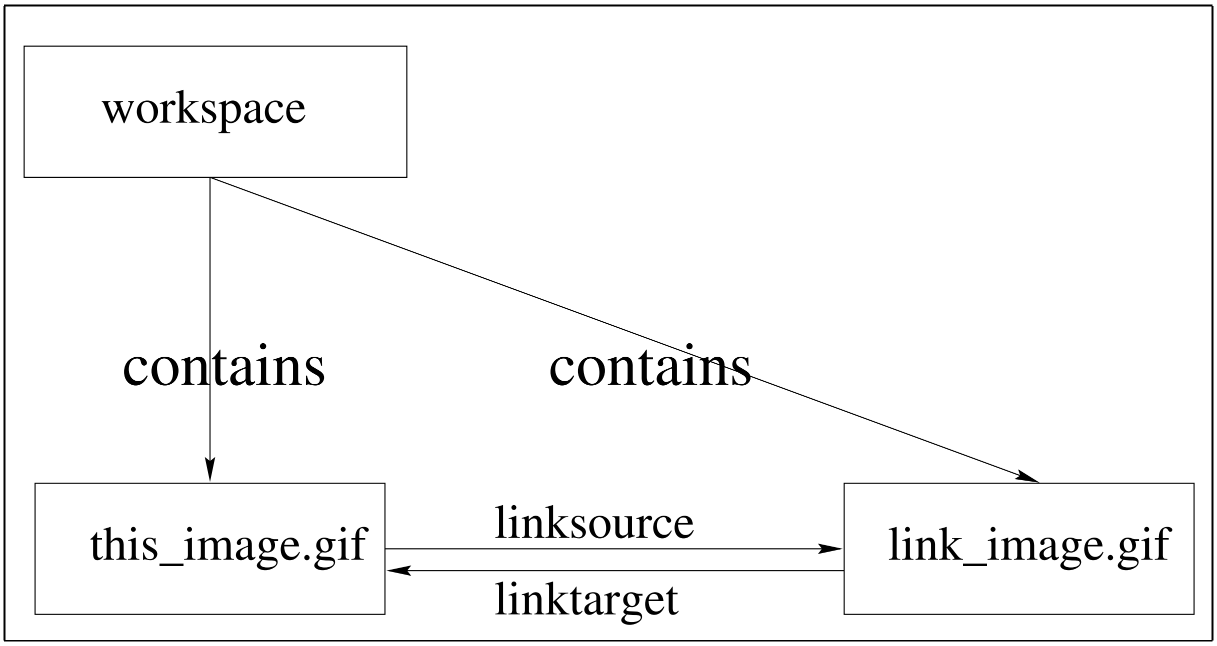


Abbildung 4.6: Konkrete Beschreibung einer hierarchischen Klassifikation

Die XML-Repräsentation der graphischen Darstellung ist in Abbildung 4.7 zu sehen. Sie ist gemäß der strukturbeschreibenden DTD (Abbildung 4.5) aufgebaut. Die Datei „this_image.gif" wird durch ein Tag `FILE` beschrieben, während die Verknüpfung „link_image.gif" durch das Tag `LINK` dargestellt wird. Die Beziehung zwischen den beiden Objekten entsteht über die `ref`-Attribute, welche Objektlokationen enthalten (s. u.). Das `ref`-Attribut des `LINK` kennzeichnet das „linktarget", also das der Verknüpfung zugehörige Objekt. Daneben kann ein Objekt mehrere `ref`-Attribute besitzen, die jeweils einem `LINKSOURCE` angehören. Jedes dieser `LINKSOURCE`s beschreibt dabei eine Verknüpfung, die ihrerseits auf dieses Objekt verweist.

```
<REPOSITORY>
 <WORKSPACE>
    <FILE mode="-rwr---" owner="behrens"
     name="this_image.gif">
     <DESCRIPTION> ... </DESCRIPTION>
     <LINKSOURCE
      ref="/workspace/link_image.gif"/>
    </FILE>
    <LINK name="link_image.gif"
     ref="/workspace/this_image.gif">
     <DESCRIPTION> ... </DESCRIPTION>
    </LINK>
 </WORKSPACE>
</REPOSITORY>
```

Abbildung 4.7: Beispielinstanz

Objektlokationen werden über *XPath*-ähnliche Ausdrücke [W3C99c], sogenannte *MDM-Pfadausdrücke*, formuliert. Ein Pfad wird durch eine über '/' getrennte Aneinanderreihung der Objektbezeichner definiert. So spezifiziert der Ausdruck `/workspace/this_image.gif` (s. Attribut `ref` des Link) das Objekt `this_image.gif`, welches sich direkt unterhalb der Wurzelhierarchie `workspace` (Abbildung 4.6) befindet.

Die Tags `repository` und `workspace` dienen der Trennung von Informationen in einzelne Benutzerbereiche. Wir werden im Rahmen des Modells hierauf nicht näher eingehen.

Wie oben beschrieben, existiert zu jedem Objekt eine Beschreibung. Eine solche Beschreibung (`DESCRIPTION`) stellt eine Sammlung von Eigenschaften (`PROPERTY`s) dar. Weiterhin besitzt eine Beschreibung ein Attribut `about`, welches den MDM-Pfadausdruck dieses Objekts für die Lokation im MDM symbolisiert.

```
<!ELEMENT DESCRIPTION PROPERTY*>
<!ATTLIST DESCRIPTION about #PCDATA>
<!ENTITY  % sdtd_type (all_formats)>
<!ELEMENT PROPERTY    (#PCDATA)|%sdtd_type;>
<!ATTLIST PROPERTY    name #PCDATA
                      required "yes|no"
                      implied "yes|no"
                      fixed "yes|no"
                      system "yes|no"
                      semantic #PCDATA
                      type #PCDATA>
```

Der folgende Ausschnitt zeigt die Beschreibung einer Eigenschaft (vgl. Abbildung 4.3) in der konkreten MDM-Syntax:

```
<PROPERTY name="author" required="yes" implied="no" fixed="yes" system="no"
          semantic="Autor der Ressource" type="String">
          Ralf Behrens
</PROPERTY>
```

Strukturwertige Eigenschaften (im Spezialfall Datentypen) werden über DTDs (bzw. einer Erweiterung) definiert, die durch das `ENTITY sdtd_type` gekennzeichnet sind. Innerhalb dieses `ENTITY` werden die verwendeten Typen eingetragen. Der Wert `all_formats` dient hier als Platzhalter für die möglichen Datentypen. Im Folgenden sehen wir die Strukturdefinition zweier unterschiedlicher Graphikformate:

```
<!ELEMENT staff_gif          color_intense, geometry >
<!ELEMENT geometry           height, width >


<!ELEMENT medical_gif        color_intense, geometry >
<!ELEMENT geometry           height, width, colordepth >
```

Beim modularen Zusammenbau von Dokumenttypen ergibt sich das Problem, dass die verwendeten DTDs disjunkte Elementmengen besitzen müssen. Diese Forderung ist bei den beiden vorangegangenen Typen verletzt, da hier das Element `geometry` auf zwei unterschiedliche Arten definiert ist.

Um diesem Problem zu begegnen, könnten die Inhaltsmodelle der kollidierenden Elemente vereinigt werden. Dieses würde jedoch den Wertebereich erweitern, da beispielsweise `staff_gif`-Objekte mit einer untergeordneten `colordepth` möglich wären.

Aus diesen und weiteren Gründen werden für die Definition der Datentypen *s-xscheme*s verwendet (s. Kapitel 3). Eine gemeinsame Struktur, die die beiden oben aufgeführten Bildtypen vereinigt, sieht wie folgt aus:

```
<!ENTITY  % sdtd_type        (staff_gif, medical_gif)>
<!ELEMENT staff_gif          color_intense, staff_geometry >
<!ELEMENT staff_geometry     height, width >
<!ELEMENT medical_gif        color_intense, medical_geometry>
<!ELEMENT medical_geometry   height, width, colordepth >
<!MAPPING geometry           staff_geometry, medical_geometry >
```

Durch die Anweisung `MAPPING` werden dem Ursprungselement `geometry` unterschiedliche erweiterte Elementnamen (`staff_geometry, medical_geometry`) zugeordnet, die je nach Kontext angewendet werden.

Die XML-Beschreibung der strukturierten Eigenschaft kann nun nachstehendes Aussehen haben:

```
<PROPERTY name="content" required="no" implied="no" fixed="no"
          system="no" semantic="Strukturierter Inhalt" type="staff_gif">
<staff_gif>
 <color_intense> 4 </color_intense>
 <geometry>
   <height> 100 </height>
   <width>  200 </width>
 </geometry>
</staff_gif>
</PROPERTY>
```

In umgekehrter Richtung können die bisher separat benutzten Ebenen der Objekt- und Hierarchiebeschreibung zusammengeführt werden. Abbildung 4.8 zeigt die Modellierung der beiden Ebenen mit einheitlichen Mitteln.

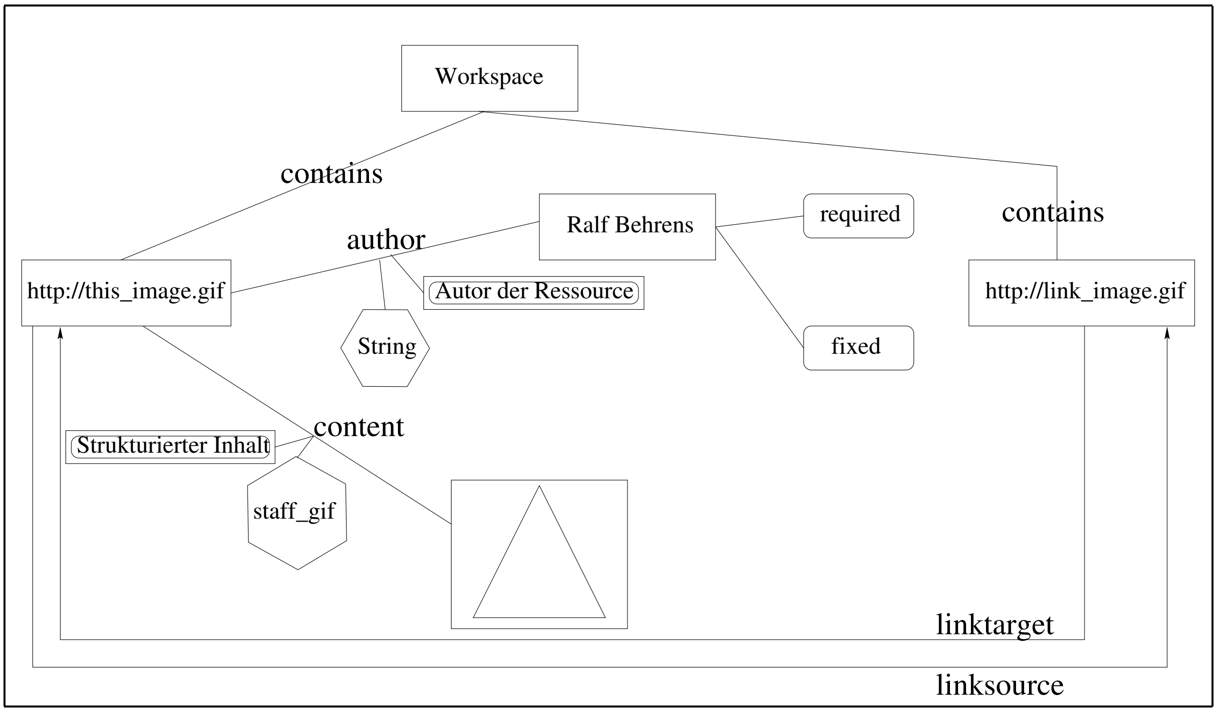


Abbildung 4.8: Gemeinsame Beschreibung von Hierarchien und Objekten

4.3 Semantik des Modells

Um ein Begriffsgerüst für die Beschreibung von Medienobjekten zu erhalten, ist eine Standardmenge von Eigenschaften notwendig. Idealerweise wäre hier die Verwendung von standardisierten Klassifikationsvokabularen möglich. Als Standardeigenschaften wurden die Attribute des *Dublin Core* ([DCM99]) gewählt, wobei die Semantik gemäß [Beh00b] verfeinert wurde. Das Dublin Core besteht aus einer Menge von fünfzehn Attributen, die im

Bereich „Digital Libraries“ zum Einsatz kommen. Diese Attribute werden bei unserer Verwendung zusätzlich, je nach Medientyp, um weitere Eigenschaften erweitert.

Wir werden hier unsere Interpretation der Dublin Core-Attribute für das Medienarchiv erläutern. Insbesondere wird dabei die Benutzung im Hinblick auf Lehr- und Forschungsmaterial berücksichtigt.

Title (Titel): Die Wahl des richtigen Titels ist oftmals eine nichttriviale Aufgabe. Im Fall von Vorlesungsskripten mag die Antwort klar sein. Werden einzelne Objekte betrachtet, so sollte der Titel deren Inhalt zusammenfassen oder deren Namen im Filesystem beschreiben.

Author (Autor): Die Person bzw. die Gruppe von Personen, welche das Objekt erzeugt hat. Dieses muss nicht zwangsläufig die Person sein, welche das Objekt innerhalb des Archivs besitzt.

Subject (Thema): Das Thema sollte eine Liste von Schlüsselwörtern in einer Standard-Terminologie sein. Für den medizinischen Bereich gibt es verschiedene solcher Klassifikationsvokabulare (wie z. B. das angesprochene MESH [mesb]).

Contributor (Verantwortlicher): Dieses ist die verantwortliche Person des Instituts (oder der Klinik), welche das Material zur Verfügung stellt.

Description (Beschreibung): Die Beschreibung gibt eine kurze verbale Charakterisierung des Objekts wider.

Publisher (Verleger): Der Begriff des Verlegers kollidiert unter Umständen mit dem Verantwortlichen. Wir haben hier für diesen Zweck die Oberorganisation (z. B. die Universität) gewählt.

Date (Datum): Das Datum, an dem das Objekt im Archiv abgelegt wurde. Das Datum kann in verschiedenen Formaten und Zeitzonen angegeben werden. Darüber hinaus gibt es unterschiedliche Verfeinerungen, wie **Date.LastModified** und **Date.Created**, mit der entsprechenden Semantik.

Type (Typ): Der Typ der Ressource ist eine wichtige Information für die Art und Weise wie diese verarbeitet wird. Typbezeichnungen soll informativer als herkömmliche Dateiendungen sein. Z. B. sollte es nicht nur ausdrücken, dass es sich bei einer Graphikdatei um das `jpg`-Format handelt, sondern auch, dass diese eine spezielle medizinische Konvention erfüllt.

Format (Format): Das Format spiegelt die MIME-Kodierung des Objekts, wie sie in RFC2046 [FB] definiert ist, wider.

Identifier (Identifikator): Dieses ist ein Identifikator, welcher das Objekt eindeutig innerhalb des Archivs kennzeichnet.

Source (Quelle): Die Quelle eines Objekts ist deren ursprüngliche Lokation außerhalb des Archivs. Diese Lokation kann eine URI (für den Fall, dass das Objekt allgemein zugreifbar ist) oder eine textuelle Beschreibung der Herkunft sein.

Language (Sprache): Die Sprache sollte „deutsch“ für die meisten Lehrmaterialien und

„english“ für zusätzliche Unterlagen und Begleitliteratur sein. Hier sind natürlich auch andere Sprachen möglich.

Relation (Beziehung): Durch die Beziehung werden Verknüpfungen zu verwandten Materialien ausgedrückt. Im Fall einer Graphikdatei könnte das eine Voranschau („Thumbnail“) sein.

Coverage (Bereich): Hierdurch wird der Bereich eingegrenzt, in dem das Objekt anzusiedeln ist. Dieses kann explizit vergeben werden oder auch durch den Benutzer oder dessen Institut abgeleitet werden.

Rights (Rechte): Jedes Objekt besitzt eine Reihe von Rechten, die durch dieses Attribut beschrieben werden.

Zusammenfassend ist festzustellen, dass nicht alle Dublin Core-Attribute genau den Bedürfnissen eines Medienarchivs entsprechen. Dennoch liefern sie einen mittlerweile anerkannten Rahmen zur Metabeschreibung von Objekten, welcher den Austausch der Informationen über das System hinaus ermöglicht. Bei Such- und Restrukturierungsoperationen können Attribute als „kleinster gemeinsamer Nenner“ aufgefasst werden, um so ein sprachliches Grundgerüst zu formen.

4.4 MDM-Integration von Objekten

Wenn ein Multimedia-Objekt in das Archiv importiert wird, werden mehrere aufeinanderfolgende Schritte ausgeführt (s. Abbildung 4.9).

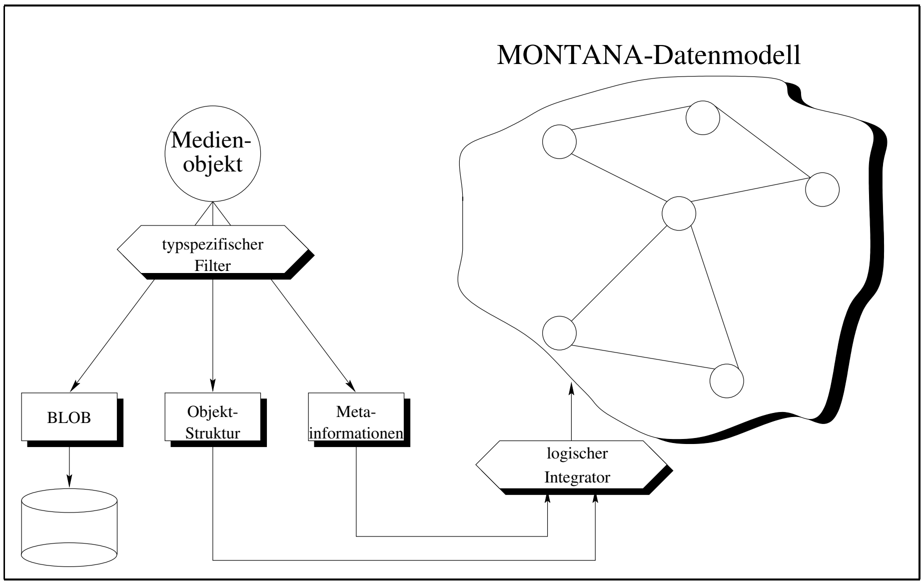


Abbildung 4.9: Überführung eines Medienobjekts in das MDM

Der spezielle Medientyp des Objekts wird ermittelt, und es wird der entsprechende Filter bestimmt. Der Filter extrahiert alle automatisch erkennbaren Eigenschaften und Strukturen. Diese werden über einen logischen Integrator in das MONTANA Datenmodell abgebildet.

Im Spezialfall eines XML-Objekts gibt es zusätzliche Möglichkeiten für die Speicherung und Transformation. Zunächst wird das Objekt mittels des XML-Schemaparsers in eine DOM-Form gebracht. Nun können reichhaltige Transformationsdienste (z. B. mit XSL), sowohl zur Formattransformation als auch für die benutzerdefinierte Layoutgestaltung angewendet werden.

4.5 Anfragen auf MDM-Daten

Die Suche von Medienobjekten stellt eine wichtige Funktionalität für die Benutzung des Archivs dar. Die entwickelte Anfrageschnittstelle ist allgemein gehalten. Es kann sowohl eine Stichwortsuche als auch ein komplexer Suchausdruck formuliert werden, wobei inhaltliche und strukturelle Gesichtspunkte herangezogen werden können.

Zur konkreten Suche wird eine XQL-Implementierung verwendet [Tec]. Beliebig komplexe Suchausdrücke werden gemäß der MDM-Anfragesyntax (s. Abbildung 4.10) formuliert. Die Ausdrücke werden gegen die Syntax geparst und in XQL-Anfragen umgesetzt. Für die Definition der Transformationsregeln und deren Ausführung wird der *JavaCC* [Met] verwendet.

<table>
<tr><td><suchstring></td><td>→</td><td><ausdruck> [[AND || OR] <suchstring>]?</td></tr>
<tr><td><ausdruck></td><td>→</td><td>NOT? [$x <xql-statement> $x
|| (<suchstring>)
|| “ ALPHANUM_SEQUENZ ”
|| <wort>]</td></tr>
<tr><td><wort></td><td>→</td><td>[$s || $i]? [<wertvergleich> || <attributvergleich>]</td></tr>
<tr><td><wertvergleich></td><td>→</td><td>WERT || ALPHANUM_SEQUENZ</td></tr>
<tr><td><attributvergleich></td><td>→</td><td>$w <attributname> || <attributname> <wertzuweisung></td></tr>
<tr><td><attributname></td><td>→</td><td>ALPHANUM_SEQUENZ</td></tr>
<tr><td><wertzuweisung></td><td>→</td><td>RELATIONSZEICHEN [[$s || $i]? WERT || ALPHANUM_SEQUENZ]]</td></tr>
</table>

Abbildung 4.10: Abstrakte Grammatik einer MDM-Anfrage

Ein Suchstring (<suchstring>) ist so aufgebaut, dass er beliebig viele Ausdrücke (<ausdruck>) durch Boolesche Operationen (AND, OR, NOT) verknüpft. Ausdrücke können sich entweder auf spezielle Eigenschaften (Merkmalssuche mit <attributname>) oder auf das gesamte Objekt (Volltextsuche) beziehen. Für Vergleiche können unterschiedliche Relationszeichen benutzt werden (RELATIONSZEICHEN). Das Nonterminal ALPHANUM_SEQUENZ steht dabei für eine beliebige alphanumerische Zeichenkette. Über ein <xql-statement> lassen sich XQL-Anweisungen spezifizieren, die auch eine Suche in der Struktur ermöglichen.

Über die Operatoren

- $w: lediglich nach dem Vorkommen des Attributs suchen
- $x: ein allgemeines XQL-Statement eingeben
- $s, $i: Groß- / Kleinschreibung beachten / nicht beachten

können spezielle Sucheigenschaften für Teilanfragen festgelegt werden.

Die folgenden Beispiele sollen die Definition von einfachen Anfragen auf einen MDM-Datenbestand veranschaulichen :

`"Großer Tumor" AND $sname=GIF`	Alle Objekte, die das Schlagwort „Großer Tumor“ (bzgl. Volltextsuche) enthalten und deren Name ein (großgeschriebenes) „GIF“ enthält.
`format=image/jpg OR size<=2.7MB`	Alle Objekte, deren MIME-Type „image/jpg“ lautet oder deren Größe 2.7 MB nicht übersteigt.
`$x //DESCRIPTION/PROPERTY [@name=structure]/head[2]/title $x`	Objekte, die eine Eigenschaft „structure“ besitzen, deren zweites Element ein Unterelement „title“ besitzt.

Auf der internen Ebene werden die MDM-Anfragen nach XQL umgesetzt. Die erste Anfrage (`"Großer Tumor" AND $sname=GIF`) hat beispielsweise die nachfolgende XQL-Repräsentation:

```
//DESCRIPTION[. $icontains$ 'Großer Tumor']/@about $intersect$
//DESCRIPTION[property='GIF' $and$
              property/@name='name']/@about
```

Die Anfragesprache ist auf unterschiedlichen Ebenen nutzbar (Stichwortsuche, inhaltliche Suche, strukturelle Suche). Im Gegensatz zu herkömmlichen Websuchmaschinen, die hauptsächlich stichwortartige Anfragen zulassen, können so mittels MDM-Anfragen auch strukturspezifische Eigenschaften angefragt werden.

5

Implementierung des Prototypen MONTANA

„Es gibt nichts Gutes, außer man tut es!“

Erich Kästner

Im Rahmen dieser Arbeit wurde das Medienarchiv MONTANA [MON] prototypisch implementiert. Ziel ist die Unterstützung von Lehre und Forschung an der Medizinischen Universität zu Lübeck. Das Medienarchiv entstand als Teil des Multimedia-Entwicklungszentrums Schleswig Holstein (*MESH*) [MESa][1].

5.1 Historie

Der erste Prototyp des Medienarchivs [LL98] wurde im Jahr 1997 fertig gestellt. Um den potentiellen Benutzern möglichst schnell ein lauffähiges Medienarchiv zur Verfügung zu stellen, wurde die erste Version relativ zügig implementiert. Bei dieser Version handelte es sich im Wesentlichen um einen Webserver, der seine Daten innerhalb einer relationalen Datenbank verwaltete.

Die erste Medienarchiv-Version präsentierte dem Benutzer ein virtuelles, NFS-ähnliches Filesystem. Dabei war es möglich, Objekte mit Attributen aus einer vorgegebenen Menge zu annotieren und anhand einzelner Merkmale wiederzufinden.

Für die Benutzer stellte die Einführung des Medienarchivs grundsätzlich eine große Erleichterung dar. So profitierte insbesondere der Lehrbetrieb von den neu zur Verfügung stehenden Möglichkeiten. Es stellte sich aber auch heraus, dass die manuelle Attributierung von Medienobjekten eine aufwendige Arbeit war, die nur von wenigen Anwendern

[1]Gefördert im Rahmen des Hochschulsonderprogramms (HSP3) durch das Bundesministerium für Bildung, Forschung und Kultur (BMBFK) und das Land Schleswig-Holstein

genutzt wurde. Auch führten das Fehlen von Linkobjekten und mangelnde serverseitige Restrukturierungs- und Suchmöglichkeiten zu einer eingeschränkten Einsetzbarkeit des Archivs.

Aus technischer Sicht hat sich gezeigt, dass die verwendeten Technologien für die Benutzung innerhalb des Medienarchivs ungeeignet sind. Das eingesetzte *Common-Gateway-Interface (CGI)* ist aus Performance-Gesichtspunkten ein Flaschenhals, da bei einer großen Anzahl paralleler Benutzerzugriffe die Kapazität des Servers schnell erschöpft war. Zudem offenbarte sich die Implementierung in der Sprache *C* als sehr umständlich und fehlerträchtig, da grundlegende Webtechniken nicht unterstützt werden.

Die Erfahrungen mit dem ersten Prototypen führten darüber hinaus zu dem Schluss, dass die konventionelle Informationsmodellierung unter Verwendung eines Datenbanksystems mit festen Datenstrukturen zu unflexibel für die wechselnden Anforderungen im Kontext eines Medienarchivs ist. Daher wurde 1999 damit begonnen, einen zweiten Prototypen zu konzipieren und zu realisieren, bei dem die Informationsmodellierung basierend auf XML und den in den vorherigen Kapiteln ausgeführten Überlegungen durchgeführt wird.

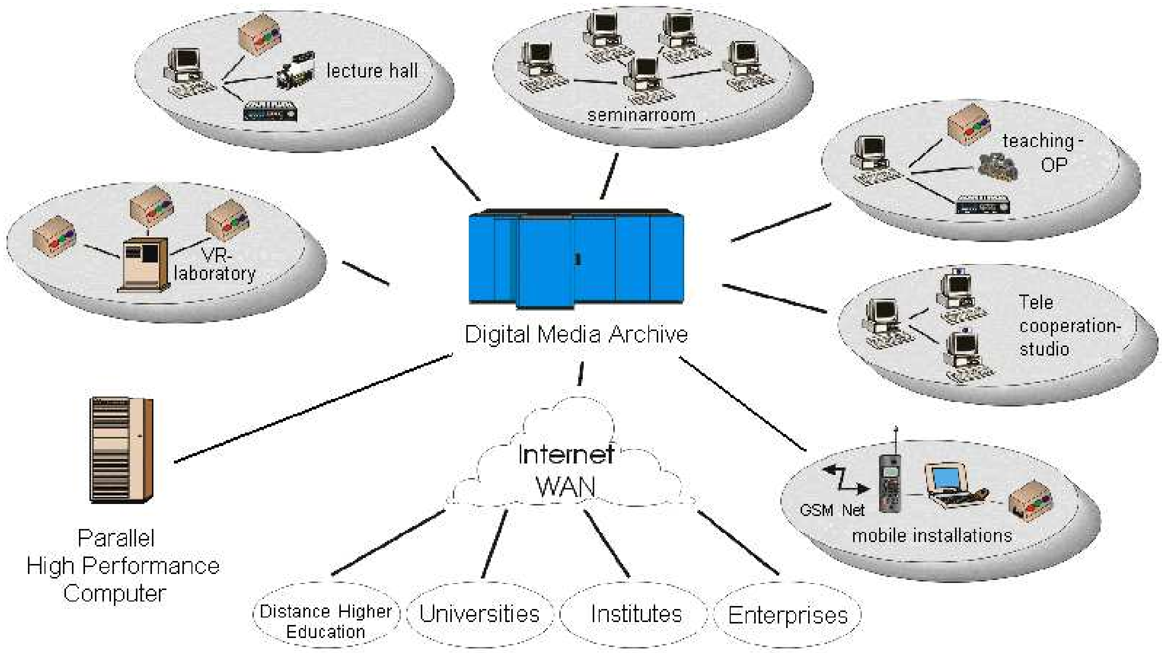


Abbildung 5.1: Medienarchiv MONTANA in der globalen Infrastruktur

5.2 Szenario

Die Projektgruppe „*Medienarchivserver*“, die sich die Konzipierung und den Entwurf des Medienarchivs zum Ziel gemacht hat, besteht aus 15 Instituten und Kliniken der Medizinischen Universität zu Lübeck. Innerhalb dieser Projektgruppe wurden die strategischen Entscheidungen getroffen. Da die Gruppe sowohl aus Informatikern als auch aus Medizinern besteht, ergab sich ein breiter Forderungs- und Anwendungskatalog, welchen es innerhalb des Archivs zu realisieren galt. Abbildung 5.1 zeigt die zu Beginn des Projekts erarbeitete Vision für das Archiv im Universitätsumfeld.

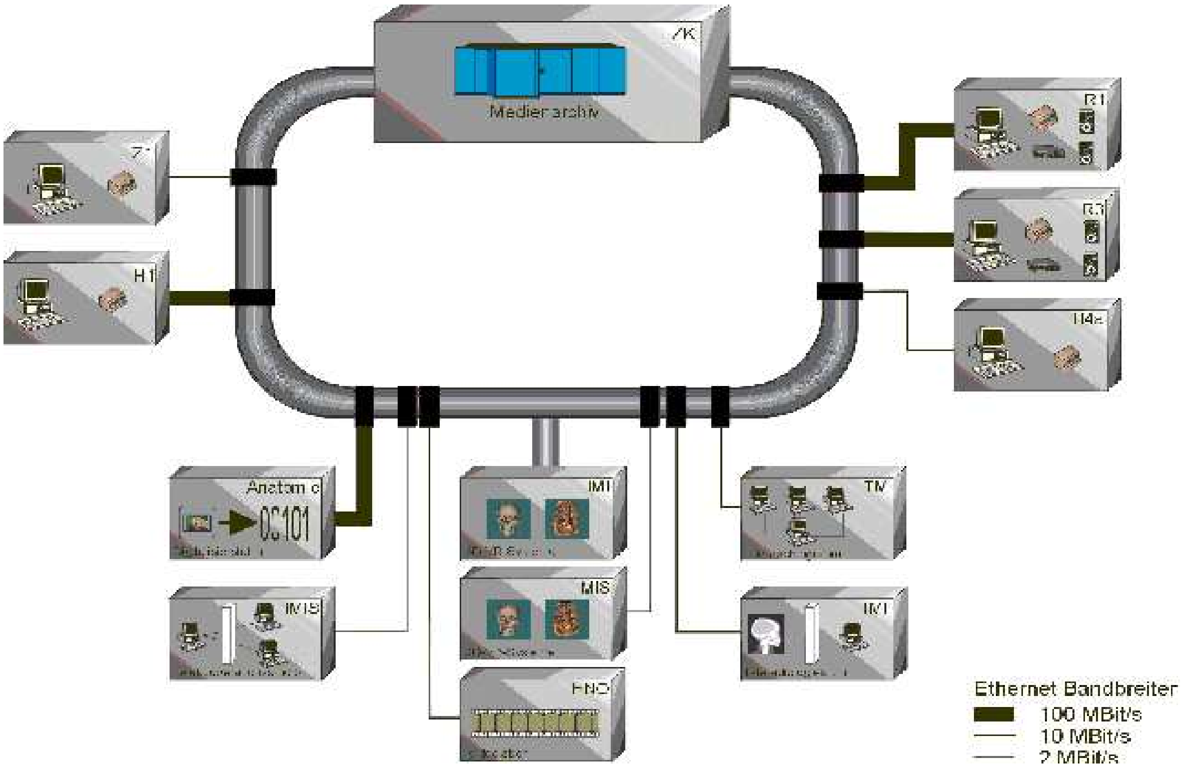


Abbildung 5.2: Anbindung der Institute und Kliniken der MUL an das Archiv, Stand Mitte 2000

Eine andere Herausforderung ergab sich aus der Tatsache, dass die verschiedenen Einrichtungen sowohl auf dem Campus als auch in externen Gebäuden untergebracht waren (und immer noch sind). Dieses führte zu sehr unterschiedlichen Anbindungsbandbreiten. So wurden innerhalb des Campus 100 MBit Verbindungen installiert, welche teilweise durch Lichtfaserleitungen (FDDI) ergänzt wurden. Dagegen waren außenliegende Institute nur über 2-10 MBit-Anbindungen oder gar gebündelte ISDN-Leitungen angeschlossen. Abbildung 5.2 [2] spiegelt diese Situation wider.

[2]Graphik stammt aus einem Vortrag von M. Herczeg (IMIS/MUL) im Rahmen eines MESH-Projekttreffens

Für das Archiv wurde ein Server (*SGI Origin 2000*) als Ablaufplattform angeschafft, welcher zentral positioniert wurde und für alle Einrichtungen (mit den oben angesprochenen Bandbreiten) erreichbar ist.

5.3 Architektur des Prototypen

Es sollen nun die zugrundeliegende Architektur und die Implementierung der eigentlichen Funktionalitäten beschrieben werden [Beh00b]. Der realisierte Prototyp [MON] genügt einer 3-schichtigen Architektur, die in Abbildung 5.3 zu sehen ist.

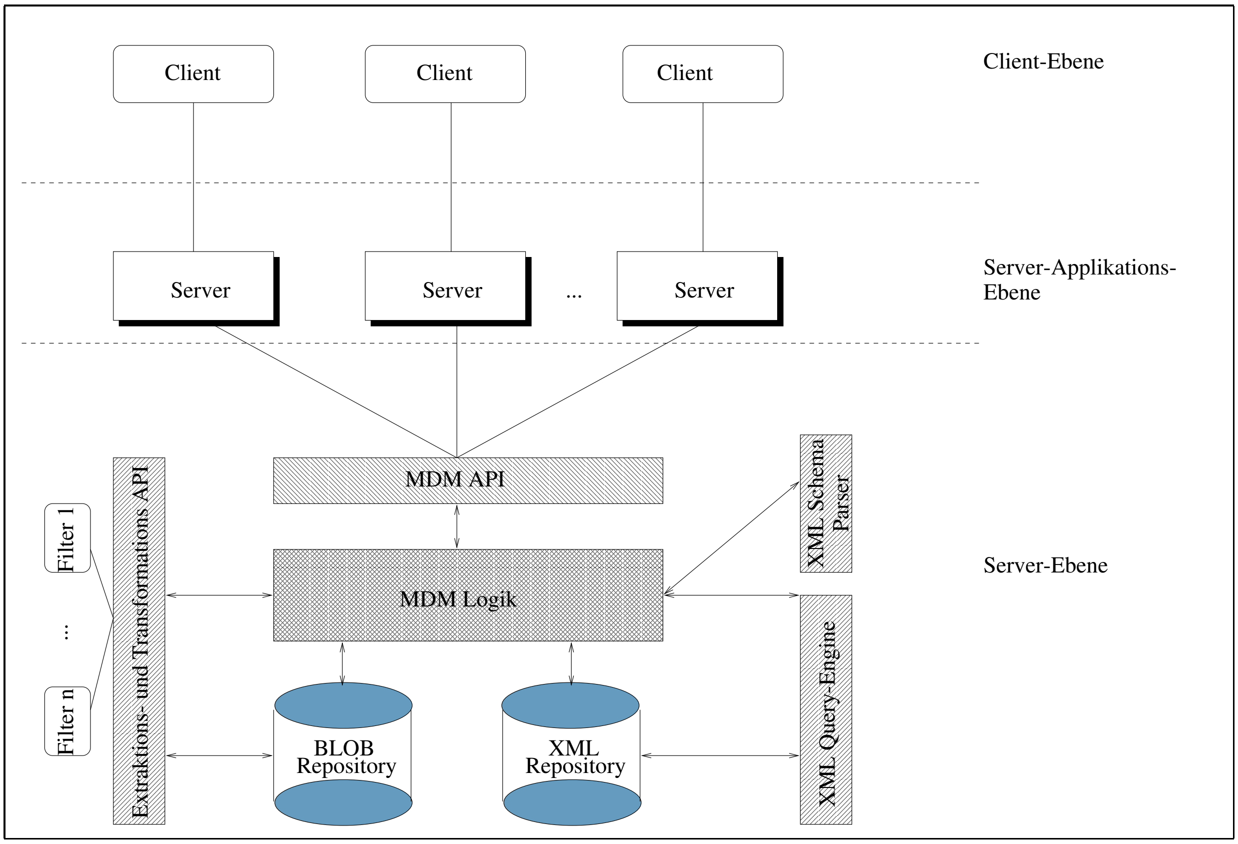


Abbildung 5.3: Architektur des Medienarchivs

Die Architektur setzt das *Model-View-Controller-Konzept* [GHJV95, Rum94] um. Dieses Konzept ist ein mittlerweile allgemein anerkannter und genutzter Ansatz, der für größere Implementierungen die Trennung der logischen Ebenen (Modell, View, Controller) definiert.

Im MVC-Paradigma sind die Benutzereingabe, die Modellierung der realen Welt und die Präsentation für den Anwender explizit getrennt. Das MVC-Konzept besteht demzufolge aus drei Teilen.

Der *Controller* beinhaltet die Logik und bearbeitet die anstehenden Ereignisse. Das *Modell* enthält die Daten oder weiß zumindest, wo es die Daten findet. Der *View* ist für die Darstellung der Daten zuständig. Der Controller erhält alle auftretenden Ereignisse. Er entscheidet, ob ein Ereignis für den View oder für das Modell gültig ist. Er informiert daraufhin das Modell oder den View über Änderungen, die aufgrund eines Ereignisses durchgeführt werden sollen. Das Modell wertet diese Informationen aus und bearbeitet sie. Falls sich dadurch Änderungen im Modell ergeben, informiert es seinerseits den View darüber. Der View holt sich nun die Daten vom Modell, die zur Darstellung der gewünschten Informationen benötigt werden.

Einer der Vorteile des MVC-Konzepts ist die Unabhängigkeit des Modells vom View. Es ist völlig unerheblich, woher die Daten kommen, wie sie gespeichert und verwaltet werden.

Die drei Ebenen des MVC-Konzepts entsprechen im Wesentlichen der 3-schichtigen Architektur des Archivs in Abbildung 5.3. Das Modell wird durch die MDM-Logik umgesetzt. Der View, also die benutzerorientierte Präsentation, geschieht je nach Clientanwendung über die entsprechende Schnittstelle. Der Controller ist im Zusammenspiel zwischen Clientanwendung und MDM-API positioniert. Er dient dem Anstoßen der Aktionen aufgrund von Ereignissen im jeweiligen View.

Wir werden nun die Implementierung der einzelnen Teile des Archivs näher beschreiben. Dabei werden die verwendeten Technologien erläutert, soweit sie nicht als bekannt vorausgesetzt werden können und für das Verständnis erforderlich sind.

5.3.1 Java

Die Implementierung des Archivs wurde komplett in der Sprache *Java* [Javb, AG00, Li98] realisiert. Java wird als eine objektorientierte, einfache, interpretierte, architekturunabhängige, portable, netzwerkfähige, sichere und Multithread-fähige Programmiersprache propagiert. Was diese Eigenschaften bedeuten, soll im Folgenden näher erläutert werden.

Objektorientierung:

Die Objektorientierung ist eine der zentralen Eigenschaften von Java und bringt im Vergleich zu prozeduralen Programmiersprachen durch die Modellierung in Klassen wesentliche Vorteile mit sich. Zu erwähnen sind dabei vor allem die einfachere Wiederverwendbarkeit und die Erweiterbarkeit der Klassen, sowie die saubere Definition der Schnittstellen des Systems.

In Java gibt es mit Ausnahme der primitiven Datentypen (`boolean, char, byte, short, int, long, float, double`) nur Objekte. Im Gegensatz zu hybriden Sprachen wie z. B. C++ [Str86], wird der Programmierer hierdurch gezwungen, objektorientiert zu programmieren. Mit Hilfe von Zugriffsdeklarationen (`public, protected, private`) kann der Programmierer einer Klasse die exportierte Schnittstelle schützen. Er kann damit entscheiden, ob und welche Implementierungsdetails für andere Klassen sichtbar sind.

Bei den Klassen wird zwischen Schnittstellen (`interface`) und „echten“ Klassen (`class`) unterschieden. Während ein Interface lediglich die Schnittstelle, also die Methoden mit Übergabe- und Rückgabetyp definiert, enthält eine Klasse die Implementierung aller Methoden. Eine Mischform bilden abstrakte Klassen (`abstract class`), welche sowohl Methodensignaturen als auch Methodenimplementierungen enthalten.

Eine weitere wichtige Eigenschaft eines Java-Objekts ist dessen Fähigkeit, die Lebensdauer des Objekts, von dem es erzeugt wurde, zu überdauern. Damit ergibt sich vor allem im Bereich Client/Server-Anwendungen die Möglichkeit, ein Objekt zu erzeugen, über das Netz zu verschicken und an anderer Stelle zu speichern.

Einfachheit:

Die Syntax von Java ist ähnlich der von C/C++. Im Gegensatz zu C/C++ finden aber explizite Zeiger in Java keine Verwendung mehr. Mit der impliziten Zeigerarithmetik wurde eine große Fehlerquelle beseitigt. Ein zweiter wesentlicher Punkt ist die Speicherverwaltung, die bei Java durch eine automatische *Garbage Collection* übernommen wird. Das bedeutet, dass der von einem Objekt belegte Speicher automatisch wieder freigegeben wird, falls keine Referenzen mehr auf das Objekt existieren.

Im Unterschied zu C/C++ besitzen die Basisdatentypen in Java auf jedem Rechner dieselbe Bitlänge. Auch das von C++ bekannte *Operator-Overloading* und die *Mehrfachvererbung* wird bei Java nicht unterstützt. Die Interface-Technik bietet sich jedoch als eingeschränkter Ersatz für die Mehrfachvererbung an.

Die Ausnahmebehandlung erfolgt bei Java mit Hilfe von *Exceptions*, die vom Programmierer abgefangen werden müssen. Damit wird der Datenfluß vom Kontrollfluß getrennt, was eine einfachere Fehlerbehandlung ermöglicht.

Architekturunabhängigkeit und Portabilität:

Java ist eine auf die Verwendung in heterogenen Netzwerken ausgelegte Programmiersprache. Um den Einsatz von Java auf verschiedenen Plattformen mit unterschiedlichen Betriebssystemen und Benutzeroberflächen zu ermöglichen, wurde schon bei der Entwicklung der Sprache großer Wert auf Architekturunabhängigkeit und Portabilität gelegt. Der Source-Code wird nicht, wie bei vielen anderen Sprachen, vom Compiler in einen Maschinencode übersetzt, sondern in einen Byte-Code transferiert. Dieser Byte-Code ist maschinenunabhängig und wird durch die virtuelle Java-Maschine (*JVM*), ein plattformspezifischer Interpreter, ausgeführt.

Architekturabhängige Betriebssystemfunktionen kann der Programmierer aus Java heraus nicht ohne Aufwand aufrufen. Vielmehr sollten die mitgelieferten Klassenbibliotheken genutzt werden, um die Portabilität des Programms zu gewährleisten.

Netzwerkfähigkeit und Sicherheit:

Java besitzt eine native Netzwerkfähigkeit. Das bedeutet, dass durch die Standardklassenbibliothek bereits Zugriffe auf lokale und globale Netze unterstützt werden. Diese Netzwerkfähigkeit ist die Grundlage für Techniken wie Classloading, das später angesprochene

RMI oder auch der allgemeinen relationalen Datenbankschnittstelle JDBC.

Durch die Möglichkeit, Klassen über das Netz zu laden und zur Laufzeit dynamisch einzubinden, kommt dem Sicherheitsaspekt in Java eine besondere Bedeutung zu. Wie angesprochen, gibt es in Java keine expliziten Zeiger. Durch die automatische Speicherverwaltung kann ein zu frühes oder zu spätes Freigeben eines Speicherbereichs nicht mehr vorkommen. Der Vorteil liegt darin, dass ein Java-Programm durch Implementierungsfehler auf dieser Ebene nicht in einen undefinierten Zustand gebracht werden kann, was wesentlich zur Stabilität und Betriebssicherheit der Java-Programme beiträgt.

Vor dem Programmstart wird der Code, der nicht vom lokalen System stammt, vom Byte-Code-Verifizierer zu weiteren Tests herangezogen. Von Hand modifizierter oder von einem manipulierten Compiler erzeugter Byte-Code kann somit nicht zur Ausführung kommen. Ist dieser Sicherheitscheck passiert, wird jeder Klasse vom *Classloader* ein eigener Namensraum zugeordnet. Für lokale Klassen, die sich im Filesystem des Client befinden, und für Klassen, die über das Netzwerk geladen werden, existieren verschiedene Namensräume. Klassen unterschiedlicher Herkunft aber mit gleichem Namen werden also in verschiedenen Namensräumen abgelegt und können sich so nicht gegenseitig beeinflussen. Mit dem Konzept der getrennten Namensräume ist es nicht möglich, gewisse Klassen auszublenden und durch manipulierte Klassen, die von einer anderen Netzwerkadresse stammen, zu ersetzen. Wird eine Klasse benötigt, sucht der Classloader immer zuerst im Namensraum des lokalen Systems und dann erst in den externen Namensräumen. Lokal installierte Klassen haben damit immer Vorrang gegenüber Klassen, die vom Netzwerk geladen werden.

Multithreading und Ausführungsgeschwindigkeit:

Nebenläufige Prozesse (*Threads*) erlauben die quasi-parallele Abarbeitung von verschiedenen Programmteilen. So kann zum Beispiel ein Java-Server gleichzeitig Anfragen von mehreren Java-Clients bearbeiten. Für den Benutzer ergibt sich dadurch eine subjektiv schnellere Ausführungsgeschwindigkeit. Im Sprachumfang von Java existieren Klassen, die das Arbeiten mit solchen Threads sehr bequem machen. Dieses ist eine Erleichterung gegenüber Sprachen wie C/C++, wo diese Technik nicht direkt durch die Sprache unterstützt wird. Der Programmierer muss dort auf Betriebssystemfunktionen zurückgreifen, was die Programmierung erschwert und die Portabilität einschränkt.

Da der Java-Byte-Code zur Laufzeit auf jeder Maschine von einem Java-Interpreter interpretiert werden muss, ergeben sich im Vergleich zu C/C++ Geschwindigkeitsnachteile. Wird aus Performance-Gründen mehr Wert auf eine schnellere Ausführung des Programms gelegt, so kann der Code vor der Ausführung dynamisch mit Hilfe eines Native-Code-Generators in Maschinencode umgewandelt werden. Der damit erzeugte Programmcode ist ähnlich effizient wie nativer Code. Bedingt durch diverse Syntaxchecks des Byte-Code-Verifizierers ist allerdings vor dem Programmstart eine etwas längere Zeitspanne nötig, bis das Programm zur Ausführung kommt. Diese muss durch das dynamische Laden des Java-Codes für den Anwender allerdings nicht unbedingt zu bemerken sein. Eine andere Technik benutzen die Just-In-Time-Compiler, die den Byte-Code während des Interpretierens, d. h. kurz vor der tatsächlichen Ausführung, in Maschinencode umsetzen.

Applets und Applikationen:

Prinzipiell gibt es bei Java zwei unterschiedliche Programmtypen. *Applets* sind kleine Java-Programme, die in HTML-Seiten eingebettet und beim Laden dieser Seiten automatisch von einem Java-fähigen WWW-Browser ausgeführt werden. Für sie gelten zusätzliche Sicherheitsbeschränkungen, die sich über den Browser einstellen lassen. Abhängig vom WWW-Browser kann so das Lesen, Modifizieren, Erzeugen oder Löschen von ursprünglich geschützten Ressourcen (wie z. B. lokalen Dateien) erlaubt werden. Die Sicherheitsrestriktionen werden durch die sogenannte „Sandbox“ überwacht, welche Java-Klassen aufgrund ihrer Ausführungsumgebung einzelne Rechte gewährt oder verweigert.

Für Java *Applikationen* gelten diese Sicherheitsbeschränkungen nicht. Sie sind vom Funktionsumfang direkt mit Programmen in anderen Programmiersprachen vergleichbar. Einer Applikation ist es im Unterschied zu einem Applet also möglich, Dateioperationen durchzuführen oder über das Netzwerk Verbindung zu anderen Rechnern aufzunehmen. Applikationen können beliebig große Programme sein, die ohne einen WWW-Browser mit Hilfe der JVM ausgeführt werden.

Weitere Entwicklung von Java:

Der Umfang der Java-Klassenbibliothek hat seit der ersten Version stetig zugenommen. Mittlerweile ist Java in der Version 1.3, welche auch als *Java Platform 2* bezeichnet wird, verfügbar. Diese Plattform wird in drei Varianten angeboten:

- *Java 2 Standard Edition (J2SE)*: Die J2SE ist die frei verfügbare Version, die als Nachfolger der ersten Java-Versionen anzusehen ist. Laut Aussagen des Herstellers SUN soll der Funktionsumfang in Zukunft nicht weiter steigen. Zusätzliche Pakete werden dann in speziellen Bibliotheken zur Verfügung gestellt.

- *Java 2 Enterprise Edition (J2EE)*: Bei der J2EE handelt es sich um eine kommerzielle Version, die für die Entwicklung unternehmensweiter Java-Anwendungen eingesetzt wird. Inbesondere kommen hier die unten näher erläuterten Enterprise Java Beans zum Einsatz.

- *Java 2 Micro Edition (J2ME)*: Die J2ME soll für die Entwicklung von Applikationen für mobile Geräte wie Handhelds oder Mobiltelefone genutzt werden.

Die wichtigsten Neuerungen der aktuellen Java-Versionen gegenüber Vorgänger-Releases werden hier kurz angerissen:

- **Java Database Connectivity (JDBC)**: Hierbei handelt es sich um eine allgemeine, relationale Datenbankschnittstelle. Durch diese Schnittstelle wird es über entsprechende Treiber möglich, auf relationale Datenbanken unterschiedlicher Hersteller zuzugreifen. Der Code bleibt im Idealfall nach einem Austausch des Datenbanksystems (und des entsprechenden Treibers) unverändert.

- **Java Remote Methode Invocation (RMI)**: Java RMI bietet die Möglichkeit eines verteilten Objektzugriffs. Damit wird eine transparente Aufteilung zwischen Client und Server möglich (Abschnitt 5.3.3).

- **Java Beans**: Durch Java Beans wird der Komponentenansatz auf Java übertragen [Pie99]. Softwarekomponenten sollen der stärkeren Wiederverwendbarkeit einmal entwickelter Programmteile dienen (Abschnitt 5.3.4.).

- **Enterprise Java Beans (EJB)**: EJB dienen der Umsetzung unternehmensweiter Geschäftsprozesse unter Verwendung von Java Beans. Es werden insbesondere in einem solchen Umfeld notwendige Dienste wie Transaktionen, Concurrency Control und Persistenz unterstützt (Abschnitt 5.3.5).

- **Swing**: Swing ist eine plattformübergreifende Graphikbibliothek, die weit über die Möglichkeiten des *Abstract Windowing Toolkit (AWT)* hinausgeht. Das AWT beschränkt sich auf graphische Elemente, die von allen gängigen Plattformen unterstützt werden. Dagegen erweitert Swing diese Bibliothek dahingehend, dass es eine Vielzahl von erweiterten graphischen Elementen und Tools auf der Basis der AWT-Komponenten nachbildet.

- **Java Native Interface (JNI)**: Das *JNI* dient der Einbindung und dem Aufrufen von Funktionen und Methoden anderer Sprachen (wie z. B. C/C++) aus Java-Programmen heraus.

- **Java Naming and Directory Interface (JNDI)**: Das *JNDI* ist eine API, die Namens- und Verzeichnisdienste zur Verfügung stellt. Die Idee eines Namens- und Verzeichnisdienstes ist die Verknüpfung von Objekten mit eindeutigen Identifikatoren, um sie so namensorientiert zugreifbar zu machen. Diese Dienste dienen als Grundlage für weitere Technologien wie RMI und Enterprise Java Beans.

Zusammenfassung und Verwendung im Archiv:

Die Sprache (oder genau genommen Plattform) Java stellt eine vielversprechende Alternative zu herkömmlichen Sprachen dar. Durch die systemübergreifend einheitliche Plattform werden Entwicklungsprozesse vereinfacht und kostengünstiger realisierbar. Auch die implizite Netzwerkfähigkeit stellt einen wichtigen Schlüssel für den Einsatz in globalen Netzen dar. Während erste Java-Versionen noch einen bescheidenen Funktionsumfang besaßen, warten die aktuellen Editionen mit einer Flexibilität auf, die sie für die Verwendung in großen Projekten und Unternehmensanwendungen geeignet erscheinen lassen.

Als nachteilig erweist sich die geringe Performance von Java-Applikationen. Um diese zu umgehen, sind oftmals plattformabhängige Optimierungen notwendig. Es ist anzunehmen, dass zukünftige Entwicklungen der Sprache hier Abhilfe schaffen.

Die Implementierung des Medienarchivs wurde aufgrund der genannten Vorteile nahezu ausschließlich Java-basiert realisiert. Wir werden im weiteren Verlauf des Kapitels auf die für unsere Zwecke wichtigen Java-Techniken näher eingehen.

5.3.2 XML und Java

Obwohl sich zur Verarbeitung von XML-Daten grundsätzlich jede Programmiersprache eignet, kommt Java eine besondere Bedeutung zu. Java kann als programmiersprachliches Pendant zur Dokumentbeschreibungssprache XML gesehen werden. Während XML einen anwendungsunabhängigen Datenaustausch gewährleistet, ist Java als plattformunabhängige Sprache ideal für dessen Verarbeitung geeignet [Bos97].

Darüber hinaus kommen der Kombination dieser beiden Paradigmen auch technische Details zu Gute. XML benutzt einen plattformübergreifenden Unicode-Zeichensatz [Uni], der die meisten internationalen Zeichensätze subsumiert. Dieser 16-Bit Zeichensatz wird auch in Java für die Zeichenkodierung verwendet.

Eine Folge dieser engen Beziehung ist, dass eine große Anzahl von XML-Werkzeugen anfänglich und oftmals ausschließlich in Java implementiert wird.

5.3.3 Java Remote Method Invocation (RMI)

Die *Java Remote Method Invocation (RMI)* zielt auf die Entwicklung verteilter *Java-to-Java*-Anwendungen hin, die in unterschiedlichen JVMs ablaufen. Das Konzept vom RMI ist nicht völlig neu, sondern ähnelt *CORBA* [Gro], welches seinerseits den *Remote Procedure Call (RPC)* auf die objektorientierte Welt überträgt. Im Gegensatz zu CORBA ist RMI eine homogene Java-Lösung. Mit RMI kann der Programmierer verteilte Anwendungen schreiben, ohne bei deren Programmierung in die Tiefen der Netzprogrammierung vorstoßen zu müssen.

RMI-Architektur:

Die RMI-Architektur besteht aus vier Komponenten; einem Server, der die Remote-Dienste bereitstellt, einem Client der diese Dienste in Anspruch nimmt, einer *Registry* mit *Naming Service*, die ein Repository für die Remote-Dienste darstellt und der Kommunikationskomponente. Client und Server können in dieser Architektur auf zwei verschiedene Hosts verteilt sein. Trotzdem ermöglicht es RMI dem Client, die Methoden des Servers aufzurufen, als seien sie lokal vorhanden.

Die Komponenten des RMI sind, wie in Abbildung 5.4 sichtbar, in drei Ebenen untergebracht. Dieses sind die *Stub/Skeleton*-Ebene, die *Remote-Reference-Layer* und die *Transport*-Ebene. Jede Ebene besitzt eine eigene Schnittstelle mit einem spezifischen Protokoll und ist dadurch von den anderen Ebenen unabhängig.

Auf der Transport-Ebene ist die Netzkommunikation angesiedelt. Hierfür kommt TCP/IP zum Einsatz, auf welchem das für die Kommunikation zwischen Client und Server verwendete *RMI Wired Protocol (RWP)* aufbaut. Im proprietären Protokoll RWP benutzt RMI zudem die *Java Object Serialization*, die der Überführung von Objekten in Byteströme zwecks Netzübertragung dient. *SUN* kündigte bereits an, in den nächsten Versionen von RMI dieses Protokoll durch das standardisierte *Internet Inter ORB Protocol (IIOP)* der *Object Management Group (OMG)* [OMG] zu ersetzen, welches auch in CORBA eingesetzt wird.

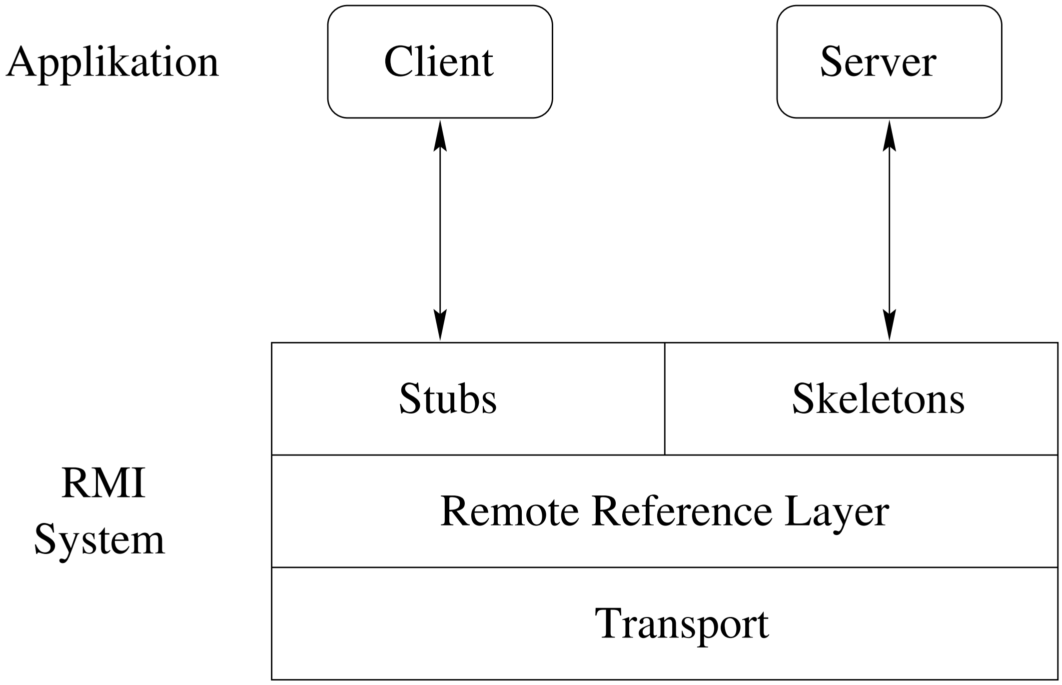


Abbildung 5.4: Allg. Ebenenarchitektur von RMI

Die Remote-Reference-Layer ist für die Umsetzung der Operationen in entfernte Aufrufe verantwortlich. Hier werden die Anfragen in transparenter Weise an Remote-Objekte weitergereicht, Rückgabewerte verarbeitet und Spezialfälle abgehandelt.

Der Programmierer hat nur Kontakt mit der obersten Schicht, den *Stubs* und *Skeletons*. Bei der Methodenausführung kooperieren die Komponenten der Client- (Stub) und Serverseite (Skeleton).

Die Client-Komponente enthält die relevanten Informationen über die Serverimplementierung des Objekts und kann z. B. bei einem replizierten Server-Objekt den Methodenaufruf an alle Replikate weiterleiten. Die serverseitige Komponente der Remote-Reference-Layer dient der Weiterleitung des Aufrufs an das Skeleton-Objekt. Wenn von einem Objekt im Client eine Methode eines Serverobjekts aufgerufen werden soll bzw. dem Serverobjekt eine Nachricht gesendet werden soll, wird der Stub verwendet, welcher als lokales Stellvertreterobjekt dient. Dieser existiert im Adressraum der JVM, die das Clientobjekt ausführt und stellt somit für die restliche Applikation ein reguläres Java-Objekt dar. Die Stub-Methode fasst die Parameter des Methodenaufrufs zusammen (*parameter marshalling*). In Verbindung mit der eindeutigen Referenz des Remote-Objekts und der Nummer der aufgerufenen Methode wird ein Datenblock gebildet, der über die Remote-Reference-Layer zum Server gesendet wird.

Auf der Serverseite empfängt das Skeleton über die dortige Remote-Reference-Layer den

Datenblock und ruft die gewünschte Methode des Serverobjekts auf. Rückgabewerte, wozu auch ausgelöste Ausnahme- bzw. Fehlerobjekte (Exceptions) gehören, werden auf analogem Weg zum Clientobjekt zurückgesendet.

Dynamic Class Loading:

Ein Client im RPC-Umfeld benötigt kompilierten Code für Stubs, der vor der Ausführung der Routine statisch oder dynamisch (über ein Netzwerk-Filesystem) zum aktuellen Programm gebunden wird.

RMI generalisiert diese Technik insofern, dass neutraler Bytecode (d. h. herkunftsunabhängiger Code) verwendet wird. Dieses ermöglicht bzw. vereinfacht den heterogenen Mix von Client- und Serverplattformen. Ein wahlfreies Laden von Klassen der lokalen oder entfernten Quelle wird möglich, indem ein spezieller *RMIClassLoader* zur Verfügung gestellt wird. Damit ist eine Kontrolle und genaue Steuerung der Klassenherkunft gewährleistet, da das Laden

- von entfernten Objektklassen und -schnittstellen
- von Stub- und Skeletonklassen
- auch von anderen, in der RMI-Anwendung benötigten Klassen, wie z. B. Parametern von Methoden oder Rückgabeobjekten

ermöglicht wird.

Lokalisierung entfernter Objekte:

Eine Kommunikation zwischen Client und Server erfordert eine Referenz (Stub) des entfernten Objekts. Man erhält diese im allgemeinen als Rückgabewert eines Methodenaufrufs für ein entferntes Objekt. Dafür benötigt man jedoch wiederum den Stub dieses entfernten Serverobjekts.

Um erstmalig eine Referenz auf ein entferntes Objekt zu erhalten (*bootstraping*), wird die Registry verwendet. Die Registry liefert einen einfachen Namensdienst. Ein Registry Service kann dabei Namen an eine Referenz binden, Bindungen ändern und löschen.

Das Java Development Kit enthält eine Standard-Registry, welche das Java Naming and Directory Interface (JNDI) benutzt, bietet dem Anwendungsentwickler aber auch über entsprechende Packages die Möglichkeit, einen eigenen Registry Service zu realisieren.

Ein Registry Service läuft als Dämonprozess auf dem Serversystem und wacht an einem Port auf eine Dienstanforderung. Zum Ansprechen der Registry wird die Syntax für einen Uniform Resource Locator (URL) verwendet, z. B. `rmi://www.medienarchiv/myObject`.

Der Name eines Objekts wird durch die Implementierung auf Serverseite vergeben (*binding*). Diese bindet die frei gewählte, aber eindeutige Bezeichnung an eine Instanz eines Remote-Objekts. Ein Client erhält über den Namen (*lookup*) des entfernten Objekts eine Referenz und kann somit das Serverobjekt ansprechen.

Beispiel für das Medienarchiv:

Der Entwicklungsprozess einer RMI-Applikation soll anhand eines kleinen Beispiels schrittweise skizziert werden. Hierbei handelt es sich um die zentrale Klasse `MDMObject` des MDM, die später noch genauer erläutert wird.

Im ersten Schritt der Erstellung einer verteilten RMI-Anwendung muss die Schnittstelle des Remote-Objekts (*Remote Interface*) definiert werden. Diese Schnittstelle legt die zu exportierenden Methoden und Variablen des Remote-Objekts fest. Aus syntaktischen Gründen muss das Remote Interface das Java Interface `java.rmi.Remote` erben. Außerdem muss jede definierte Methode zwecks Fehlerbehandlung eine `java.rmi.RemoteException` werfen (`throws`).

```
public interface MDMObject extends java.rmi.Remote // Remote Interface
{
    ...
    // Definition der Methode, die
    // später 'remote' genutzt werden soll
    public String getName() throws java.rmi.RemoteException;
    ...
}
```

Nach der Definition der Schnittstelle wird die Implementierung des zugehörigen Objekts auf dem Server benötigt, d. h. eine Klasse, die auf dem Server die Methoden des Interface umsetzt und auch syntaktisch die zugehörige Schnittstelle erbt.

```
public class MDMObjectImpl implements MDMObject // Remote Object
{
   ...
   // Implementierung der 'remote'-Methode
   public String getName() throws java.rmi.RemoteException
   {
       String name;
       ...
       return name;
   }
   ...
}
```

Bei diesem Beispiel ist zu beachten, dass der zurückgegebene Wert ein `String` ist. Generell können nicht Objekte beliebiger Klassen übergeben werden. Voraussetzung ist, dass diese *serialisierbar* sind. Serialisierbarkeit kennzeichnet die Fähigkeit eines Objekts, in einen Bytestrom überführt werden zu können. Dabei muss die Serialisierbarkeit transitiv für alle in diesem Objekt benutzten Klassen gelten. Serialisierbarkeit wird in Java durch Erben der Schnittstelle `Serializable` erreicht. Die wichtigsten Standardklassen (wie z. B. `String`) implementieren bereits diese Schnittstelle.

Für die Kommunikation zwischen Client und Server sind die entsprechenden Stub- und Skeletonklassen notwendig. Diese Klassen werden mit einem RMI-Compiler, der Teil des Java Development Kit ist, automatisch erzeugt. In diesem Fall erhalten die Klassen die Namen `MDMObjectImpl_Skel` und `MDMObjectImpl_Stub`. Die beiden Klassen erben ebenfalls das Interface `MDMObject`.

Auf der Serverseite muss der Registry Service aktiv sein. Um das Serverobjekt den Clientanwendungen zur Verfügung zu stellen, ist die Anmeldung in der Registry notwendig (mittels `Naming.bind()`). Die Lebenszeit des Serverobjekts ist auf die Lebenszeit des ausführenden Prozesses beschränkt. Aus diesem Grund folgt in unserem Beispiel auf die Initialisierung des Objekts eine Endlosschleife.

```
public class MDMServer
{
      public static void main (String[] args){
        try {
                // Objekt wird im Server erzeugt
                MDMObjectImpl server_obj = new MDMObjectImpl();
                // und für den entfernten Zugriff über die Registry
                // mit dem Namen myObject zur Verfügung gestellt
                Naming.bind("myObject",server_obj);
        }
        catch (Exception e ) {
                ...
        }
        // Warteschleife um Beenden zu verhindern
        while (true) {
               sleep(1000);
       }
     }
}
```

Auf Clientseite kann nun eine Anwendung implementiert werden, die auf die entfernten Methoden zugreift. Diese Anwendung holt sich über die Registry des Servers zunächst eine Referenz auf das serverseitige Objekt (`Naming.lookup()`), wodurch es dieses Objekt in transparenter Weise nutzen kann.

```
 public class myClient
{
      public static void main (String[] args){
         try {
                 // Referenz zum Serverobjekt holen
                 MDMObject object = (MDMObject)
                      Naming.lookup("www.medienarchiv/myObject");
                 // und Aufrufen der entsprechenden Methode
```

```
                System.out.println(object.getName());
        }
        catch ( java.rmi.RemoteException e ) {
                ...
        }
    }
}
```

Der allgemeine Zusammenhang zwischen den einzelnen Klassen und Schnittstellen, ist der Abbildung 5.5 zu entnehmen.

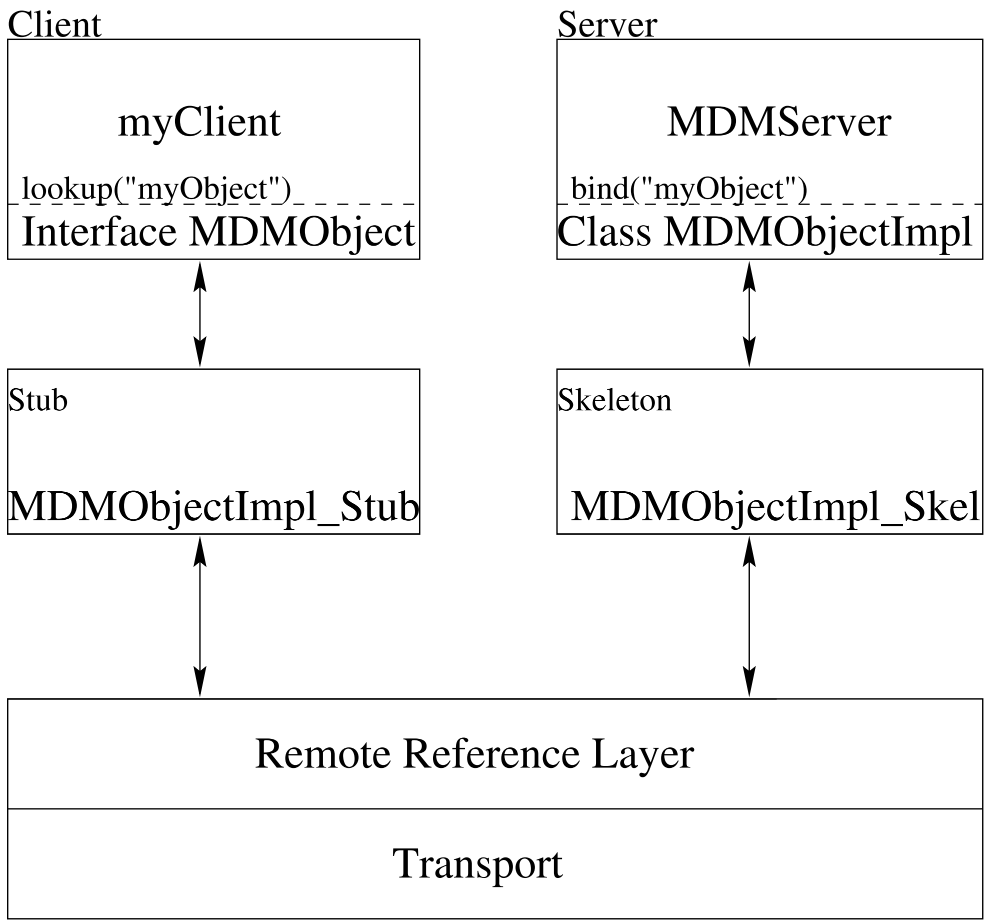


Abbildung 5.5: Zusammenspiel der Klassen bei RMI

Zusammenfassung und Verwendung im Archiv:

Das Java RMI-System bietet die Möglichkeit, eng gekoppelte, verteilte Systeme (Server/Client-Applikationsstrukturen) innerhalb der Java-Umgebung zu realisieren. Das verteilte Objektmodell und die API des Java RMI fügen sich nahtlos in die Java-Architektur ein. Das Java-Objektmodell wird durch RMI erweitert und das Handling der Remote-Objekte ist analog zu lokalen Java-Objekten. Damit ist die Einführung in einem Java-Umfeld vereinfacht möglich und mit geringem Einführungsaufwand behaftet.

Aufgrund dieser Vorteile wurde Java RMI als Basis für die MDM-Serverimplementierung gewählt. Die MDM-API realisiert einen für den Benutzer transparenten Zugriff auf die serverseitigen Dienste.

Nachteilig bleibt zu erwähnen, dass RMI unter den grundsätzlichen Performance-Problemen der Java-Plattform leidet. Mittlerweile gibt es jedoch Implementierungen von Drittanbietern, die deutlich bessere Laufzeiteigenschaften aufweisen. Die propriäteren Protokolle bedeuten bei tiefergehenden Optimierungsverfahren einen zusätzlichen Einarbeitungsaufwand. Hier bleibt zu hoffen, dass der Hersteller SUN möglichst bald die angesprochenen Standardprotokolle integriert. Eine weitere Fehlerquelle ist die Garbage Collection. Aufgrund der Verteilung der Objekte über mehrere Hosts sind hier in der Regel vom Entwickler zu realisierende Strategien zur Speicherfreigabe notwendig.

5.3.4 Komponenten und Java Beans

Komponentensoftware verfolgt das Ziel, Programmfragmente, weit über den objektorientierten Ansatz hinaus, wiederverwendbar zu gestalten. Komponenten sind selbstenthaltende Elemente, welche dynamisch gesteuert und zu neuen Bausteinen verschmolzen werden können. Selbstenthaltend bedeutet dabei, dass eine Komponente alle für ihren Einsatz benötigten Bestandteile (also Klassen, Konfigurationsinformationen usw.) enthält. Komponenten müssen in der Lage sein, nach vordefinierten Regeln miteinander zu kooperieren.

Java Beans [Eng97] übertragen die Idee des komponentenbasierten Softwareentwurfs auf die Sprache Java. Mittels Java Beans wird es dem Benutzer somit ermöglicht, Applikationen aus fertigen Teilen zusammenzusetzen. Dieses kann sowohl durch Programmierung als auch durch graphische Werkzeuge geschehen. Bei der graphischen Vorgehensweise werden fehlende Programmteile (sogenannter „glue code“) von der Entwicklungsumgebung automatisch erzeugt.

Ein Java Bean kann aus einer einzelnen oder auch mehreren Klassen bestehen. Für die Erstellung dieser Java-Klassen ist prinzipiell keine Erweiterung der JVM nötig. Java Beans müssen weder bestimmte Klassen erben noch vorgegebene Schnittstellen implementieren. Sie müssen vielmehr einigen Konventionen und Charakteristiken genügen, um sicherzustellen, dass sie innerhalb einer Komponentenarchitektur anwendbar sind. Diese im Folgenden aufgeführten Konventionen machen letztendlich das Konzept der Beans aus.

Eigenschaften und Methoden:

Eigenschaften sind ausschließlich über Methoden ansprechbare Attribute eines Beans. Eine Eigenschaft repräsentiert einen Wert. Der Zugriff erfolgt über ein Paar von `get/set`-Methoden, deren weitere Namen sich von der entsprechenden Eigenschaft ableiten. Für

Werte von Typ `boolean` wird statt einer `get`-Methode eine `is`-Methode verwendet. Eigenschaften folgen somit nachfolgender Namenskonvention:

```
public <PROPERTY_TYPE> get<PROPERTY_NAME>()
public boolean is<PROPERTY_NAME>()
public void set<PROPERTY_NAME>(<PROPERTY_TYPE> value)
```

Ereignisse und Aktionen:

Java Beans kommunizieren über ein *„Event-Action“*-Paradigma. Verändert sich der Zustand eines Java Beans, tritt also ein Ereignis (Event) ein, dann besteht die Möglichkeit einer Benachrichtigung anderer Instanzen durch Methodenaufrufe. Die Versendung der Ereignisse geschieht an registrierte Komponenten. Diese Komponenten reagieren nach vordefinierten Regeln auf das Eintreffen eines Ereignisses (Action). Sowohl die Methoden zum Registrieren als auch die Ereignisse selbst unterliegen den Java Beans-Namenskonventionen. Durch solche Ereignisse wird es möglich, verschiedene Bausteine innerhalb einer visuellen Entwicklungsumgebung miteinander zu koppeln.

Selbstprüfung :

Der Begriff Selbstprüfung (*Introspection*) kennzeichnet die Untersuchung einer Java-Klasse zur Laufzeit. Hiermit können die Methoden und Eigenschaften eines Beans dynamisch ermittelt werden. Dieses ist notwendig, um ein im Bytecode vorliegendes Bean in einer graphischen Entwicklungsumgebung einzusetzen.

Persistenz :

Java Beans müssen die persistente Speicherung ihres Zustands und dessen spätere Rekonstruktion unterstützen. Die Speicherung kann mittels unterschiedlicher Verfahren realisiert werden, z. B. der Java Object Serialization (s. Abschnitt 5.3.3) oder der Ablage in einer Datenbank.

Beispiel:

Die folgende Klasse `AnmeldeBean` ist ein Beispiel für ein einfaches Bean mit drei Eigenschaften (`user`, `pass`, `connected`). Dabei ist die Eigenschaft `pass` nur schreibbar und die Eigenschaft `connected` nur lesbar. Dieses Bean soll sich durch die Übermittlung von Benutzerinformationen an eine Datenbank (z. B. an den Medienarchivserver) anmelden.

Mithilfe der Methode `setLoggedListener()` kann sich ein fremdes Objekt für das Empfangen von Nachrichten registrieren. In diesem Fall muss es ein Objekt der Klasse `LoggedListener` sein, welches benachrichtigt wird (`connectionEstablished()`), sobald eine Datenbankverbindung aufgebaut wurde.

Geht man davon aus, dass die Klasse `LoggedListener` serialisierbar ist, so sind für die Gewährleistung der Persistenzfähigkeit keine weiteren Methoden notwendig, da die verwendeten Objekte allesamt serialisierbar sind.

```
public class AnmeldeBean{
  private String user;
  private String pass;
  private boolean connected;
  private LoggedListener log;

  // Lesen der Eigenschaft 'user'
  public String getUser(){
    return user;
  }

  // Schreiben der Eigenschaft 'user'
  public void setUser(String user){
    this.user=user;
  }

  // Schreiben der Eigenschaft 'pass'
  public String setPass(String pass){
    this.pass=pass;
    // DB-Connection erzeugen
    connected=montana_server.connect(user,pass);
    if(connected) log.connectionEstablished(this);
  }

  // Gibt Auskunft, ob Anmeldung erfolgreich war
  public boolean isConntected() {
     return connected;
  }

  // Methode zum Registrieren einer fremden Komponente
  // um dieser beim Eintreten eines Ereignisses eine
  // Nachricht zu senden
  public void setLoggedListener(LoggedListener log){
    this.log=log;
  }
}
```

5.3.5 Enterprise Java Beans

Enterprise Java Beans (EJB) [Java, MH00] liefern eine hochskalierbare, verteile und plattformunabhängige Architektur, welche die effiziente Entwicklung unternehmensweiter Anwendungen unter der Verwendung bestehender Ressourcen unterstützt.

Das EJB-Konzept zielt vordergründig auf den Einsatz in unternehmensweiten Anwendungen hin. Daher fallen in diesem Zusammenhang des öfteren Begriffe aus dem betriebswirtschaftlichen Umfeld. Die EJB-Architektur ist jedoch so allgemein, dass sie nicht auf diesen Anwendungsfall beschränkt ist.

Durch die Benutzung von EJB ist der Entwickler in der Lage, Geschäftsobjekte zu entwerfen und Geschäftsprozesse abzubilden, wobei diese in einer Vielzahl von Plattformen, Middlewaresystemen und Applikationsservern eingesetzt werden können. Die EJB-Technologie ermöglicht somit die Entwicklung von portierbaren eBusiness-Anwendungen, welche über unterschiedliche Server verteilt sein können. Aus Entwicklersicht erweitern EJB dabei den verteilten Objektzugriff von RMI.

Da Enterprise Java Beans ein sehr komplexes Gebiet darstellen, soll hier lediglich auf die wichtigsten Grundlagen eingegangen werden. Für tiefergehende Betrachtungen sei auf die genannte Literatur verwiesen.

EJB sind keine Erweiterung der Java Beans. Daher gelten nicht zwangsläufig die im vorangegangenen Abschnitt beschriebenen Konventionen.Vielmehr übertragen EJB die Komponentenidee auf eine verteilte Client/Server-Architektur. Wenn in diesem Abschnitt von Beans gesprochen wird, sind daher stets die Enterprise Java Beans gemeint.

EJB sind serverseitige Komponenten, die in einer Laufzeitumgebung (dem Container) zum Einsatz kommen. EJB werden in *Entity Beans* und *Session Beans* unterteilt. Entity Beans sind persistente Geschäftsobjekte, die in einer Datenbank gespeichert werden können. Sie sind über die Laufzeit eines Containers hinaus lebensfähig. Session Beans sind dagegen Objekte, die Geschäftsabläufe repräsentieren. Sie hängen im Wesentlichen vom Lebenszyklus eines Client ab und sind auf die Dauer einer Session beschränkt.

Die wesentlichen Ablaufumgebungen der EJB-Architektur sind Client, Server und Container.

EJB-Server:

Ein *EJB-Server* ist eine spezielle Form eines Applikationsservers. Ein Applikationsserver ermöglicht dem Entwickler die Aufteilung von Geschäftslogik und Präsentation im Rahmen eines Softwareprojekts. Im Unterschied zu einer Client/Server-Lösung wird durch die Verwendung eines Applikationsservers ein mehrschichtiger Ansatz gewählt. Der Applikationsserver dient in diesem Zusammenhang als „Middleware“. Viele Applikationsserver liefern darüber hinaus zusätzliche Dienste wie Transaktionsmanagement oder Lastverteilung. Eine Konsequenz der Migration der Geschäftslogik auf die Serverseite ist die Verkleinerung der Clientanwendungen. Man spricht vom Übergang der „Fat Clients“ zu den „Thin Clients“. Ein weiterer großer Vorteil solcher „Multi-Tier”-Umgebungen ist die Unabhängigkeit des Client vom Server. Änderungen können weitgehend unabhängig voneinander realisiert werden.

EJB-Server werden (neben der Referenzimplementierung der Firma SUN) von unterschiedlichen Herstellern angeboten. Ein EJB-Server kann mehrere Container beinhalten, welche als Laufzeitumgebung für EJB-Objekte dienen. Die Schnittstelle zwischen dem Server und den Containern ist in der EJB-Spezifikation ([Java]) nicht eindeutig festgelegt. Daher implementieren die Hersteller eigene Schnittstellen, so dass der EJB-Container eines Herstellers nicht unbedingt in einem Server eines anderen Herstellers betrieben werden kann.

Ein EJB-Server erlaubt den erweiterten und transparenten Zugriff auf Dienste wie Sicherheit, Resourcenmanagement, Concurrency, Transaktionen und Persistenz. Insbesondere bietet ein EJB-Server grundlegende Funktionalitäten eines *Component Transaction Monitor (CTM)* [CLMY93], wie

- Thread- und Prozessmanagement für mehrere Container, die parallel auf dem Server laufen können.
- Dienste im Bereich Clustering, Lastverteilung und Ausfallsicherheit.
- Management von Betriebssystemressourcen.

EJB-Client:

Ein *EJB-Client* benutzt entfernte Objekte im EJB-Server zur Ausführung seiner Aufgaben. Er holt sich dabei, ähnlich zu RMI, Referenzen auf bestehende und neu zu erzeugende Objekte im Server. Diese Anforderung von Objekten geschieht über das bereits erwähnte Java Naming and Directory Interface (JNDI).

EJB-Container:

In der EJB-Umgebung gibt es neben Client und Server den angesprochenen *Container*. Dieser ist für die Implementierung und Generierung aller zusätzlichen, von der EJB-Umgebung unterstützten Dienste verantwortlich. Der EJB-Container ist eine Laufzeitumgebung für EJB-Komponenten, wobei ein EJB-Server mehrere EJB-Container besitzen kann und ein EJB-Container wiederum beliebig viele Objekte beinhalten kann.

Bestandteile der Enterprise Java Beans:

Bei der Entwicklung eines Enterprise Java Bean werden verschiedene Klassen bzw. Schnittstellen für den Einsatz in einem EJB-Container benötigt. Ein Bean besteht aus folgenden Teilen:

- *Remote Interface*: Das Remote Interface definiert die „Geschäftsmethoden" eines Bean, also die Methoden die nach außen hin sichtbar sind. Es erbt dabei von der Schnittstelle `javax.ejb.EJBObject`, welche ihrerseits auf `java.rmi.Remote` aufsetzt.
- *Home Interface*: Während Remote Interface und die spätere Bean-Implementierung ähnlich zu den entsprechenden RMI-Komponenten sind, ist das Home Interface für Lebenszyklusaufgaben verantwortlich. Hierzu zählen die Erzeugung, die Lokalisierung und das Löschen eines EJB-Objekts. Damit geht es über RMI hinaus, wo ein Client lediglich Verbindung zu einem existierenden Objekt aufnimmt.
- *Bean Class*: Die Bean Class implementiert die Geschäftsmethoden des Bean. In der Regel erbt es jedoch weder Remote Interface noch Home Interface. Es muss lediglich (alle) Methoden besitzen, die den Signaturen der Methoden des Remote Interface entsprechen und (einige) andere Methoden für die es Entsprechungen im Home Interface gibt. Entity Bean und Session Bean implementieren die Schnittstellen `javax.ejb.EntityBean` bzw. `javax.ejb.SessionBean`, wobei diese beiden Schnittstellen von `javax.ejb.EnterpriseBean` abgeleitet sind.

- *EJB Home und EJB Object*: EJB Home und EJB Object sind zwei Klassen, die (genauso wie die zugehörigen Stubs) automatisch vom Container erzeugt werden. Das EJB Object ist ein auf serverseite verteiltes Objekt, welches das Remote Interface implementiert und zusätzlich `javax.ejb.EJBObject` erbt. Das EJB Home implementiert alle im Home Interface definierten Methoden. Das EJB Home arbeitet eng mit dem Container zusammen. Es besitzt Methoden zum Erzeugen und Suchen von Enterprise Beans (z. B. `ejbCreate()`).

- *Deployment Descriptor*: Der Deployment Descriptor definiert die Verwendung eines oder mehrerer EJB-Komponenten. So werden das Verhalten innerhalb des Containers und die Verwendung höherwertiger Dienste (wie z. B. Transaktionsmanagement) näher spezifiziert. Der Deployment Descriptor wird für die automatische Erzeugung der nicht vom Benutzer erstellten Bean-Teile benutzt. Dadurch ist eine explizite Änderung der Implementierung durch den Benutzer zwecks etwaiger Anpassungen in der Regel nicht notwendig.

- *Primärschlüssel*: Der Primärschlüssel ist ein zwingender Bestandteil von Entity Beans. Der Primärschlüssel dient als Identifikator für die Ablage in einer Datenbank. Da Session Beans definitionsgemäß nicht persistenzfähig sein müssen, besitzen diese keinen Primärschlüssel.

Abbildung 5.6 zeigt das Zusammenspiel der einzelnen Teile eines Enterprise Java Bean. Während die zu implementierenden Teile weiß sind, sind die vom EJB-Container generierten Klassen schraffiert.

Ein Hauptgrund für das allgemein schwierige Verständnis der EJB-Architektur entstammt der Tatsache, dass Klassen oftmals Interface-Methoden, nicht aber das Interface selbst implementieren. Der Grund dafür ist, dass Enterprise Java Beans praktisch in der Mitte, zwischen Clientanwendung und Datenbank untergebracht sind. Der Client kommuniziert niemals direkt mit einem Bean; er benutzt stets dessen Remote und Home Interface. Genauso ruft ein Bean nicht Container-Methoden auf, sondern wird ausschließlich vom Container angesprochen. Anstatt der starren Vererbung wird die dynamische „Introspection“, also die Bestimmung von Methoden zur Laufzeit genutzt.

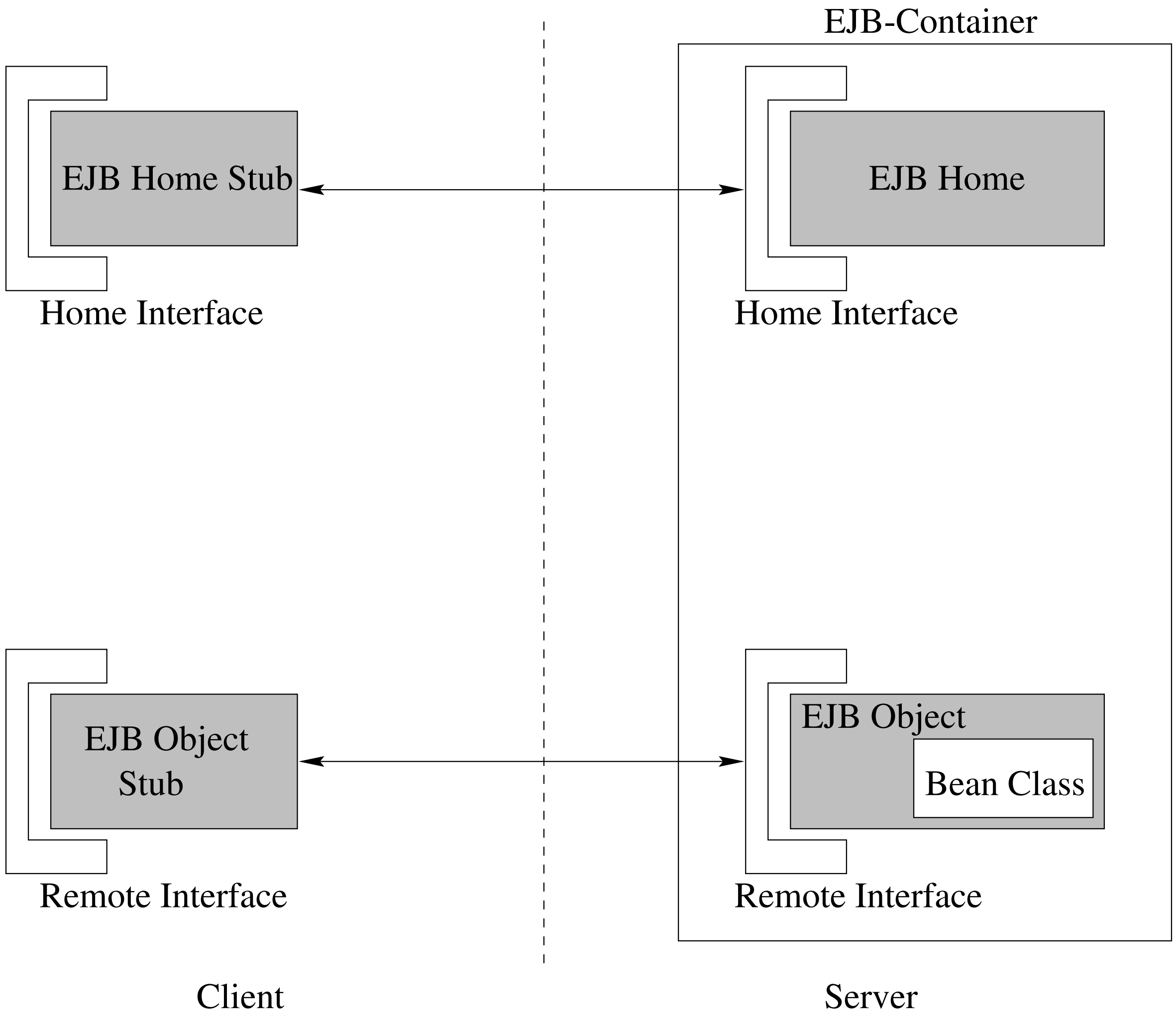


Abbildung 5.6: Zusammenspiel der Teile eines Enterprise Java Bean

Beispiel für die Nutzung im Archiv:

Die mögliche Verwendung von Enterprise Java Beans im Medienarchiv soll anhand des schon im RMI-Beispiel verwendeten `MDMObject` veranschaulicht werden. Diese Klasse wird hier auf die EJB-Architektur übertragen.

Bei der Definition der einzelnen Teile werden Standard-EJB-Klassen (Package `javax.ejb`) geerbt. Diese Klassen implementieren höherwertige EJB-Dienste, die so automatisch zur Verfügung stehen.

Da es sich bei einem `MDMObject` um ein real innerhalb des Medienarchivs existierendes Objekt handelt, wird es als Entity Bean umgesetzt. Bei der Entwicklung eines Entity Bean müssen wir zuerst dessen Remote Interface festlegen. Das Remote Interface definiert die Geschäftsmethoden eines Bean, welche nach außenhin sichtbar sind. In unserem Fall ist über ein Methodenpaar die Eigenschaft `name` ansprechbar.

```
public interface MDMObject extends javax.ejb.EJBObject //Remote Interface
{
    ...
    // Definition der Methoden, die
    // später 'remote' genutzt werden sollen
    public String getName() throws java.rmi.RemoteException;
    ...
    public void setName(String name) throws java.rmi.RemoteException;
    ...

}
```

Das Interface erbt von `javax.ejb.EJBObject`, welches seinerseits von `java.rmi.Remote` abgeleitet ist.

Nach der Festlegung des Remote Interface müssen wir über das Home Interface die Lebenszyklus-Methoden spezifizieren. Die Klasse, welche das Home Interface später implementiert, wird entsprechend als EJB Home bezeichnet. Diese Klasse wird durch den Container automatisch erzeugt.

```
public interface MDMObjectHome extends javax.ejb.EJBHome //Home Interface
{
    ...
    // Erzegung von Objekten
    public MDMObject create(int)
              throws java.rmi.RemoteException;
    ...
    // Auffinden eines Objekt anhand eines Schlüssels
    public MDMObject findByPrimaryKey(MDMObjectPK)
              throws java.rmi.RemoteException;
    ...

}
```

Die Klasse `MDMObjectHome` erbt die Klasse `javax.ejb.EJBHome` und definiert zwei Lebenszyklus-Methoden zum Erzeugen und Lokalisieren von Objekten.

Da Entity Beans serverseitig in einer Datenbank abgelegt werden, müssen sie (im Gegensatz zu Session Beans) einen serialisierbaren Primärschlüssel besitzen. Daher führen wir eine Klasse `MDMObjectPK` ein.

```
public class MDMObjectPK implements Serializable // Primärschlüssel
{
    // Schlüsselwert
    int id;
```

```
    // Konstruktor
    public MDMObjectPK(int id){
      this.id=id;
    }
}
```

Die Klasse `MDMObjectPK` enthält ein Attribut `id`. Dieses Attribut dient zur eindeutigen Lokalisierung eines `MDMObject` zur Laufzeit.

Nachdem alle clientseitigen Schnittstellen definiert wurden, benötigen wir die zugehörige Implementierung der Bean Class (`MDMObjectBean`).

```
public class MDMObjectBean implements javax.ejb.EntityBean // Entity Bean
{
    ...
    int id;
    String name;

    public void ejbCreate (int id){
     this.id=id;
    }

    public String getName() {
     return name;
    }

    public void setName(String name) {
      this.name=name;
    }
    ...
    public void ejbLoad() {
     nicht implementiert!
    }
    public void ejbStore() {
     nicht implementiert!
    }
    ...
}
```

Die `get`- und `set`-Methoden sind die Geschäftsmethoden des Bean. Sie sind die einzigen Methoden, die für einen Client sichtbar sind; alle anderen Methoden sind nur für den Container erreichbar. Die Methode `ejbCreate()` dient der Erzeugung eines neuen Bean. Die Methoden `ejbLoad()` und `ejbStore()` sind für das Laden und Speichern des Bean in einer Datenbank verantwortlich.

Schließlich ist ein Deployment Descriptor erforderlich, der die wesentlichen Eigenschaften des Bean spezifiziert und der serialisierbar ist. Während der Installation eines Bean wird der

Deployment Descriptor deserialisiert und die Eigenschaften können entsprechend verändert werden. Danach wird von der Laufzeitumgebung unter Benutzung dieser Informationen die fehlende Infrastruktur (d. h. die notwendigen Klassen) automatisch erzeugt.

Die meisten EJB-Umgebungen erzeugen den Deployment Descriptor auf der Basis einer XML-Konfigurationsdatei. Da wir in unserem Beispiel Dinge wie Transaktionsmanagement außer Betracht lassen wollen (obwohl dieses natürlich einer der wesentlichen Vorteile des EJB-Modells ist), werden wir hier auf den Deployment Descriptor nicht weiter eingehen.

Für die Verwendung der entwickelten Teile werden diese zu einem Java Archiv (JAR-File) zusammengefasst und in einem EJB-Server installiert. Diese Installation wird auch *Deployment* genannt. Hierbei erzeugt der Container die angesprochenen fehlenden Teile.

Eine Clientapplikation, die das obige Entity Bean benutzt, kann wie folgt aussehen:

```
...
   static void main(String[] args) {
      try{
       // Aktivierung des JNDI-Context
       Context jndiContext = getInitialContext();

       // Bestehendes Objekt vom Server holen
       MDMObjectHome home= (MDMObjectHome)
                 jndiContext.lookup("MDMObjectHome");

       // Neues Objekt erzeugen
       MDMObject object_1 = home.create(1);
       object_1.setName("Ein Objekt");

       // Objekt anhand des Schlüssels suchen
       MDMObjectPK key = new MDMObjectPK(1);
       MDMObject object_2 = home.findByPrimaryKey(key);


     } catch (Exception e) {...}
   }
...
```

Um ein Enterprise Java Bean zu erhalten, muss ein Client zunächst eine Verbindung (`Context`) zum JNDI-Nameserver aufbauen. Mittels der Verbindung kann dann das Home Interface des Bean angesprochen werden. Damit ist es möglich, ein Enterprise Java Bean zu erzeugen (`create()`) oder auch ein bestehendes Bean zu erhalten (`findByPrimaryKey()`).

Wie wir sehen können, ist durch die EJB-Realisierung von `MDMObject` eine höhere Benutzungstransparenz und Unabhängigkeit der Klassen gegeben. Die Erzeugung und Lokalisierung von Objekten wird durch den Server gekapselt. Auch ist die persistente Ablage genauso wie die Steuerung des Transaktionsverhaltens bereits implizit möglich. Die Änderung der serverseitigen Implementierung wirkt sich nicht zwangsläufig auf die definierten Schnittstellen aus. Ein Enterprise Bean ist auf jedem EJB-fähigen Server, im Idealfall ohne spezielle Anpassungen, einsetzbar.

Zusammenfassung und Verwendung von Java Beans und EJB im Archiv:

Soweit dieses realisierbar war, wurden Java Beans-Konventionen bei der Implementierung von MDM-Klassen eingehalten. Intensivere Nutzung erfahren Java Beans bei der Umsetzung von logischen Medienarchivinhalten in Webseiten, wie sie im folgenden Abschnitt beschrieben wird.

EJB kommen in der aktuellen Implementierung des Medienarchivs noch nicht zum Einsatz. Bei der zukünftigen Serverimplementierung ist aber an eine Umsetzung mittels EJB gedacht, um so die Nutzung der CTM-Dienste zu ermöglichen und eine stärkere Skalierbarkeit und Wiederverwendbarkeit zu gewährleisten. Da der EJB-Standard noch relativ neu ist, bleibt zu verfolgen, wie sich diese Technik auf dem Markt etabliert und in welche Richtung sich zukünftige Versionen entwickeln werden.

5.4 Web-Anbindung des Archivs

Anfang der 90'er Jahre begann sich das *World Wide Web* (kurz Web) als zentraler Internet-Dienst zu etablieren. Das Web basiert auf einer Client/Server-Architektur. Der Webserver stellt über einen Standard-Port Objekte (in der Regel HTML-Seiten) zur Verfügung. Ein Client (Browser) kann diese Seiten abrufen und für den Benutzer darstellen. Die Kommunikation geschieht über das gedächtnislose *Hypertext Transfer Protocol (HTTP)*. Gedächtnislosigkeit bedeutet hierbei, dass für jede Übertragung eines Objekts eine neue Verbindung zwischen Client und Server aufgebaut werden muss.

Webbrowser :

Zur Darstellung von Webinhalten auf dem Clientrechner dienen Browser (*z. B. Netscape Communicator, Internet Explorer*). Browser sind mittlerweile sehr mächtige Anwendungen geworden. Neben der Darstellung einfacher HTML-Seiten unterstützen sie *Java Script*, *Cascading Style Sheets* und *DHTML* (s. u.), um damit aktive Seiteninhalte zu präsentieren. Browser sind daneben in der Lage, Applikationen im Kontext eines Dokuments ablaufen zu lassen. Solche Applikationen können *Java Applets* oder *ActiveX*-Komponenten sein. Schließlich können Browser durch die Installation von *Plugin*s um beliebige Funktionalitäten erweitert werden.

Webserver :

Die ersten über Webserver angebotenen Seiten waren statisch. Zur Erzeugung dynamischen Inhalts wurden proprietäre Lösungen eingesetzt.

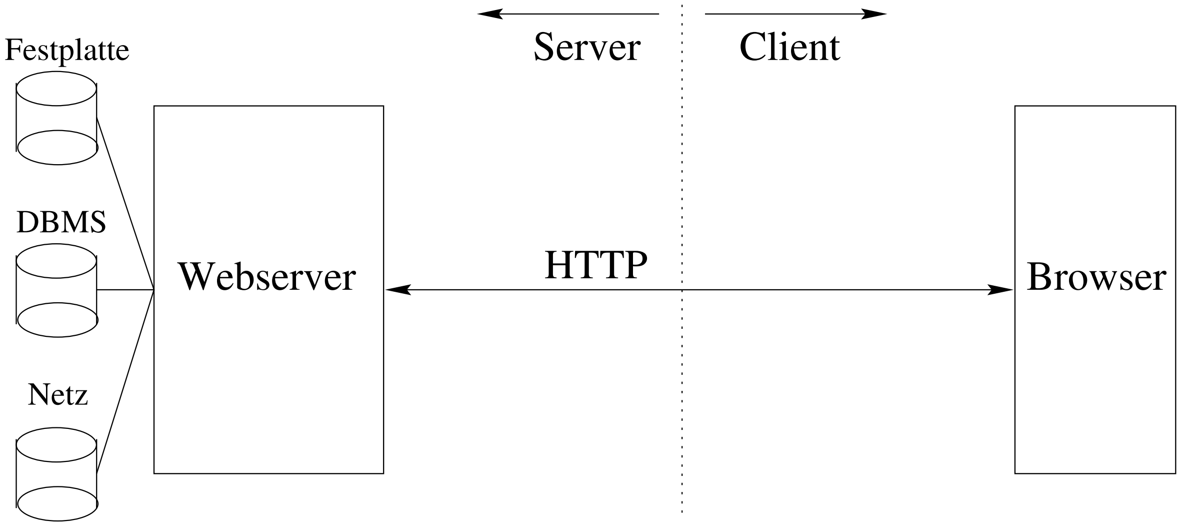


Abbildung 5.7: Architektur des Web

Um diesen Mißstand zu beseitigen, wurde das *Common Gateway Interface (CGI)* entworfen. CGI stellt eine standardisierte Schnittstelle für den Aufruf von Prozessen durch den Webserver dar. Lange Zeit diente die CGI-Schnittstelle als zentrales Werkzeug für die Erstellung dynamischer Webinhalte. Doch gerade bei größeren Anwendungen zeigt CGI gravierende Nachteile:

- Für jede Webanfrage wird ein neuer Prozess auf dem Server gestartet, was in der Regel sehr zeitaufwendig ist.
- Werden von CGI-Prozessen Datenbankzugriffe benötigt, so ist bei jedem Aufruf die Erstellung einer neuen Verbindung notwendig.
- Zur Verarbeitung von Webzugriffen sind spezielle Methoden notwendig, die in den gängigen Programmiersprachen nicht vorhanden sind und jeweils neu implementiert werden müssen.
- CGI-Programme können in vielen verschiedenen Sprachen implementiert werden. Durch diese Vielfalt wird jedoch eine Umsetzung auf andere Plattformen sehr aufwendig.

Die Hersteller von Webservern entwickelten APIs, um die Performance-Nachteile und eingeschränkten Möglichkeiten von CGI zu überwinden. Diese APIs waren jedoch an die jeweiligen Webserver gebunden und somit nicht kompatibel. Applikationen, die diese APIs verwendeten, waren schwierig zu implementieren und im allgemeinen nicht portierbar. Außerdem liefen so erstellte Applikationen im Adreßraum des Webservers ab, was dazu führen konnte, dass fehlerhafte Applikationen den Webserver zum Absturz brachten.

5.4.1 Java Servlets

Java Servlets bringen die Funktionalität der Sprache Java in den Webserver ein. Die Java Servlet API wurde entwickelt, um mittels der Vorteile der Java Plattform die durch CGI und propriätere APIs aufgeworfenen Probleme zu lösen. Die API ist vergleichsweise einfach gehalten und wird derweil von allen gängigen Webserveranbietern direkt oder mittels Erweiterungen unterstützt.

Abbildung 5.8 zeigt den Ablauf einer HTTP-Anfrage, die von einem Servlet verarbeitet und beantwortet wird. Eine herkömmliche HTTP-Anfrage wird vom Webserver entgegen genommen und an die Servlet-Engine weitergeleitet. Der Webserver erkennt dabei anhand der URL, ob es sich um einen Servletaufruf handelt oder um eine direkt durch den Webserver zu beantwortende Anfrage. Die Servlet-Engine übernimmt die Parameter der Anfrage und formt sie in Java-Parameter um, welche anschließend der Servicemethode des Servlet (`HTTPServletService()`) übergeben werden.

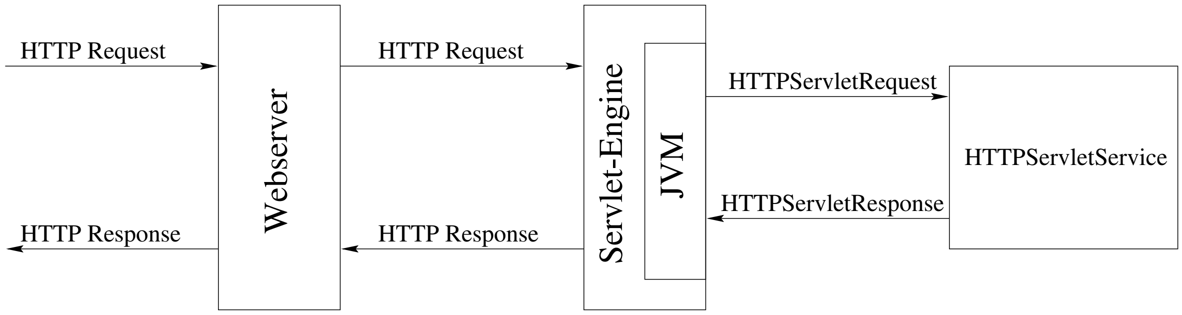


Abbildung 5.8: HTTP-Zugriff auf Servlets

Vergleicht man Java Servlets mit herkömmlichen Lösungen, so sind sie effizienter, leichter zu benutzen, mächtiger, portierbarer, kostengünstiger und sicherer.

Effizienz:

Bei herkömmlichen CGI-Skripten wird mit jedem HTTP-Zugriff ein neuer Prozess auf dem Server erzeugt. Wenn durch ein CGI-Skript lediglich eine kleine Operation ausgeführt wird, ist der Overhead für die Prozesserzeugung im Verhältnis sehr groß. Bei Servlets läuft eine JVM permanent mit dem Webserver zusammen, wodurch für einen Webzugriff nur ein leichtgewichtiger Java-Thread erzeugt werden muss. Wenn ein CGI-Programm n-Mal aufgerufen wird, wird der CGI-Code auch n-Mal in den Speicher geladen. Dagegen werden in der Servlet-Engine n Threads erzeugt, wobei es jedoch nur eine einzige Kopie der Servlet-Klasse gibt. Servlets besitzen weitergehende Möglichkeiten für Optimierungen, wie Caching oder permanente Datenbankverbindungen.

Komfort:

Die Servlet-API bietet zahlreiche Funktionen, u. a. zum automatischen Parsieren und Dekodieren von HTTP-Anfragen, Lesen und Schreiben von HTML-Headern, Verwendung von *Cookies* (zur Speicherung kleiner Informationspakete im Browser) und Verarbeiten von *Sessions* (aufeinanderfolgende Seitenaufrufe).

Mächtigkeit:

Mit Java Servlets lassen sich Problemlösungen finden, die mit CGI schwierig oder gar unmöglich wären. So können Servlets direkt mit dem Webserver kommunizieren oder Daten ohne Umwege zwischen einzelnen Servlets austauschen.

Portierbarkeit:

Da Servlets reine Java-Anwendungen sind, können sie auf jedem Java-fähigen Webserver ablaufen. Dadurch wird eine hohe Portierbarkeit erreicht.

Kostenfaktor:

Mit wenigen Ausnahmen (z. B. *Apache-Webserver*) sind Webserver für den Einsatz in datenintensiven Anwendungen teuer. Dagegen entfallen aufgrund der freien Verfügbarkeit der JVM weitere Kosten für Erweiterung eines Webservers um Servlet-Unterstützung.

Sicherheit:

Werden Skriptsprachen für die Verarbeitung von Webanfragen verwendet, so werden auszuführende Betriebssystembefehle oftmals während der Laufzeit aus übergebenen Parametern zusammengesetzt. Hierdurch ist es schnell möglich, dass Hintertüren für unerwünschte Zugriffe unbefugter Personen entstehen. Da die Java-Umgebung diese Betriebssystemoperationen über Methoden direkt unterstützt, werden solche Sicherheitslücken vermieden.

Der folgende Beispiel-Code zeigt den grundsätzlichen Aufbau eines Servlets:

```
import javax.servlet.*;                   // Einbindung der Servlet API
import javax.servlet.http.*;

public class SomeServlet extends HttpServlet {

  public void init(Config conf) {
  // Beim Starten der Klasse einmalige Operationen
  ...
  }

  public void service(HttpServletRequest request,
                      HttpServletResponse response)
                      throws ServletException, IOException {

    // "request" dient zur Auswertung des HTTP-Headers
    // und evtl.  der HTML-Fomulardaten

    // "response" erlaubt die Erzeugung des Rückgabeobjekts
```

```
    PrintWriter out = response.getWriter();
    out.println("<html>...</html>");
  }

  public void destroy() {
  // Beim Entladen der Klasse zu vollziehende Aufgaben
  ...
  }

}
```

Durch Erben der Klasse `HttpServlet` erhält die Klasse `SomeServlet` alle für die Nutzung in einer Servlet-Engine notwendigen Methoden, so dass diese nicht explizit implementiert werden müssen. Die Methode `init()` wird beim Initialisieren des Servlets einmalig aufgerufen. Innerhalb dieser Methode kann zum Beispiel der Aufbau einer Datenbankverbindung untergebracht werden. Die Methode `service()` ist die eigentliche Dienstmethode. Sie wird beim Aufruf des Servlet durch den Client angesprochen, wobei sie die übergebenen Parameter der Webanfrage verarbeitet und eine Ausgabe erzeugt. Diese Ausgabe wird dann an den Browser zurückgeschickt. Die Methode `destroy()` wird vor dem Beenden der Servlet-Engine oder dem Entladen der Klasse aufgerufen.

5.4.2 Java Server Pages

Java Server Pages (JSP) sind eine Technologie, die es erlaubt, statische HTML-Seiten mit dynamisch generiertem HTML-Code anzureichern. Der Hauptvorteil von JSP gegenüber anderen Techniken wie Servlets oder CGI ist die strikte Trennung von Layout- und Business-Logik. So werden bei herkömmlichen Lösungen die Layout-Informationen ebenfalls im Programm-Code definiert (über einfache Ausgabeanweisungen). Diese Daten sind so schwer pfleg- und erweiterbar.

Bei JSP sind die Programm- und Layout-Teile in unterschiedlichen Bereichen untergebracht. Java Server Pages können als logische Schicht oberhalb der Servlets gesehen werden. Aus Systemsicht wird eine Java Server Page zur Laufzeit in ein Servlet umgesetzt.

Bei einem JSP-Programm handelt es sich um eine Mischung aus normalem Quellcode für Webseiten (HTML, XML, Java Script) und reinem Java-Code. Die Trennung der verschiedenen Sprachebenen erfolgt durch sogenannte *Taglets* (z. B. <%%>, <jsp:useBean>). Es gibt dabei unterschiedliche Typen von Taglets:

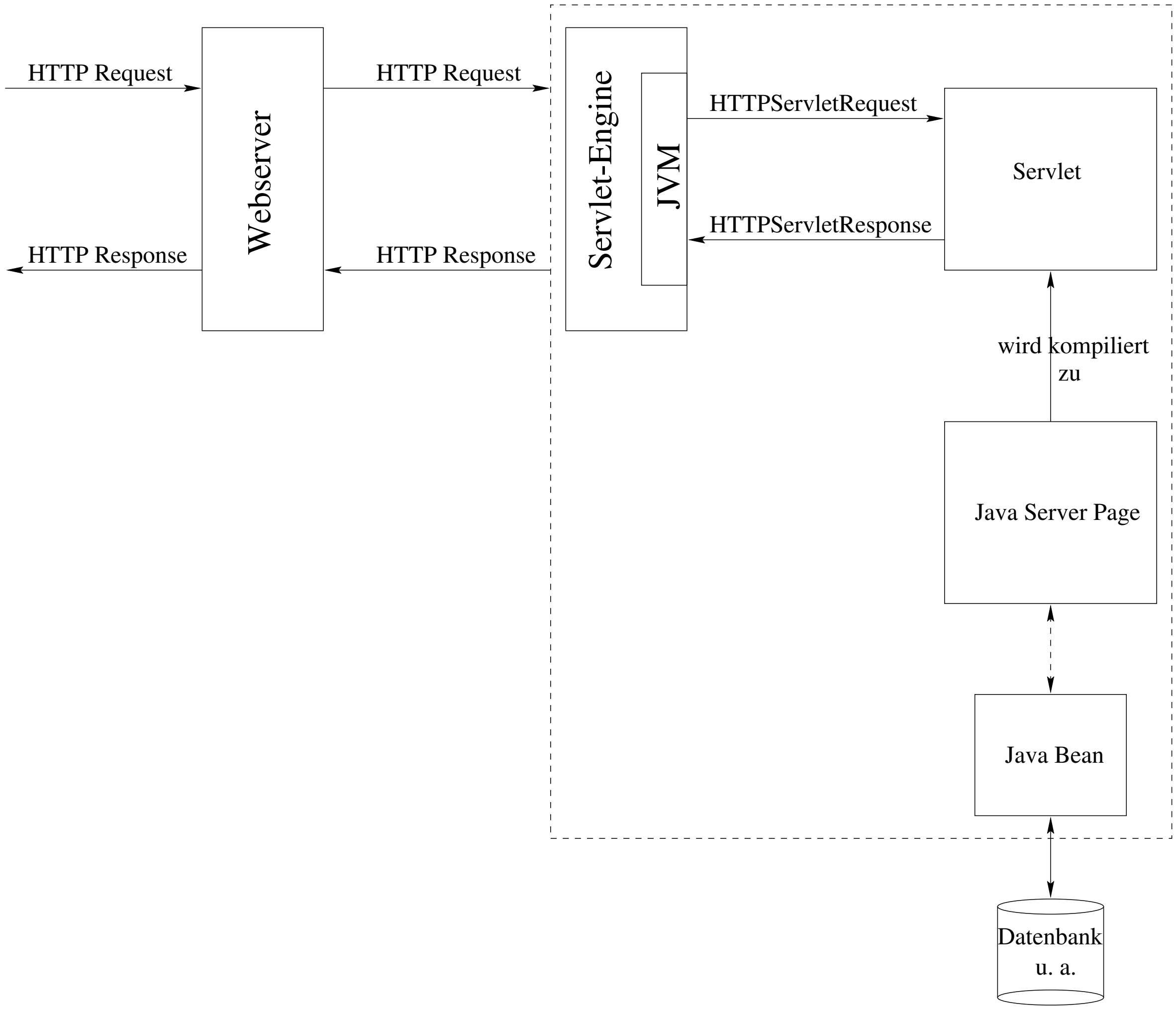


Abbildung 5.9: Webzugriff auf Java Server Pages

- **Skriptelemente:** Skriptelemente erlauben das Einfügen von Java-Code in das später erzeugte Servlet. Zur Vereinfachung der Skriptelemente erhält der Benutzer Zugriff zu einer Reihe vordefinierter Variablen (z. B. Ein-/Ausgabeströme, Java Beans).
- **Direktiven:** Direktiven dienen der Kontrolle über die Gesamtstruktur des Servlets. Hiermit werden zu importierende Teile festgelegt oder eine etwaige Session spezifiziert.
- **Deklarativen:** Mit Deklarativen können innerhalb der Seite geltende Variablen und Methoden definiert werden. Auch kann damit die Benutzung von Java Beans festgelegt werden.
- **Ausdrücke:** Ausdrücke dienen dem direkten Einfügen von (textuellen) Operationsergebnissen in die Ausgabeseite.

Wird eine JSP-Seite aufgerufen, überprüft die Servlet-Engine ob bereits ein Servlet aus dieser Seite erzeugt wurde. Ist dieses nicht der Fall, generiert sie aus dem Quellcode der JSP-Seite ein Servlet.

Bei größeren Applikationen wird in der Praxis meist eine Kombination von Servlets und Java Server Pages benutzt (s. Abbildung 5.9). Dabei werden JSP-Seiten für die Präsentationsgestaltung verwendet, während Servlets prozessintensive Aufgaben erledigen. Durch die besondere Unterstützung von Java Beans innerhalb von JSP-Seiten werden diese oft für die Anbindung an serverseitige Applikationen, wie Datenbanken oder EJB-Server genutzt.

Das folgende Beispiel zeigt eine JSP-Seite, die Anmeldedaten an das bereits oben vorgestellte Java Bean `AnmeldeBean` weitergibt.

```
<jsp:useBean id="anmeldeBean" class="AnmeldeBean" />
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD><TITLE>Willkommen im Medienarchiv MONTANA</TITLE></HEAD>
<BODY>
   <%
   ...
   // Wenn Seite zum ersten Mal aufgerufen wird,
   // wenn es also noch keine Übergabewerte aus
   // den Formularfeldern gibt, folgt die
   // Ausgabe des Einlogformulars
   if(kein_uebergabewert){
   %>
<H1>Geben Sie ihren Benutzernamen ein:</H1>
<FORM>
<INPUT TYPE=TEXT NAME=user />
<INPUT TYPE=PASSWORD NAME=pass />
</FORM>
   <%}
   // Wenn Eingabe-Daten übergeben wurden,
   // an Bean weiterleiten
   else {
   %>
   // Im zu erzeugenden Servlet werden aus <jsp:setProperty>
   // Java-Anweisungen der Form
   // anmeldeBean.setUser(request.getParameter("user"));

   <jsp:setProperty="anmeldeBean" property="user"/>
   <jsp:setProperty="anmeldeBean" property="pass"/>

   Sie sind angemeldet!
   <%
   }
   %>
</BODY>
</HTML>
```

Beim Aufruf der Java Server Page wird zunächst ein äquivalentes Servlet erzeugt. Die HTML-Anteile werden in entsprechende Ausgabeanweisungen umgesetzt. Zusätzlich einzufügender Java-Code ist von entsprechenden Klammern (<%, %>) eingeschlossen. Durch die erste Anweisung (`<jsp:useBean>`) wird ein Objekt der Klasse `AnmeldeBean` erzeugt und mit der Session in Verbindung gebracht. Das Objekt erhält dabei den Bezeichner `anmeldeBean`. Beim Aufruf der Seite wird unterschieden (`if(kein_uebergabewert)`), ob die Seite vom Benutzer erstmalig aufgerufen wurde (Ausgabe des Anmelde-Formulars) oder ob bereits Anmeldeinformationen eingegeben wurden (Weiterleitung an das Java Bean). Über die Anweisung `<jsp:property>` werden dem Bean (durch Aufruf der den Eigenschaften zugehörigen Methoden) die zurückgegebenen Formularwerte (`user,pass`) der HTML-Seite direkt übermittelt.

5.4.3 Zertifizierung / SSL

Da es sich bei den innerhalb des Medienarchivs verwalteten Daten zum Teil um sehr sensible Informationen handelt, spielt die Gewährleistung eines sicheren Zugriffs eine wichtige Rolle. Die Übertragung mittels des herkömmlichen HTTP-Protokolls stellt sich hierfür als unzureichend heraus, da es nur einen unverschlüsselten Informationstransport zulässt.

Es gibt verschiedene Verfahren zur verschlüsselten Übertragung von Informationen auf Netzwerkebene. Auf Webebene hat sich hier die *Secure Socket Layer (SSL)* als Standard herausgebildet.

Wir werden hier kurz auf die wichtigsten Grundlagen von Verschlüsselungsverfahren eingehen und deren Anwendung für SSL vorstellen [Fuh00].

Symmetrische Verschlüsselung:

Bei den symmetrischen (auch *Secret-Key* genannten) Verschlüsselungsverfahren wird für die Ver- und Entschlüsselung der Daten der gleiche Schlüssel verwendet, der zwischen beiden Kommunikationspartnern auf sicherem Wege ausgetauscht werden muss. Die Sicherheit dieses Verfahrens hängt somit wesentlich von der Geheimhaltung des Schlüssels ab.

Asymmetrische Verschlüsselung:

Bei asymmetrischen (auch *Public-Key* genannten) Verschlüsselungsverfahren werden für die Ver- und Entschlüsselung der Daten zwei unterschiedliche Schlüssel verwendet. Diese Schlüssel hängen zwar über mathematische Verfahren voneinander ab, trotzdem können die Schlüssel nicht ohne weiteres voneinander abgeleitet werden. Um dies zu erreichen, werden sogenannte Einwegfunktionen verwendet. Eine solche Funktion ist zwar ohne großen Aufwand berechenbar, die Umkehrung jedoch praktisch nicht möglich.

Der Anwender erzeugt zunächst zwei Schlüssel, von denen der eine, der *Public-Key*, veröffentlich wird. Der andere Schlüssel (*Privat-Key*) bleibt in der Verwahrung des Anwenders. Eine geheime Nachricht für den Anwender wird dann mit Hilfe seines öffentlichen Schlüssels verschlüsselt und verschickt. Da niemand in der Lage ist, die Operation der Verschlüsselung ohne Kenntnis des privaten Schlüssels rückgängig zu machen, ist die Nachricht geschützt und kann nur mit Hilfe des privaten Schlüssels des Anwenders wieder entschlüsselt werden.

Der Austausch der öffentlichen Schlüssel kann auf zwei Wegen geschehen. Entweder die beiden Anwender tauschen diese Schlüssel persönlich aus, oder sie nutzen dazu eine vertrauenswürdige Einrichtung, eine sogenannte *Zertifizierungsstelle*.

Zertifizierung von öffentlichen Schlüsseln:

Für das Funktionieren des Public-Key-Verfahrens ist es von zentraler Bedeutung, Gewißheit über die Glaubwürdigkeit des öffentlichen Schlüssels zu besitzen. Dieses wird durch die angesprochenen Zertifizierungsstellen gewährleistet, die hierarchisch oder vernetzt aufgebaut sind. Jeder Benutzer kann nach einer Identitätsprüfung seinen öffentlichen Schlüssel von einer Zertifizierungsinstanz mit deren geheimem Schlüssel signieren lassen. Mithilfe des öffentlichen Schlüssels der Zertifizierungsinstanz kann dann jederzeit die Authentizität des öffentlichen Schlüssels von registrierten Benutzern überprüft werden.

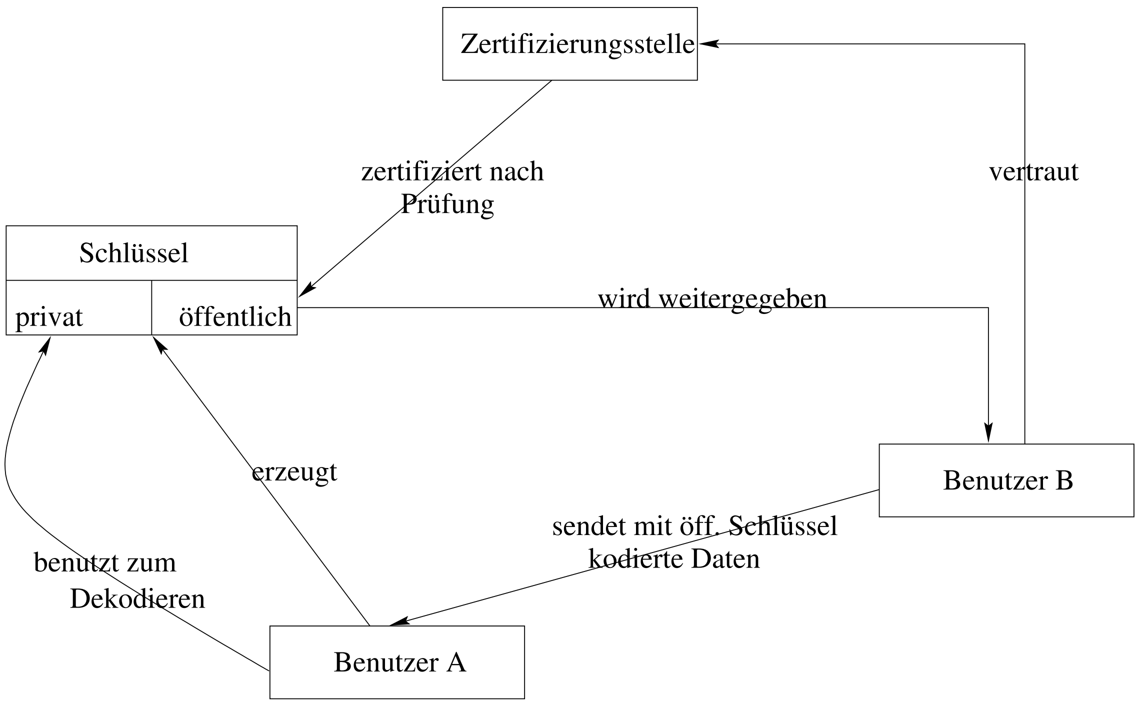


Abbildung 5.10: Ablauf eines Datenaustauschs mit Verschlüsselung

Socket Secure Layer (SSL):

Das SSL-Protokoll wurde im Jahr 1994 von der Firma Netscape [Netb] als Sicherungsprotokoll für die Transportschicht vorgestellt, 1995 bei der Internet Engineering Task Force (IETF) eingereicht und ist gegenwärtig in der Version 3.0 die Grundlage für die Arbeit der Transport

Layer Security Working Group bei der IETF.

Die Nutzung von SSL war primär für die Absicherung der Kommunikation zwischen einem WWW-Server und einem WWW-Client geplant, es existieren allerdings inzwischen auch Anpassungen für weitere Anwendungen wie Telnet oder FTP. Bei SSL handelt es sich genau genommen um mehrere Protokolle, die im ISO/OSI-Schichtenmodell als Schicht zwischen der Transportschicht (d. h. TCP oder UDP) und der Anwendungsschicht angesiedelt sind (s. Abbildung 5.11).

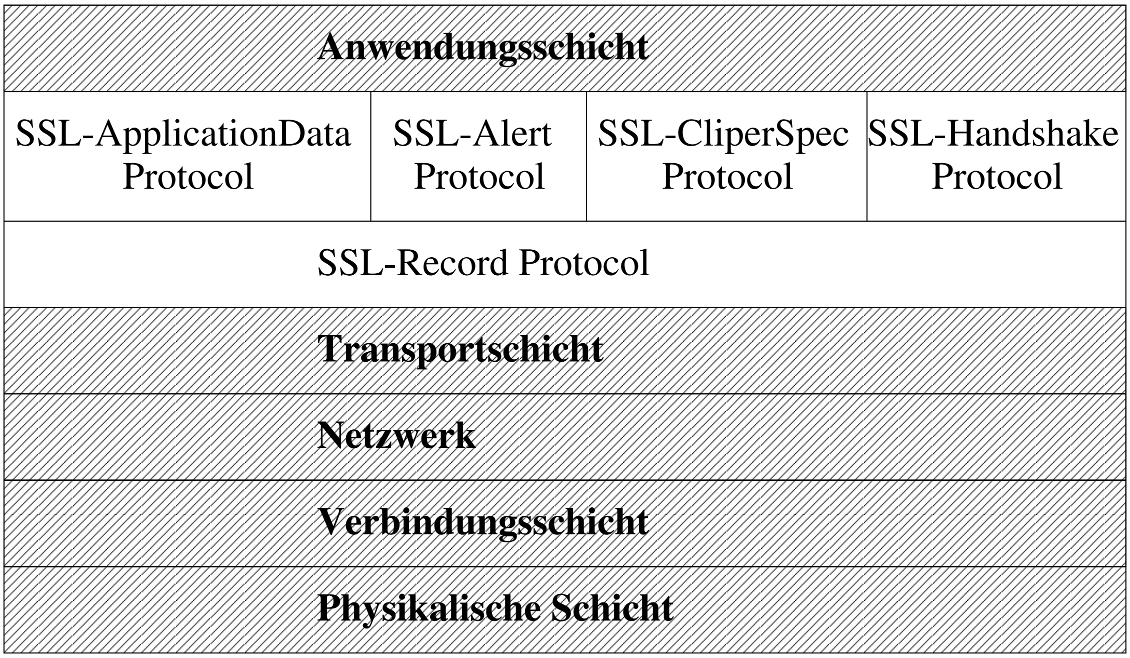


Abbildung 5.11: Einbettung von SSL in die unteren Schichten des ISO/OSI-Schichtenmodells

Der Verschlüsselungsalgorithmus von SSL arbeitet mit einer Kombination aus asymmetrischen und symmetrischen Verfahren. Auf der Basis des Public-Key-Verfahrens wird bei der Kontaktaufnahme (*handshake*) zwischen Client und Server ein geheimer Schlüssel ausgetauscht. Der SSL-Client generiert dazu mithilfe eines Zufallszahlengenerators eine Bitsequenz und sendet sie dem SSL-Server, verschlüsselt mit dessen öffentlichen Schlüssel. Aus dieser Bitsequenz generieren Server und Client nach einem festgelegten Algorithmus den *Master-Key*. Jeder weitere Datenaustausch erfolgt nun nach der ausgehandelten symmetrischen Verschlüsselungsmethode unter Benutzung des Master-Key.

In der Praxis läuft es so ab, dass ein Browser bereits Informationen über die wichtigsten Standardzertifizierungsstellen vorinstalliert hat. Der Betreiber eines SSL-Servers erzeugt sich ein Schlüsselpaar, in das der genaue Name des Servers kodiert ist. Nun muss der öffentliche Schlüssel von einer Standardzertifizierungsstelle (z. B. *VeriSign*, *Thawte*) signiert werden. Ein

Anwender der mittels eines Browsers auf den Server des Anbieters zugreift muss diese Zertifizierungsstelle ebenfalls in seinem Browser angemeldet haben.

5.4.4 Signierte Applets

Browser besitzen verschiedene Einschränkungen, die für die Benutzung innerhalb des Archivs Probleme aufwerfen. So ist beispielsweise der Austausch von Objekten mittels der nativen Browserfunktionalitäten unkomfortabel, da jeweils nur einzelne Dateien bewegt werden können. Da der Austausch von Objekten zwischen lokalem Rechner und Medienarchivserver ein wesentlicher Anwendungsfall ist, mussten hier spezielle Lösungen gefunden werden. Grundsätzlich wäre eine erweiterte Übertragung über Java Applets möglich. Leider haben herkömmliche Applets den Nachteil, dass sie nicht auf Ressourcen des Benutzerrechners zugreifen können. Aus diesem Grund werden signierte Java Applets eingesetzt.

Ein signiertes Java Applet kann die engen, durch den Browser vorgegebenen Sicherheitsvorgaben, nach Rückfrage an den Benutzer überwinden. Voraussetzung ist, dass der Code mittels eines *Zertifikats* (s. o.) signiert ist, um eine gesicherte Herkunft zu gewährleisten. Benutzt wird hierbei die bereits angesprochene asymmetrische Verschlüsselung.

Um auf Systemressourcen zuzugreifen, muss je nach verwendetem Browser bei der Implementierung eine spezielle Java-API eingebunden werden. Das führt dazu, dass eine Implementierung nur noch auf einer speziellen Plattform (Browser) anwendbar ist. Auch das Signieren der Applets geschieht je nach Browsertyp mit unterschiedlichen Werkzeugen.

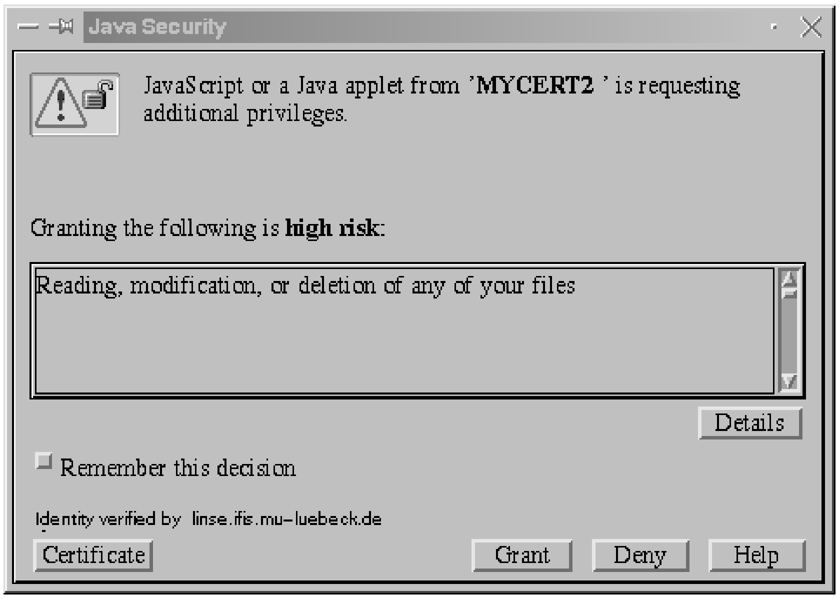


Abbildung 5.12: Sicherheitsabfrage beim Aufruf eines signierten Applet

Aus Benutzersicht gestaltet sich die Verwendung von signierten Applets vergleichsweise einfach und weitestgehend transparent. Beim Aufruf einer HTML-Seite, die ein signiertes Applet enthält, erscheint ein Hinweis-Fenster (Abbildung 5.12). Dieses Fenster informiert ihn darüber, dass es sich um ein signiertes Applet handelt, welches erweiterte Rechte beansprucht. Der Benutzer kann daraufhin diese Rechte gewähren (*grant*) oder verweigern (*deny*). Die Rechte bleiben (falls nicht anders spezifiziert) bis zur Beendigung des Browser erhalten.

In Abhängigkeit von der Benutzerentscheidung stehen dem Applet anschließend erweiterte Möglichkeiten und Rechte zur Verfügung.

5.4.5 Java Script, CSS und Dynamic HTML

Java Script (welches in seiner standardisierten Form ECMAScript [Neta] heißt) ist eine im Webkontext entwickelte Skriptsprache. Die Ähnlichkeit des Namens zur Programmiersprache Java ist durch Marketingüberlegungen begründet. Inhaltlich unterscheiden sich diese Sprachen sehr stark. Während Java als objektorientierte Sprache zu sehen ist, kann man Java Script höchstens als *objektbasiert* bezeichnen. Der Grund dafür ist, dass in Java Script zwar Objekttypen definiert und mit ihnen Objekte erzeugt werden können, Vererbung und die sich daraus ergebenden softwaretechnischen Vorteile aber entfallen. Darüber hinaus ist die Typisierung in Java Script sehr viel flexibler, damit aber auch fehlerträchtiger. Für praktische Anwendungen ist die Möglichkeit der Code-Erzeugung zur Laufzeit eine nützliche Spracheigenschaft.

Die wichtigsten Unterschiede zwischen Java Script und Java sind der folgenden Tabelle zu entnehmen:

Java Script	Java
Interpreter	Compiler
objektbasiert	objektorientiert
Objekttypen	Klassen und Vererbung
Objekthierarchie	Klassenbibliothek
„loose typing“	„strong typing“
Code in HTML eingefügt	separate Klassendatei
Browser erhält Code im Klartext	Browser erhält Byte-Code

Java Script wurde für die Anreicherung von HTML-Seiten mit aktivem Code entwickelt. Auf diesem Wege lassen sich Funktionen definieren und Objekten Ereignisse zuordnen (Event-Action-Paradigma, s. o.). Dadurch gewinnt eine Seite in gewisser Weise an Aktivität, da bereits auf Clientseite Eingaben verarbeitet werden und für die Präsentation genutzt werden können. Aus praktischer Sicht entfällt so der ständige Austausch zwischen Client und Server.

Leider sind die Browserhersteller bei erweiterten Sprachelementen unterschiedliche Wege gegangen. Daher ist es bei komplexeren Anwendungen i. d. R. browserbedingt notwendig, unterschiedliche Fassungen von Webinhalten bereitzustellen.

Cascading Style Sheets (CSS) sind ein von Microsoft definierter Standard zur Spezifikation des Layout von HTML-Seiten. Mit CSS wird es möglich, das Aussehen von Tags neu zu gestalten und allgemeine Seiteneigenschaften festzulegen.

Die gleichzeitige Nutzung von Java Script und CSS wird häufig als *Dynamic HTML (DHTML)* bezeichnet. Bei der Erstellung von Webpräsentationen lassen sich mit diesen Techniken sehr viel elegantere und optisch ansprechendere Seiten erzeugen. Diese sind jedoch auch mit einem erhöhten Programmieraufwand behaftet.

Zusammenfassung und Verwendung im Archiv:

Innerhalb des Webservers kommt eine Kombination aus Servlets, Java Server Pages und Java Beans zum Einsatz, die ihrerseits auf den RMI-Server zugreift. Die geringere Performance von Java wirkt sich in diesem Zusammenhang aus den bereits angesprochenen Gründen nur marginal aus.

Die Präsentation des Medienarchivs auf dem Webclient ist aufgrund der notwendigen Funktionalitäten komplett mittels der Techniken HTML, Java Script und DHTML realisiert. Um den Programmieraufwand im Rahmen halten zu können, wurden teilweise kleinere Skriptbibliotheken von Drittanbietern genutzt.

Die Verwendung von SSL für das Medienarchiv war eine direkte Folgerung aus den Benutzeranforderungen. Der Zugriff auf das Archiv ist demzufolge wahlweise mit oder ohne verschlüsselte Übertragung möglich. Die Übertragung von Dateien und Verzeichnissen zwischen Client und Server wird aufgrund der angesprochenen Probleme mit signierten Applets umgesetzt. Als Zertifikataussteller wurde der Anbieter *Thawte* [Ltd] gewählt.

5.5 Spezielle Lösungen

Neben den allgemeinen Techniken wurden für die Realisierung des Medienarchivs Java-Produkte von Drittanbietern benutzt. Auf die verwendeten Produkte werden wir hier kurz eingehen.

5.5.1 Infonyte XQL-Suite

Die GLOBIT Infonyte XQL-Suite [Tec, HMF99] ist eine Java-basierte Applikation zur Speicherung und Anfrage von umfangreichen Mengen mitunter großer XML-Dokumente in DOM-Form. Sie vereint zwei grundsätzliche Techniken,

1. eine persistente Implementierung des Document Object Model, genannt *PDOM*; und
2. eine komplette Implementierung der XQL-Sprache mit zusätzlichen Erweiterungen.

Das persistente DOM (PDOM) implementiert die komplette DOM API auf indizierten, binären Dateien. Dokumente werden geparst und in binärer Form gespeichert. Sie sind so über DOM-Operationen ohne zusätzliches Parsieren erreichbar. Eine Caching-Architektur erhöht die Geschwindigkeit von Zugriffen. Ein weiterer Vorteil des PDOM ist, dass DOM-Instanzen für deren Bearbeitung nicht komplett in den Speicher geladen werden müssen. Dadurch wird es möglich, sehr große Dokumente zu verarbeiten und zu manipulieren.

Die XQL-Engine implementiert den XQL-Vorschlag aus [RLS98]. Zusätzlich besitzt es Optimierungsstrategien, insbesondere beschleunigte Anfragen auf mehreren Dokumenten. Außerdem ist die Sprache durch die Verwendung benutzerdefinierter Funktionen erweiterbar. Der XQL-Prozessor kann für Anfragen auf beliebigen DOM-Bäumen genutzt werden, wobei in der Kombination mit PDOM besondere Synergien entstehen.

Es bleibt wichtig anzumerken, dass es sich bei PDOM um keine vollwertige Datenbank handelt, da z. B. weder ein allgemeines Transaktionskonzept vorhanden ist, noch eine mehrbenutzerorientierte Verwaltung der Daten unterstützt wird. Auch gibt es keine Unterstützung für die Generierung eines transparenten Index. Vielmehr konzentriert sich PDOM auf die wesentlichen Aspekte der Ablage und des Retrieval von XML-Informationen.

5.5.2 JavaCC

JavaCC ist ein Parsergenerator, der auf der Sprache Java aufbaut. Er ist im gewissen Sinn die Java-Entsprechung der C-Werkzeuge *Lex/Yacc* [Her95]. Ein Parsergenerator ist ein Programm, welches eine Grammatikdefinition einliest und in ein Programm einer bestimmten Programmiersprache (hier Java) überführt. Die dabei entstandenen Programme dienen der lexikalischen und syntaktischen Analyse von Dokumenten, welche im JavaCC vereint wird.

Während der lexikalischen Analyse eines Textes werden die Textteile nach vorgegebenen Regeln erkannt und klassifiziert. Für jede Klasse von Textteilen werden dabei sogenannte Token definiert.

Die Syntaxanalyse prüft ein Dokument anschließend gegen eine vorgegebene Grammatik. In Abhängigkeit von Regeln können gleichzeitig Aktionen (insbesondere Ausgaben) ausgeführt werden.

JavaCC bietet darüber hinaus Möglichkeiten zur Erzeugung von Baumstrukturen, Annotierung von Grammatikteilen mit Aktionen, Debuggingoperationen und einiges mehr.

Zusammenfassung und Verwendung im Archiv:

PDOM wird im Archiv zusammen mit der zugehörigen XQL-Implementierung für die Verwaltung und Abfrage von MDM-Informationen benutzt. Es stellte sich dabei heraus, dass für größere Objektmengen eine anwendungsorientierte Optimierung der XQL-Anfragen notwendig ist.

Für die Umsetzung der benutzerseitigen Suchausdrücke in XQL-Kommandos wird JavaCC benutzt. Hierfür wurden entsprechende Grammatikvorgaben und -umsetzungen definiert.

5.6 Serverseitige Implementierung

5.6.1 Die MDM-API

Die MDM-API, welche die komplette MDM-Logik implementiert, ist als RMI-Lösung realisiert. Die Client-API dient als zentrales Zugriffswerkzeug für alle Applikationen. Die Abbildung 5.13 zeigt das Zusammenspiel der Klassen auf. Die komplette Hierarchie der MDM-Klassen und -Schnittstellen ist dem Anhang C zu entnehmen.

Eine clientseitige Applikation, die auf die MDM-API zugreift, muss verschiedene Phasen durchlaufen:

- Durch die statische Methode `getSession()` der Klasse `MDMClientSessionFactory` wird nach einer Authentifizierung eine (erste) Session (Interface `MDMSession`) erzeugt. Diese Session liefert so eine erste Referenz (bootstraping) auf Objekte des Archivs.

- Eine Session dient als zentrale Instanz der Erzeugung und dem Auslesen von Medienarchivobjekten (`MDMObject`), der Komposition von Anfragen (`MDMQuery`) oder der Verwaltung von Benutzerinformationen (`MDMUser`).

- Das `MDMObject` repräsentiert ein konkretes Objekt im Archiv. Über diese Schnittstelle können jegliche Manipulationen vorgenommen werden. Die genauen Methoden der Schnittstelle sind dem Anhang A zu entnehmen.

- Die Eigenschaften eines `MDMObject` sind in einer speziellen Klasse (`RdfDom`) untergebracht, welche entsprechende Methoden zum Setzen und Verändern besitzt. Die Abkürzung RdfDom ist in diesem Zusammenhang historisch gewachsen und nicht mit dem Resource Description Framework zu verwechseln.

- Zu den einzelnen Schnittstellen gibt es jeweils serverseitige Objekte, welche die tatsächlichen Funktionalitäten ausführen (`MDMSessionImpl, MDMObjectImpl` usw.). Diese serverseitigen Implementierungen werden für den Benutzer transparent über Stub- und Skeletonklassen benutzt.

- Da Java Streaming-Klassen nicht serialisierbar sind, muss die Übertragung von solchen Objekten zwischen Client und Server manuell implementiert werden. Dieses geschieht über einen Objekt-Server (`ObjectServer`), der ein `MDMObject` erzeugt oder ein bestehendes mit Daten auffüllt.

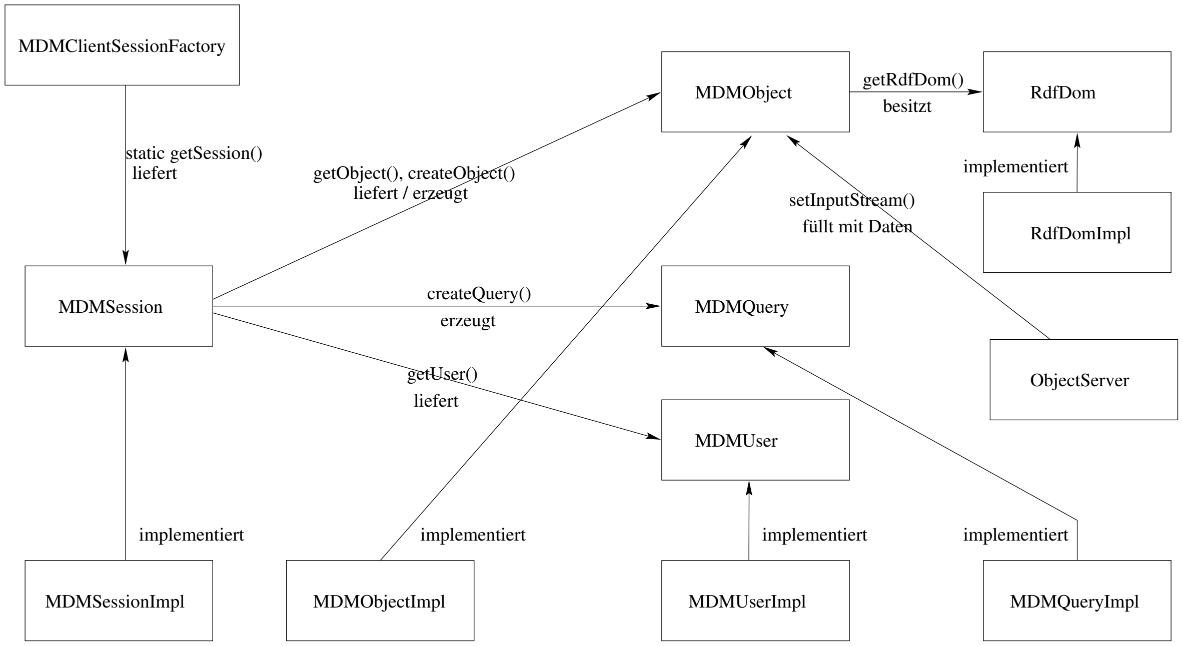


Abbildung 5.13: Zusammenspiel der MDM-Klassen

Das folgende Java-Programm erzeugt eine Session innerhalb des Archivs, liest ein Objekt aus, verändert in diesem den Wert einer Eigenschaft und gibt schließlich den Namen des zugehörigen Elternobjekts aus.

```
...
    try{
    // Session vom Server 'www.medienarchiv'
    // als Benutzer 'gast' holen
    MDMSession session=MDMClientSessionFactory
               .getSession("www.medienarchiv","gast",null);

    if(session==null) error("Fehler beim Anmelden");

    // Objekt mittels der Session anfragen
    MDMObject object = session.getObject("/workspace/hilfen.txt");

    // Eigenschaftsklasse instanziieren und Eigenschaft ändern
    RdfDom dom=object.getRdfDom();
    dom.setPropertyValue("dc:description",
                         "Hilfedatei der Anwendung");

    // Ausgabe des Namens des Elterobjekts
    Systen.out.println(object.getParent().getName());
```

```
    // RemoteException muss jeweils explizit abgefangen werden
    } catch (java.rmi.RemoteException e) {...}
...
```

5.6.2 Transformationsschnittstelle (ExCoT)

Das MDM verfolgt als Middleware-Schicht die Integration von heterogenen Datenbeständen unterschiedlichster Herkunft. Für die Integration dieser Daten spielen Transformationen eine wichtige Rolle. Da es besonders im medizinischen Kontext eine Reihe von Spezialformaten gibt, ist die Bereitstellung von allgemeinen Transformationsfunktionen im Archiv eine essentielle Aufgabe.

Es gibt bereits eine Vielzahl von Transformations- und Integrationssprachen [CDSS98, BW97, FW93] und darüber hinaus Anfragesprachen für semistrukturierte Daten mit ähnlichen Möglichkeiten. Jedoch sind Anfragesprachen nicht besonders geeignet, um Transformationen mit reichhaltigen Restrukturierungen zu beschreiben. Einige Vorschläge, welche auf Datenbanktransformationen hinzielen, nutzen einen schemaorientierten Ansatz, der vergleichsweise unflexibel ist [ACM93, ACM97].

Um die Transformationen und Metadatenextraktionen nicht auf eine vordefinierte Anzahl von Formaten oder Anwendungsszenarien zu beschränken, wurde die MONTANA *Extraction and Transformation API (ExCoT)* entwickelt. Diese Schnittstelle erlaubt die Implementierung medienspezifischer Filter, wobei bestehende Lösungen von Drittanbietern problemlos eingebunden werden können.

Eine konkrete Anwendung der ExCoT-API ist die Erzeugung von minimierten Objektvorschauen (*Thumbnails*) für die beschleunigte Präsentation im Webbrowser. Darüber hinaus dienen zahlreiche Filter der vom Client unabhängigen Darstellung von Spezialformaten (z. B. Word, Postscript oder medizinische Spezialformate).

Eine weitere zentrale Aufgabe ist die Erzeugung der Objektbeschreibungen. Eine Objektbeschreibung besteht aus einer Reihe von Eigenschaften (vgl. 4.1). Diese werden je nach Medientyp mit zusätzlichen Informationen angereichert und anschließend in den MDM-Datenbestand integriert.

Anhang B zeigt die ExCoT-Schnittstelle, die der Programmierer bei der Umsetzung eigener Transformations- und Extraktionsfunktionen implementieren muss. Diese Schnittstelle ist sowohl allgemein als auch einfach gehalten, um sie so für eine breite Benutzerschicht verständlich und anwendbar zu halten.

In der derzeitigen Fassung des Archivs sind medienspezifische Filter für drei konkrete Bereiche implementiert:

- **Graphikformate:** Die wichtigsten Graphikformate (JPEG, GIF, TIF, BMP) wurden berücksichtigt und sind wechselseitig transformier- und analysierbar.
- **Dokumentformate:** Sowohl plattformabhängige Formate (Word, Postscript) als auch

übergreifende Standards (HTML, PDF) sind umwandelbar und können einer inhaltichen Analyse unterzogen werden.

- **XML:** Besondere Bedeutung kommt XML-Dokumenten zu, die durch deren MDM-konformes Format über XML-Transformationen direkt in den MDM-Datenbestand integrierbar sind. Für die erweiterte Definition von Formaten werden *s-xscheme*s verwendet. Hierdurch können insbesondere (wie bereits im dritten Kapitel gezeigt) reichhaltigere XSL-Transformationen definiert werden.

5.7 Benutzerorientierte Zugriffsschnittstellen des Archivs

Wir werden in diesem Abschnitt auf die verschiedenen, für den Benutzer entwickelten Schnittstellen eingehen.

5.7.1 Webserver

Die Implementierung des Webservers wurde, wie oben angesprochen, über eine Kombination von Java Servlets, Java Server Pages und Java Beans realisiert.

Java Server Pages wurden für die Gestaltung der HTML-Seiten benutzt. Da die HTML-Seiten mit ihren statischen und dynamischen Inhalten im Laufe der Zeit recht komplexe Ausmaße angenommen haben, lässt sich so ein annehmbarer Pflegeaufwand erreichen.

Bei Seitenaufrufen, deren Ergebnisse für längere Zeit verfügbar sein müssen, wurden Java Beans verwendet. Dieses ist z. B. bei Suchoperationen der Fall, deren Ergebnis über mehrere Benutzeraufrufe verteilt präsentiert wird.

Servlets kommen dann zum Einsatz, wenn vergleichsweise prozessintensive Aufgaben mit kleinen HTML-Ausgaben erledigt werden müssen. Dieses geschieht z. B. beim Im- und Export von Dateien und Verzeichnisbäumen.

Die Verfolgung von Sessions wurde sowohl über clientseitige Cookies (d. h. abspeichern von Serverinformationen im Browser) als auch über URL-Rewriting (d. h. Übergabe über HTTP-Parameter) realisiert. Damit wird die Unabhängigkeit von den Browser-Einstellungen auf Benutzerseite gewährleistet.

Um dem Anwender möglichst schnell eine gewohnte Umgebung geben zu können, wurde das Design der Webschnittstelle [MON] ähnlich zum Microsoft Explorer gestaltet. Der Aufbau der Seiten geschieht innerhalb von drei Frames (s. Abbildung 5.14), die jeweils eigene Funktionalitäten besitzen, wie Navigation, Objektmanipulation oder Suche. Auch sind hier die wesentlichen Dateioperationen wie Copy, Cut & Paste, Erstellung von Verknüpfungen usw. verfügbar.

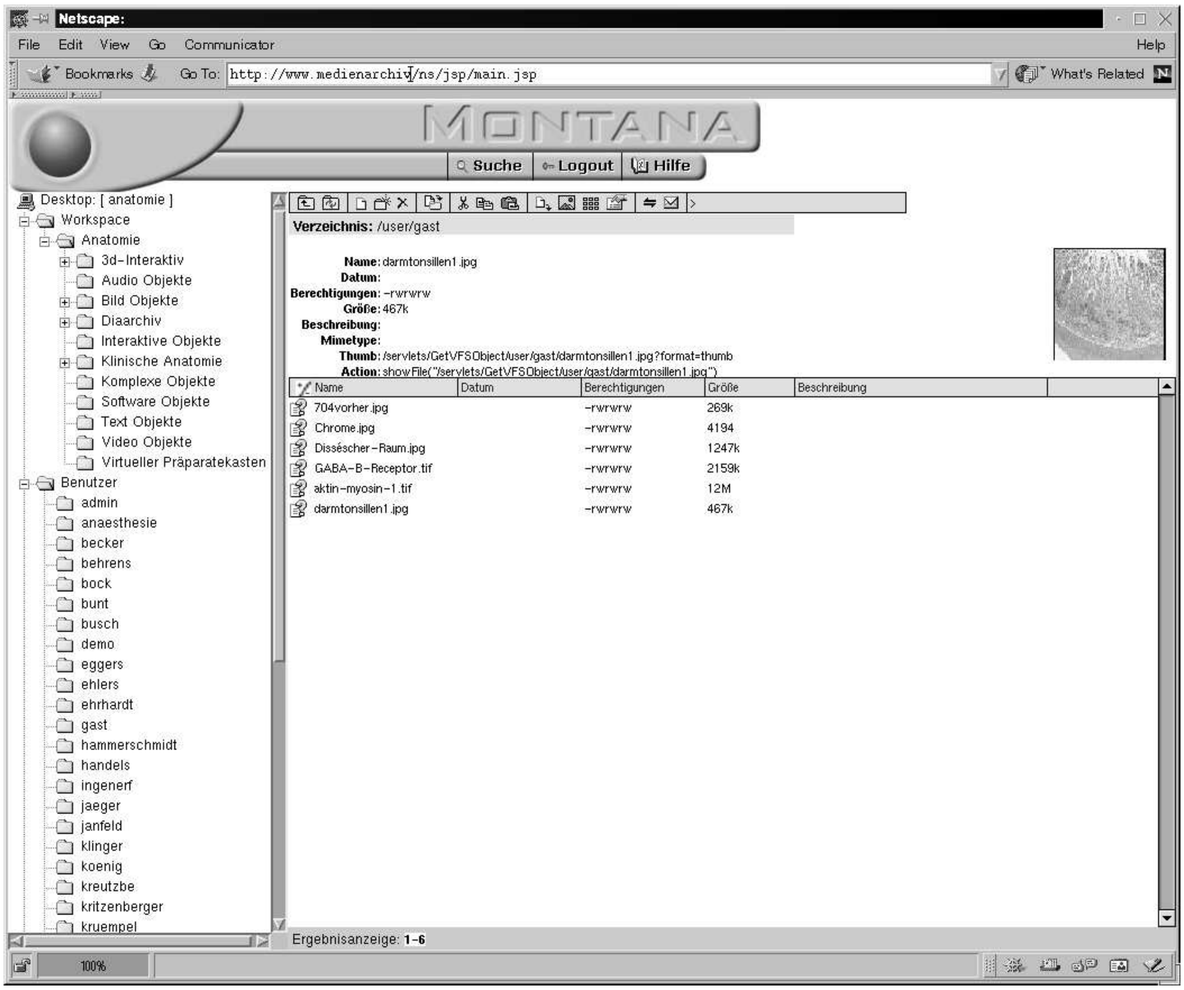


Abbildung 5.14: Screenshot des Webinterface

In der Abbildung 5.15 sind einzelne Kontrollfenster dargestellt, die das obige Hauptfenster ergänzen. Diese Kontrollfenster dienen der Manipulation von Objekteigenschaften (rechts), der Darstellung der eigentlichen Objekte (unten links) oder auch deren Voransichten (oben links).

Die zusätzlichen Kontrollfenster werden je nach Kontext automatisch geöffnet und geschlossen. Die Steuerung übernimmt dabei das Hauptfenster.

Parallel zur Nutzung der Schnittstelle sind kontextabhängige Hilfen erreichbar. Zu nahezu allen Funktionen sind solche Beschreibungen vorhanden, die dem Benutzer seine verfügbaren Möglichkeiten erläutern.

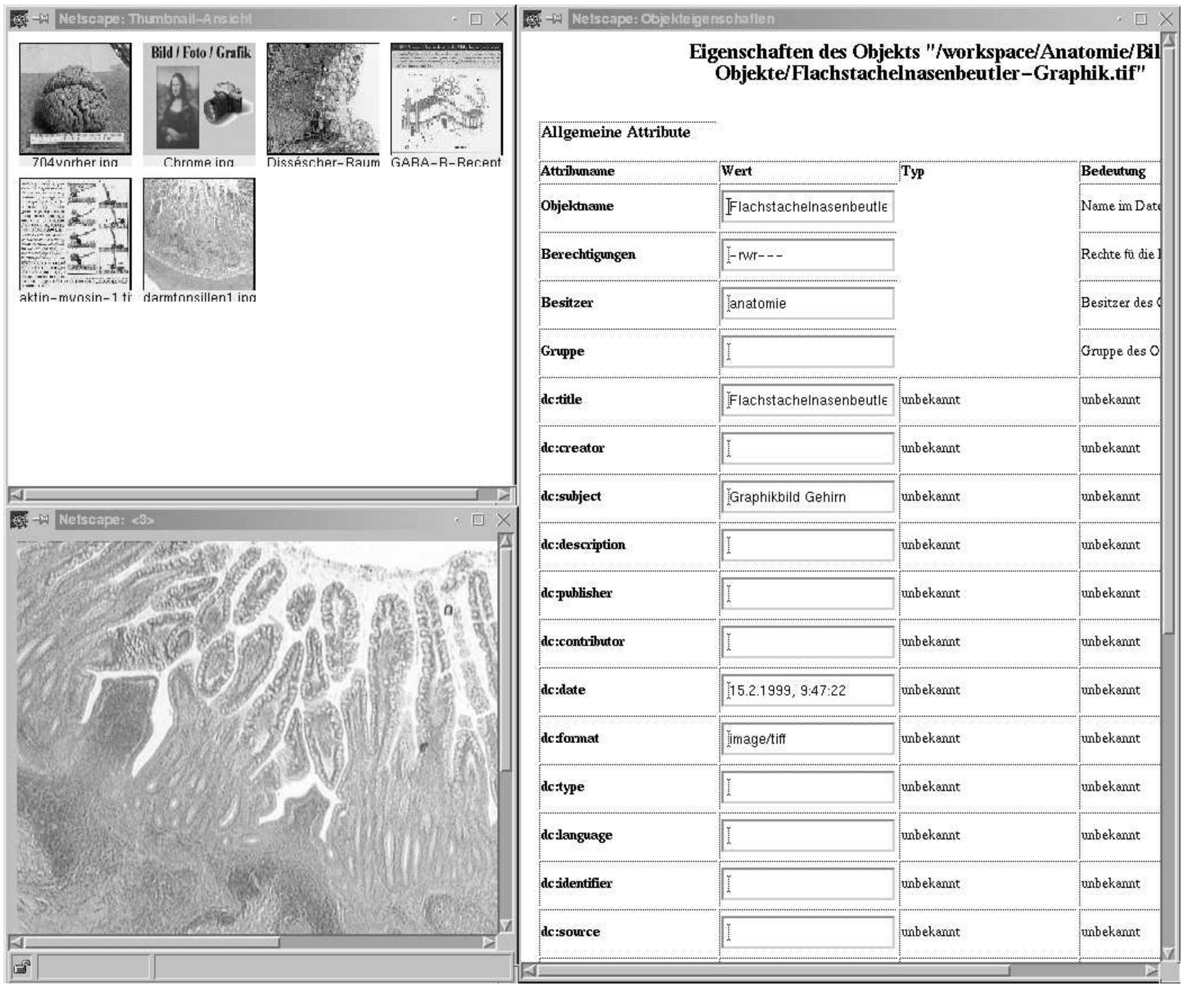


Abbildung 5.15: Screenshot einzelner Kontrollfenster

5.7.2 FTP-Server

Das *File Transfer Protocol (FTP)* ist ein aus der Unix-Welt stammendes, plattformunabhängiges Protokoll, welches der netzwerkweiten Unterstützung von allgemeinen Dateisystemoperationen dient. Für nahezu alle Betriebssysteme sind graphische Applikationen erhältlich, die den Zugriff auf andere Rechner mittels des FTP-Protokolls ermöglichen. Dieser Zugriff erfolgt oftmals in einem Maße transparent, als dass dem Benutzer der Unterschied zwischen der Arbeit auf einem lokalen Filesystem und einem entfernten FTP-Server verborgen bleibt.

Die FTP-Architektur genügt, ähnlich dem Webansatz, dem Client/Server-Prinzip. An einem FTP-Server können sich in der Regel beliebig viele FTP-Benutzer anmelden und „remote“ auf ein Dateisystem zugreifen. Der Hauptunterschied zum Webansatz liegt in der, für die

Zeit einer Sitzung, permanenten Verbindung. Es muss also nicht für jede Operation eine neue Verbindung hergestellt werden.

Der FTP-Server des MDM bildet die FTP-Funktionen auf die Hierarchien des Medienarchivs ab. Damit wird es für einen Benutzer möglich, diese Hierarchien mit FTP-orientierten graphischen Clients (s. Abbildung 5.16) oder auch der Kommandozeile zu manipulieren. Da das FTP-Protokoll nur grundlegende Operationen unterstützt, musste hier auf Funktionalitäten wie erweiterte Benutzerrechte oder Links verzichtet werden.

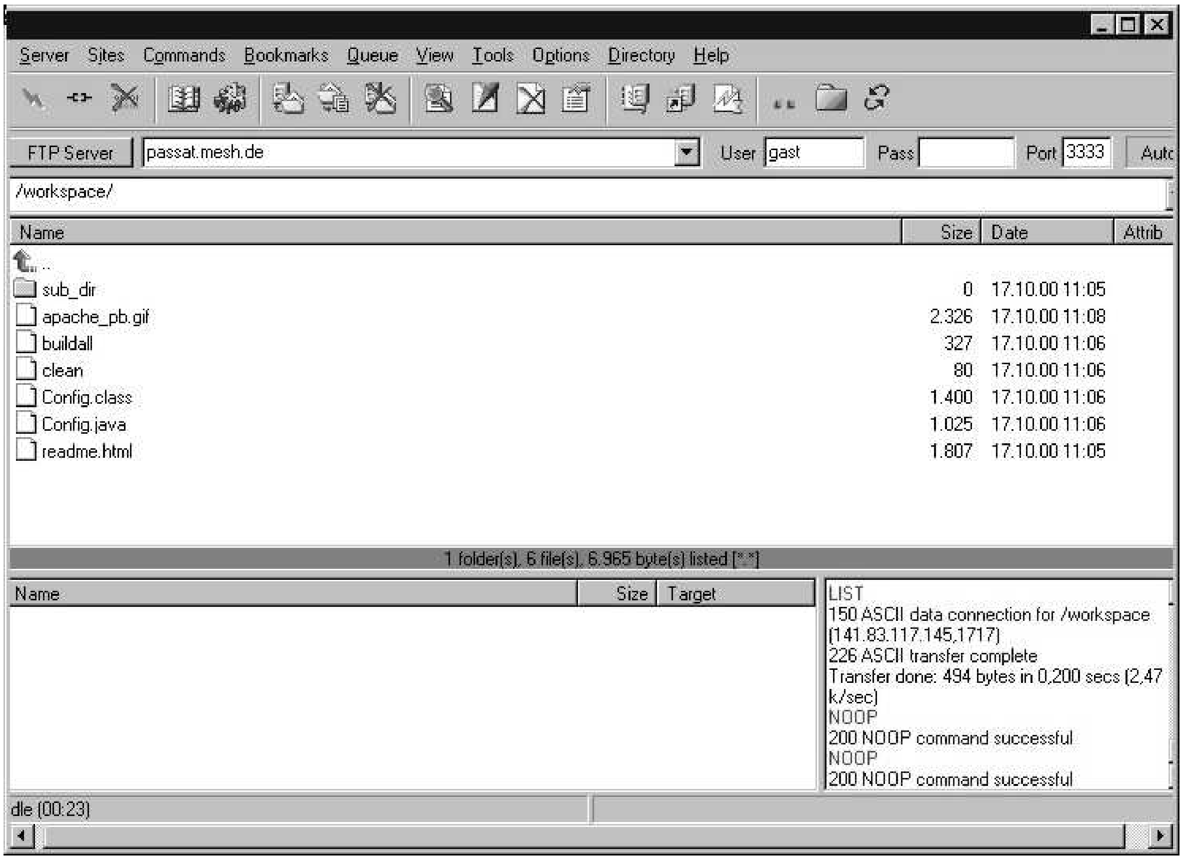


Abbildung 5.16: Screenshot eines FTP-Client

5.7.3 WAP-Server

Die *Wireless Application Protocol (WAP)*-Technologie erlaubt die Übertragung und Darstellung von Webinhalten auf mobilen Geräten wie Mobiltelefone (Handies), Handhelds oder *Personal Digital Assitants (PDAs)*. Zur Beschreibung von Seiten wird die *Wireless Markup Language (WML)* benutzt. WML ist als HTML-Gegenstück für Clients mit beschränkten Präsentationsmöglichkeiten zu sehen. WML ist eine konkrete Anwendung von XML.

WAP-fähige Geräte können neben einfachen Textelementen auch schwarz/weiß-Grafiken in beschränkter Auflösung darstellen. Die Übertragung hochaufgelöster oder farbiger Grafiken ist aufgrund der beschränkten Bandbreite mobiler Geräte (9600 Baud bei GSM) derzeit nicht vorgesehen. Aufgrund der geringen Bandbreite komprimiert das WAP-Protokoll die zu übertragenen Daten in einen speziellen Binärcode.

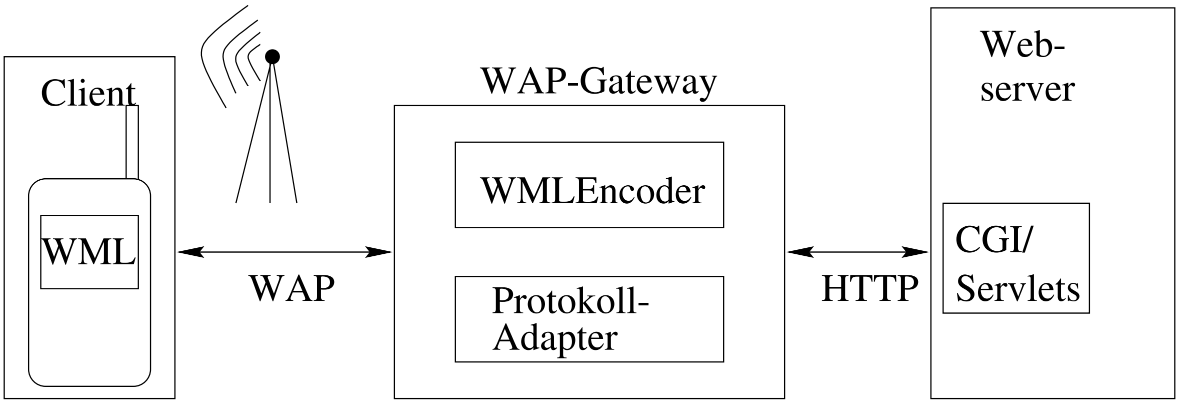


Abbildung 5.17: WAP-Netzwerkstruktur

Die WAP-Architektur ähnelt dem Webmodell, wobei es dessen Infrastruktur mitbenutzt. Der Austausch zwischen Client und Server ist in der Abbildung 5.17 dargestellt. Für den Austausch sind die folgenden Schritte notwendig:

1. Der Benutzer spezifiziert die URL eines WAP-Angebots.
2. Mittels des Protokolls WAP wird die Anfrage an ein Gateway übertragen. Die Übertragung geschieht dabei in der Regel über Funkverbindungen.
3. Das WAP-Gateway erzeugt aus der Anfrage des Benutzers eine konventionelle HTTP-Anfrage und schickt sie weiter an einen Webserver.
4. Der Webserver verarbeitet die Anfrage und erzeugt ein Dokument in WML-Form.
5. Der Webserver schickt die Antwort mittels HTTP zurück an das Gateway.
6. Das Gateway überführt die Antwort aus dem HTTP-Protokoll in eine komprimierte, binäre Form (WMLEncoder) und sendet sie gemäß dem WAP-Protokoll weiter.
7. Auf dem Benutzergerät werden die WAP-Daten entpackt und das resultierende WML-Dokument angezeigt.

Für das Medienarchiv wurde eine auf obiger Architektur basierende WAP-Schnittstelle implementiert. Dieser WAP-Zugang ist, wie schon die Webschnittstelle, mit einer Kombination aus Java Servlets, Java Server Pages und Java Beans umgesetzt worden [Kra00].

Abbildung 5.18: Screenshot einer WAP-Emulation

Dem Benutzer stehen somit die wichtigsten Archivoperationen in mobiler Weise zur Verfügung. Dadurch wird er in die Lage versetzt, Informationen jederzeit und jederorts aus und in das Archiv zu transportieren und Medieninformationen zu suchen und zu manipulieren.

Abbildung 5.18 zeigt eine Momentaufnahme des Zugriffs auf den WAP-Server des Medienarchivs mithilfe einer WAP-Emulation.

5.7.4 Virtuelle Diaschiene

In zahlreichen medizinischen Fächern existieren umfangreiche analoge Archive mit Diapositiven, Fotopositiven, und Videofilmen sowie Präparate und Modelle, die als Hilfsmittel für Präsentationen in Lehre und Forschung nicht mehr wegzudenken sind. Einige Institute innerhalb der Medizinischen Universität zu Lübeck begannen, diese Materialien nach und nach zu digitalisieren. Daraus resultierte der Wunsch, diese meist großen Medienobjekte im

Medienarchiv zu speichern und später für Veranstaltungen und wissenschaftliche Präsentationen zusammensetzen zu können.

Die digitalisierten Materialien werden im Medienarchiv (wie in der Realität) hierarchisch strukturiert, so dass sich ein digitales Datenbank-basiertes Abbild des analogen Archivs ergibt. Im analogen Archiv ist es möglich bis zu 200 Dias vor einem Lichtkasten zu betrachten, so dass relativ leicht Diapositive zu einer Thematik verglichen und zusammengestellt werden können. Eine vergleichbare Möglichkeit sollte auch für die Bild- und Graphikobjekte des Medienarchivs realisiert werden. Hierzu wurde das Java-basierte Visualisierungs- und Auswahlsystem „*Virtuelle Diaschiene*" (VDS) [3] entwickelt.

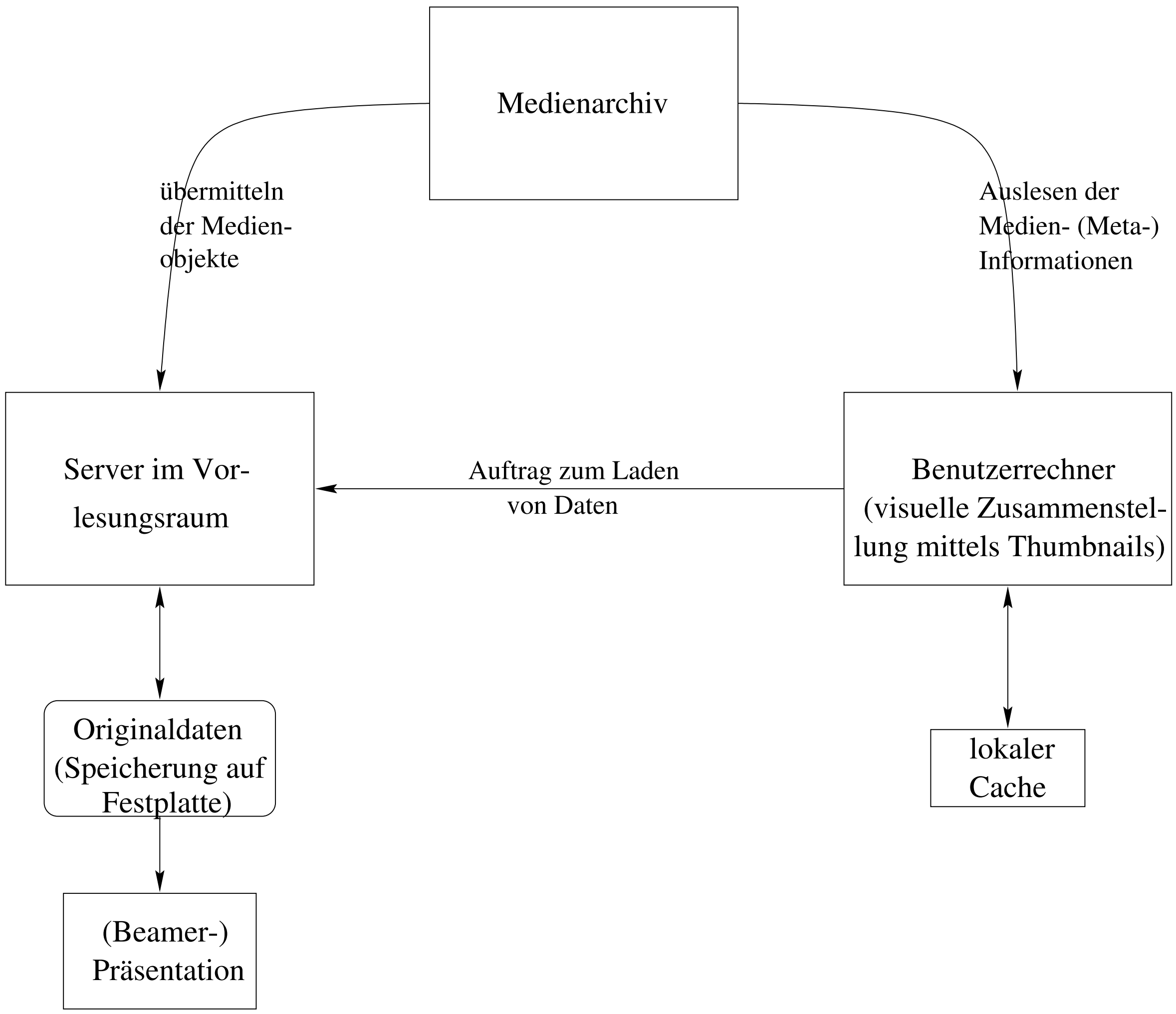


Abbildung 5.19: Architektur der virtuellen Diaschiene (VDS)

[3] Die virtuelle Diaschiene (VDS) entstand im Institut für Anatomie der MUL unter der Leitung von Dr. Oliver Schmitt.

Ein Vorteil zu realen Diaschienen besteht zum einen im einfachen Umstellen einzelner Dias, ohne die Dias von Hand neu zu sortieren und zum anderen in der Positionierung des gleichen virtuellen Dias an mehreren Stellen auf der VDS.

Die VDS-Applikation wird sowohl auf dem Vorlesungsrechner als auch auf einem Benutzerrechner, der zur Verarbeitung dient, betrieben. Auf dem Benutzerrechner wird die Präsentation zusammengestellt. Danach werden die Informationen über diese Zusammenstellung an den Vorlesungsrechner übertragen. Auf dem Benutzerrechner wird ein Übertragungsprozess gestartet, der einem Hintergrundprozess auf dem Vorlesungsrechner (von dem der zuvor zusammengestellte Inhalt einer VDS mittels Beamer projiziert werden soll) mitteilt, dass dieser die digitalen Dias aus dem Medienarchiv im Originalformat herunterladen muss.

Die fertig gestellte VDS kann für Archivierungszwecke ebenfalls wieder in das Medienarchiv eingetragen oder auf einem Client-Rechner gesichert werden.

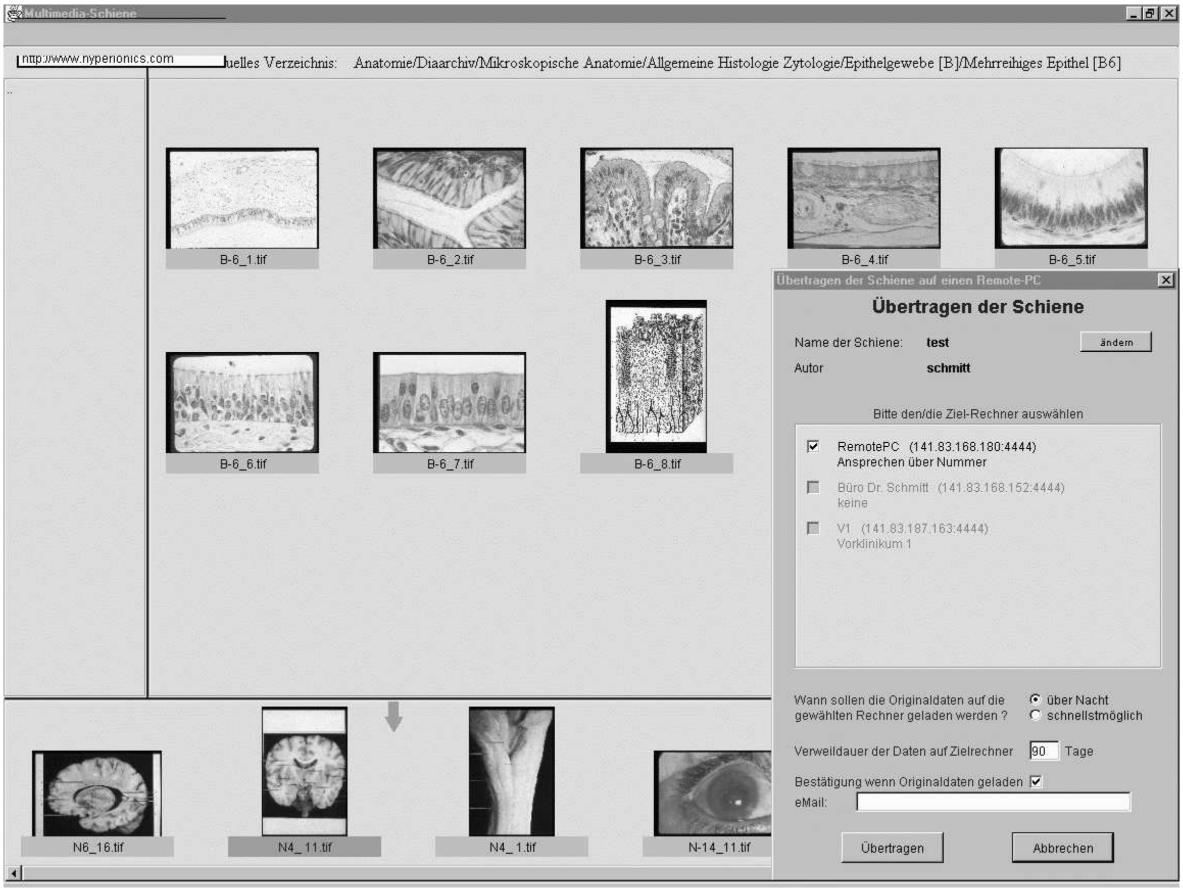


Abbildung 5.20: Screenshot der virtuellen Diaschiene

Die Daten werden auf dem Vorlesungsrechner in einem temporären Verzeichnis gesichert und stehen dann zur Darstellung zur Verfügung. Dieses Client/Server-Konzept der VDS hat den Vorteil, dass auch umfangreiche Daten für eine Präsentation frühzeitig übertragen werden können, ohne dass Hochlastphasen im Campus-Netz oder des Medienarchivservers die Performance einer Präsentation beeinträchtigen. Die Client/Server-Applikation auf dem Vorlesungsrechner ist fehlertolerant, insofern es unvollständige Übertragungen von Datenobjekten erkennt und in weiteren Anfragen an den Medienarchivserver versucht zu vervollständigen. Ist eine Datenübertragung abgeschlossen, so benachrichtigt die Applikation auf dem Vorlesungsrechner den Benutzer darüber.

Das Zusammenwirken der verschiedenen Teile ist in Abbildung 5.19 dargestellt. Die Abbildung 5.20 zeigt einen Screenshot der Diaschiene während der Zusammenstellung von Präsentationsmaterialien.

5.8 Zusammenspiel der Komponenten im Medienarchiv

Die Abbildung 5.21 zeigt das Zusammenwirken der verschiedenen Techniken und Komponenten innerhalb der konkreten Implementierung und instanziiert somit die anfangs des Kapitels vorgestellte allgemeine MONTANA-Architektur (vgl. Abbildung 5.3).

Server-Ebene:

Auf der Server-Ebene befindet sich die MDM-Logik, welche über eine API gekapselt ist. Diese API stellt den Anwendungen sämtliche Funktionalitäten des Archivs zu Verfügung. Die Speicherung der (Meta-)Daten geschieht mittels zweier Repositorien. Während das *BLOB-Repository (NFS)* die Rohdaten aufnimmt und verwaltet, werden innerhalb des *XML-Repository (PDOM)* die auf XML projizierten Struktur- und Objektmetadaten der logischen MDM-Ebene abgelegt. Die Speicherung dieser XML-Daten geschieht mithilfe des persistenten DOM [Tec]. Die API ist per *Remote Method Invocation (RMI)* für beliebige Anwendungen erreichbar.

Innerhalb des Servers wird die *Extraction and Transformation-API (ExCoT)* verwendet. Diese API dient zum einen der Transformation von Medientypen in spezielle Formate, zum anderen übernimmt sie die Extraktion von typspezifischen Metainformationen. Die API ist dabei so allgemein gehalten, dass beliebige benutzerdefinierte Filter (auch von Drittanbietern) integriert werden können.

Zur Auswertung von Anfragen werden die vom Benutzer formulierten Suchausdrücke mit einem, durch einen Parsergenerator erzeugten Compiler, überprüft, in äquivalente XQL-Statements überführt und mittels einer XQL-Query-Engine ausgewertet.

Die Beschreibung der Archivstrukturen geschieht mittels eines erweiterten XML-Schemaparsers. Dieser XML-Schemaparser setzt den DTD-erweiternden *s-xschemes*-Formalismus [Beh00a, Kre00] um. Der Parser dient der Überprüfung der schematischen Restriktionen (sowohl der allgemeinen Strukturen als auch der typspezifischen Vorgaben) und der Überführung von Daten in semantisch reichhaltigere Form.

Server-Applikations-Ebene:

Auf der Server-Applikations-Ebene befinden sich serverseitige Anwendungen wie Webserver, FTP- und WAP-Server, die die Medienarchiv-Dienste mit den jeweils protokollspezifischen Einschränkungen zur Verfügung stellen. Der Webserver ist optional über eine zertifizierte SSL-Verbindung erreichbar. Die Implementierung ist mittels Java Servlets und Java Server Pages realisiert, die ihrerseits über Java Beans auf den RMI-Server zugreifen.

Auf der Server-Applikations-Ebene ist auch an weitere Dienste zur Einbindung des Archivs auf Betriebssystemebene (Unix NFS, Windows SMB) gedacht.

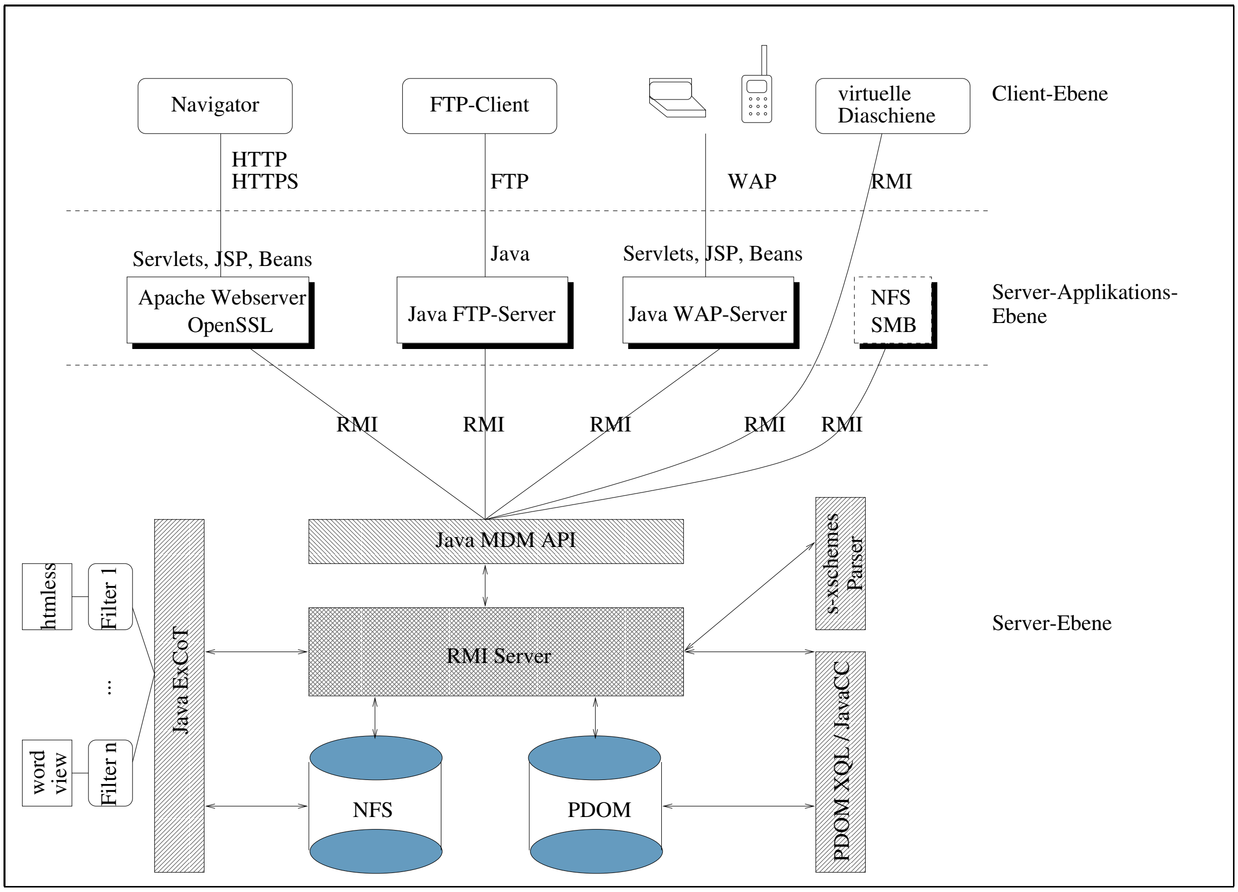


Abbildung 5.21: Zusammenspiel der konkreten Techniken im Medienarchiv

Client-Ebene:

Auf der Client-Ebene befinden sich die benutzerseitigen Anwendungen wie Webbrowser, FTP-Client oder WAP-Gerät. Daneben ist über den RMI-Ansatz auch die Möglichkeit gegeben, beliebige Applikationen zu entwerfen (wie z. B. die virtuelle Diaschiene), die mittels der MDM-API auf die volle Funktionalität des Archivs zugreifen können.

5.9 Praktischer Einsatz des Archivs und Erfahrungen

Derzeit verwaltet das Archiv ein Volumen von ca. 100 Gigabyte multimedialer und administrativer Daten, wobei dieses Volumen täglich steigt. Die Datenmenge ist auf etwa 30.000 Objekte verteilt. Der daraus resultierende XML-Datenbestand beläuft sich im Moment auf rund 500 MB.

Dieser Datenbestand stammt zu einem nicht unerheblichen Teil aus der ersten Version des Medienarchivs. Die Migration des Datenbestands wurde mit Werkzeugen vorgenommen, die speziell für diesen Zweck entwickelt worden sind.

Für die Datensicherheit ist ein Hardware-RAID zuständig. Zentrale Einrichtungen sind über Anbindungen mit hohen Durchsatzraten an das Archiv angeschlossen. Die Antwortzeiten der Anfrageschnittstelle liegen bei Objektmengen von ca. 5000 Stück im Sekundenbereich und sind damit akzeptabel.

Durch die kontinuierliche Protokollierung der Zugriffe lässt sich das Ablaufverhalten des Archivs nachvollziehen. Für die Behandlung von etwaigen Fehlersituationen wurden administrative Tools entworfen.

Wie die bisherigen Erfahrungen gezeigt haben, stellt sich die Implementierung als zuverlässig dar. Das Zusammenspiel der unterschiedlichen Komponenten und Anwendungen läuft im allgemeinen störungsfrei.

6

Zusammenfassung und Ausblick

„Zwei Dinge sind unendlich, das Universum und die menschliche Dummheit, aber bei dem Universum bin ich mir noch nicht ganz sicher.“

ALBERT EINSTEIN

Mit der zunehmenden Menge elektronisch verfügbarer Informationen wird es immer wichtiger, flexible Möglichkeiten zur Beschreibung und Strukturierung verschiedenartiger Objekte zur Verfügung zu stellen. Existierende Mechanismen zur Spezifikation proprietärer Formate erweisen sich hierfür als unzureichend.

In dieser Arbeit wird ein Modell für die Darstellung und Verarbeitung von strukturierten Informationen insbesondere bei der Organisation und Repräsentation von Medienobjekten auf der Basis der Extensible Markup Language (XML) entworfen. Dabei wird eine DTD-Erweiterung definiert, die zur Spezifikation von schematischen Restriktionen verwendet wird.

Das entworfene XML-basierte Modell erlaubt sowohl die Beschreibung interner Strukturen als auch globaler Zusammenhänge zwischen Objekten. Eine Anfragesprache stellt übersichtliche und einfache Suchoperationen für multimediale Repositorien zur Verfügung. Insbesondere können in diesem Modell Objekte um Metainformationen angereichert werden. Das Modell liefert darüber hinaus Mechanismen zur einheitlichen Beschreibung von Objekten und Hierarchien.

Die entwickelten Konzepte wurden in einem Medienarchiv für die Medizinische Universität zu Lübeck prototypisch implementiert. Dieses Archiv steht einer breiten Benutzerschicht zur Verfügung und erfährt im Universitätsumfeld eine wachsende Bedeutung.

6.1 Ergebnisse

Die vorliegende Arbeit stellt dar, wie Objekte innerhalb eines multimedialen Archivs für Forschung und Lehre mittels eines XML-basierten Modells in flexibler und ausbaufähiger Weise beschrieben werden können. Es wurde gezeigt, daß sich die Verwendung eines Modells auf der Basis von XML mit den Anforderungen eines Medienarchivs ergänzt. Der entwickelte Ansatz begründet sich dabei auf der theoretischen Untersuchung diverser formalsprachlicher Eigenschaften. Ausgehend von diesen Untersuchungen werden Erweiterungen definiert, die formalsprachliche Nachteile überwinden.

Die Plausibilität der entwickelten Ansätze und Konzepte wurde in der prototypischen Implementierung verifiziert. Zur Zeit befindet sich das System in der Einführungsphase. Die Erfahrungen, die sich nach der Einführungsphase und der intensiven Nutzung des Systems durch Institute und Kliniken der Medizinischen Universität zu Lübeck ergeben, sollten sich in entsprechenden Erweiterungen und Verbesserungen des Modells und des Systems niederschlagen.

Die Arbeit zeigt insbesondere, dass herkömmliche Datenmodelle mit ihren festen Strukturen einen zu starren Ansatz für die Modellierung von Objekten im Kontext einer multimedialen Anwendung darstellen. XML stellt hier eine flexible und vielversprechende Alternative dar.

6.2 Ausblick

Sowohl das entworfene Modell als auch der realisierte Prototyp sind offen für weitere forschungsrelevante Arbeiten. So wäre eine Anwendung der entwickelten Ideen auch in anderen Szenarien denkbar. Beispielsweise stellt der Einsatz für eBusiness-Lösungen und somit die Nutzung für verteilte Geschäftsprozesse und -datenrepositorien ein interessantes Gebiet dar.

In der Zukunft ist es von Bedeutung, durch ein allgemeines Vererbungskonzept Mechanismen zu schaffen, die es erlauben, Informationen zwischen Objekten zu transportieren. Mit einem solchen Vererbungsansatz ließe sich eine Vielzahl expliziter Attributierungen durch Benutzer oder aber Extraktionstools vermeiden. Auch könnten Metainformationen über den derzeit realisierbaren Rahmen hinaus gewonnen werden.

In diesem Zusammenhang würde eine allgemeine, auf Standards zur Metadatenbeschreibung basierende Schnittstelle, es anderen Anwendungen erlauben, die Daten des Medienarchivs zu nutzen. Somit könnten Fremdanwender und -applikationen von den gewonnenen Informationsrepositorien profitieren und ohne propriätere Konvertierungen das Datenmaterial direkt verarbeiten.

Die benutzte Anfrageschnittstelle besitzt bisher keine tiefergehenden Optimierungsstrategien. Dieses stellt noch einen Nachteil gegenüber herkömmlichen Datenbanksystemen dar, da diese eine anwendungsunabhängige Optimierung unterstützen. Hier wäre es wichtig, entsprechende Konzepte zu entwickeln und die Anfragesprache zu integrieren. Diese Problematik stellt einen über das Medienarchiv hinausgehenden Engpass für die effiziente Verwaltung von XML-Daten dar und ist aus Forschungssicht eine aktuelle Herausforderung.

Auch sind die implementierten Anfrageoperationen relativ simpel gehalten. Hier wäre es wichtig, weitere Funktionalitäten (z. B. wie sie im Information Retrieval verwendet werden) zu integrieren.

Auf praktischer Ebene wäre es daneben wichtig, ein objektübergreifendes Transaktionskonzept zu konzipieren. Dieses wird um so notwendiger, je weiter die Zahl der Benutzer ansteigt.

A

Die Schnittstelle `MDMObject`

public interface **MDMObject**

Die Klasse `MDMObject` repräsentiert die im Medienarchiv existierenden Objekte. Ein `MDMObject` kann eine Datei, ein Verzeichnis oder eine Verknüpfung darstellen. Ein Verzeichnis kann mehrere Unterobjekte besitzen, welches wiederum `MDMObjects` sind.

Ein Objekt besteht aus mehreren Inhalten, den sogenannten *Attachments*. Es gibt ein Haupt-Attachment mit dem Namen "object". Dieses stellt den eigentlichen Inhalt des Objekts dar. Alle weiteren Attachments sind Annotierungen des Haupt-Attachment. Diese Attachments können andere Versionen oder Formate des Objekts sein. Das Attachment "thumbnail" ist z. B. ein kleines Bild (100x100 Pixel), dass das Haupt-Attachment verkleinert wider gibt. Es können so auch andere Dateiformate des gleichen Objekts gespeichert werden.

Eine Verknüpfung (Link) verweist auf ein anderes `MDMObject`. Alle Operationen, die auf dem Link erfolgen, werden auf dessen Zielobjekt durchgeführt.

Public Members

boolean **canRead** () throws RemoteException

Testet, ob der aktuelle Benutzer auf den Inhalt des `MDMObject` zugreifen kann.

boolean **canWrite** () throws RemoteException

Testet, ob der aktuelle Benutzer den Inhalt des `MDMObject` verändern kann.

boolean **checkIn** (String mimetype,
InputStream in) throws RemoteException

Setzt den Inhalt des Attachment "object" und löscht das CheckOut-Flag.

InputStream **checkOut** () throws RemoteException

Gibt den InputStream des Attachment "object" zurück, wenn das CheckOut-Flag nicht gesetzt ist.

MDMObject **copyTo** (String dest) throws RemoteException

Kopiert das `MDMObject` an die Zielposition.

MDMObject **createDirectory** (String name) throws RemoteException

Erstellt ein neues Unterverzeichnis.

MDMObject **createLink** (String name) throws RemoteException

Erstellt einen Link, der auf das Ziel "target" zeigt.

boolean **createObjectInfo** () throws RemoteException

	Erzeugt für das Attachment "object" eine Objektbeschreibung (RdfDom) und erstellt ein Thumbnail.
boolean	**delete** () throws RemoteException Löscht das `MDMObject`.
String	**getChilds** () throws RemoteException Gibt eine Liste mit den Namen aller Unterobjekte zurück.
String	**getDate** () throws RemoteException Gibt das Erstellungsdatum zurück.
String[]	**getFormats** () throws RemoteException Gibt alle Formate an, in die das Attachment "object" konvertiert werden kann.
String	**getFullPath** () throws RemoteException Ein `MDMObject` wird durch die Namens- und Pfadangabe eindeutig beschrieben.
String	**getGroup** () throws RemoteException Gibt die Gruppe des `MDMObject` zurück.
InputStream	**getInputStream** (String attachment) throws RemoteException Gibt den Java Eingabestrom des Attachment zurück.
String	**getMimeType** (String attachment) throws RemoteException Gibt den MimeType eines Attachment an.
String	**getMode** () throws RemoteException Gibt die Zugriffsrechte für das `MDMObject` zurück.
String	**getName** () throws RemoteException Gibt den Namen des `MDMObject` zurück.
MDMObject	**getObject** (String path) throws RemoteException Gibt das `MDMObject` mit dem relativen Pfad "path", ausgehend vom aktuellen Objekt zurück.
String	**getOwner** () throws RemoteException Gibt den Besitzer des `MDMObject` zurück.
MDMObject	**getParent** () throws RemoteException Gibt das Vaterobjekt des aktuellen `MDMObject` zurück.
RdfDom	**getRdfDom** () throws RemoteException Gibt eine Beschreibungsobjekt der Klasse RdfDom zurück.

long	**getSize** () throws RemoteException Gibt die Größe des Hauptobjekts zurück.
String	**getTarget** () throws RemoteException Gibt bei einem Link dessen Ziel an.
boolean	**isDirectory** () throws RemoteException Testet, ob es sich bei dem Objekt um ein Verzeichnis handelt.
boolean	**isFile** () throws RemoteException Testet, ob es sich bei dem Objekt um eine Datei handelt.
boolean	**isLink** () throws RemoteException Testet, ob es sich bei diesem Objekt um einen Link handelt.
boolean	**isRootDir** () throws RemoteException Testet, ob es sich bei diesem Objekt um das Wurzelverzeichnis handelt.
MDMObject	**moveTo** (String dest) throws RemoteException Verschiebt das Objekt an die angegebene Position.
boolean	**setGroup** (String groupid, boolean deep) throws RemoteException Setzt die zugehörige Gruppe des Objekts.
boolean	**setInputStream** (String attachment, String mimetype, InputStream in) throws RemoteException Fügt dem Objekt ein Attachment "attachment" zu bzw. überschreibt dieses.
boolean	**setLinkTarget** (String target) throws RemoteException Ist das Objekt ein Link, so setzt es dessen Ziel.
boolean	**setMode** (String mode, boolean deep) throws RemoteException Setzt die Zugriffsrechte des Objekts, gegebenenfalls rekursiv ("deep").
boolean	**setOwner** (String userid, boolean deep) throws RemoteException Setzt den Besitzer des Objekts, gegebenenfalls rekursiv ("deep").

B

ExCoT API

public class **FileFormat**

Abstrakte Klasse, welche von einem Filter implementiert werden muss, um diesen im Medienarchiv anmelden zu können. Die neu implementierten Klassen werden über Konfigurationen gespeichert und sind so der ExCoT-Schnittstelle bekannt. Zum Initalisieren eines entsprechenden Objekts dient die statische Methode `getInstance()`. Diese Methode erzeugt ein Objekt der zugehörigen Klasse und liefert dieses zurück.

Public Members

abstract	**FileFormat** () throws FileFormatException Konstruktor, der in jeder erbenden Klasse implementiert werden muss.
static FileFormat	**getInstance** (InputStream in, String mimeType) throws FileFormatException Liefert ein Objekt einer von FileFormat erbenden Klasse durch Aufruf deren Konstruktors, wobei die Klasse den MimeType "mimeType" verarbeiten kann. Das Medienobjekt wird innerhalb des Eingabestroms "in" übergeben.
RdfDom	**getRdfDom** () throws FileFormatException Liefert ein Objekt zurück, welches die erzeugte Beschreibung enthält. Für die Transformationen können optional Stringparameter ("args") übergeben werden.
InputStream	**getFormat** (String format, String[] args) throws FileFormatException Gibt einen Eingabestrom zurück, welcher das Objekt im entsprechenden Format liefert.
String[]	**getFormats** () throws FileFormatException Gibt die Namen alle erzeugbaren Transformationsformate des Objekts zurück.

C

MDM API Packages

Package	Klasse	Funktion des Package / der Klasse
MDM.query		Package für Anfrageverarbeitung
	MDMQueryImpl	Implementierung der Suche
	SuchstringParser	Vom JavaCC erzeugte Klasse zum Parsen des Suchausdrucks und zum Umsetzen in einen XQL-Ausdruck
MDM.remote		Package mit Schnittstellendefinitionen
	MDMGroup	Definiert die Methoden einer Gruppe innerhalb des MDM (s. a. MDMUser)
	MDMObject	Definiert die Methoden für die zentralen Funktionen eines Objekts des Medienarchivs
	MDMSession	Startpunkt bei der Verbindung mit dem Archiv
	MDMUser	Definiert die Methoden eines Benutzers, insbesondere für dessen Account innerhalb des MDM (s. a. MDMGroup)
	MDMQuery	Definiert die Methoden und Parameter der Suche
	RdfDom	Verwaltet die Eigenschaften eines Objekts
	MDMClientSessionFactory	Bootstraping-Klasse für MDMSession
MDM.tools		Allgemeine Helferklassen
	BasicRdfDom	Helferklasse zur Manipulation von Objekteigenschaften
	DocumentTools	Hilfsmethoden für XML- und HTTP-spezifische Umwandlungen
	DOMHelper	Helferklasse zum Arbeiten auf DOM-Bäumen
	Log	Klasse zur Verwaltung der einzelnen Logdateien
	Mode	Wrapperklasse für die Manipulation von Objektrechten
	MultiPartInputStream	Helferklasse für die Verarbeitung von HTTP-Uploads
	PDocumentCache	Virtueller Cache für die zentrale Steuerung der PDOM-Stores
	XMLEncoder	Helferklasse für DOM-basierte Kodierung von XML-Dokumenten

MDM.xmlfs		Serverseitige Klassen und Implementierungen der Schnittstellen
	HttpServer	Miniserver für den beschleunigten Austausch von Objekten über HTTP
	Locator	Klasse, die der Lokation von Objekten im Archiv dient
	MDMObjectImpl	Implementierung des zugehörigen Interface
	MDMServerImpl	Implementierung des zugehörigen Interface
	MDMSessionImpl	Implementierung des zugehörigen Interface
	MDMUserImpl	Implementierung des zugehörigen Interface
	ObjectServer	Serverdienst, der dem Austausch von Objekten zwischen RMI-Client und Server dient
	RdfDomImpl	Implementierung des zugehörigen Interface
	RdfStore	Klasse zur Verwaltung der einzelnen XML-Stores und zur Modellierung der Objekteigenschaften
	XMLFSObject	Serverseitige Implementierung der MDM-Logik unter Verwendung obiger Klassen

Literaturverzeichnis

[Abi97] ABITEBOUL, S.: Querying Semi-Structured Data. In: International Conference on Database Theory (ICDT), Seiten 1–18, Delphi, 1997.

[Abi99] ABITEBOUL, S.: On Views and XML. In: ACM Symposium on Principles of Database Systems (PODS), Seiten 1–9, 1999. Invited talk.

[ABS00] ABITEBOUL, S., P. BUNEMAN und D. SUCIU: Data on the Web. Morgan Kaufman, 2000.

[ACC+97] ABITEBOUL, S., S. CLUET, V. CHRISTOPHIDES, T. MILO, G. MOERKOTTE und J. SIMÉON: Querying Documents in Object Databases. Journal on Digital Libraries, 1(1):5–19, 1997.

[ACM93] ABITEBOUL, S., S. CLUET und T. MILO: Querying and Updating the File. In: International Conference on Very Large Data Bases (VLDB), Seiten 73–84, Dublin, 1993.

[ACM95] ABITEBOUL, S., S. CLUET und T. MILO: A Database Interface for File Updates. In: ACM International Conference on Management of Data (SIGMOD), Seiten 386–397, San Jose, Mai 1995.

[ACM97] ABITEBOUL, S., S. CLUET und T. MILO: Correspondence and Translation for Heterogenous Data. In: International Conference on Database Theory (ICDT), Seiten 351–363, Delphi, 1997.

[ACM98] ABITEBOUL, S., S. CLUET und T. MILO: A Logical View of Structured Files. VLDB Journal, 7(2):96–114, 1998.

[Ado] Adobe System Incorp. `http://www.adobe.com`.

[AG00] ARNOLD, K. und J. GOSLING: The Java Programming Language. Addison-Wesley, 3. Auflage, 2000.

[AM98] AROCENA, G. und A. MENDELZON: WebOQL: Restructuring Documents, Databases, and Webs. In: International Conference on Data Engineering (ICDE), Seiten 24–33, Orlando, 1998.

[AMM97] ATZENI, P., G. MECCA und P. MERIALDO: To Weave the Web. In: International Conference on Very Large Data Bases (VLDB), Seiten 206–215, Athen, 1997.

[AQM+97] ABITEBOUL, S., D. QUASS, J. MCHUGH, J. WIDOM und J.L. WIENER: The Lorel Query Language for Semistructured Data. Journal on Digital Libraries, 1(1):68–88, April 1997.

[BBG88] BALLANCE, R., J. BUTCHER und S. GRAHAM: Grammatical Abstraction and Incremental Syntax Analysis in a Language-Based Editor. In: ACM Conference on Programming Language Design and Implementation (SIGPLAN), Band 23, Seiten 185–198, Juni 1988.

[BBLL99] BEHRENS, R., G. BUNTROCK, C. LECON und V. LINNEMANN: Is XML really enough? Technischer Bericht Institut für Informationssysteme, Med. Univ. Lübeck, 1999. Schriftenreihe der Institute für Informatik/Mathematik, A-99-11; unter `http://www.ifis.mu-luebeck.de/public/`.

[BDHS96] BUNEMAN, P., S. DAVIDSON, G. HILLEBRAND und D. SUCIU: A Query Language and Optimization Techniques for Unstructured Data. SIGMOD RECORD, 25(2):505–516, Juni 1996.

[BDLM96] BOLES, D., K. DREGER, M. GROSS, ST. LOHRUM und D. MENGE: MeDoc: Architektur und Funktionalitä des MeDoc-Dienstes. Technischer Bericht MeDoc-Interne Projektbibliothek, 1996.

[BEGO71] BOOK, R., S. EVEN, S. GREIBACH und G. OTT: Ambiguity in Graphs and Expressions. IEEE Transactions on Computers, 20:149–153, 1971.

[Beh97] BEHRENS, R.: Ein Ansatz zur Auswertung von Anfragen in objektorientierten Datenbanken. Diplomarbeit, Universität Hildesheim, 1997.

[Beh00a] BEHRENS, R.: A Grammar Based Model for XML Schema Integration. In: British National Conference on Databases (BNCOD), Seiten 172–190, 2000.

[Beh00b] BEHRENS, R.: MONTANA: Towards a Web-based infrastructure to improve lecture and research in a university environment. In: 2nd International Workshop on Advanced Issues of E-Commerce and Web-based Information Systems, Seiten 58–66, San Jose, 2000.

[Beh00c] BEHRENS, R.: On the Complexity of standard and specialized DTD Parsing. In: 12. GI-Workshop "Grundlagen von Datenbanken", Seiten 6–10, 2000.

[Bis95] BISKUP, J.: Grundlagen von Informationssystemen. Vieweg Verlag, 1995.

[BKPD95] BITTORF, A., N.C. KREJCI-PAPA und T.L. DIEPGEN: Development of a dermatological image atlas with worldwide access for the continuing education of physicians. Journal of Telemedicine and Telecare, Seiten 45–53, 1995.

[BKW98a] BRÜGGEMANN-KLEIN, A. und D. WOOD: One-Unambiguous Regular Languages. Information and Computation, 140(2):229–253, 1998.

[BKW98b] BRÜGGEMANN-KLEIN, A. und D. WOOD: Regular tree languages over non-ranked alphabets. Working paper, 1998.

[BL00] BEHRENS, R. und V. LINNEMANN: XML-basierte Informationsmodellierung am Beispiel eines Medienarchivs für die Lehre. Technical Report A-00-2, Med. Universität zu Lübeck, 2000.

[BLC] BERNERS-LEE, T. und D. CONNOLLY: Hypertext markup language - 4.0. `http://www.w3.org/ hypertext/WWW/MarkUp/MarkUp.html`.

[BLFM] BERNERS-LEE, T., R. FIELDING und L. MASINTER: RFC2396: Uniform Resource Identifiers (URI). `http://www.ietf.org/rfc/rfc2396.txt`.

[BLLS98] BEHRENS, R., C. LECON, V. LINNEMANN und O. SCHMITT: MONTANA: Ein Medienarchiv für die Lehre. Technischer Bericht Institut für Informationssysteme, Med. Univ. Lübeck, September 1998. Schriftenreihe der Institute für Informatik/Mathematik, A-98-19; unter `http://www.ifis.mu-luebeck.de/public/`.

[BMB95] BERRUT, C., P. MULHEM und P. BOUCHON: Modeling and Indexing Medical Images: the RIME approach. In: KUHLEN, R. und M. RITTBERGER (Herausgeber): HIM'95 (Hypertext - Information Retrieval - Multimedia), Seiten 105–115. Universitätsverlag Konstanz, 1995.

[Bos97] BOSAK, J.: XML, Java and the future of the Web. Technischer Bericht SUN, 1997.

[Bro95] BROY, M.: Informatik-Eine grundlegende Einführung, Band 4. Springer-Verlag, 1995.

[Bun97] BUNEMAN, P.: Semistructured data. In: ACM Symposium on Principles of Database Systems (PODS), Seiten 117–121, 1997.

[BW97] BOLES, D. und G. WÜTHERICH: Transformationelle Multimedia-Softwareentwicklung. In: HIM'97 (Hypertext - Information Retrieval - Multimedia), Seiten 95–108. Universitätsverlag Konstanz, 1997.

[CACS94] CHRISTOPHIDES, V., S. ABITEBOUL, S. CLUET und M. SCHOLL: From Structured Documents to Novel Query Facilities. In: ACM International Conference on Management of Data (SIGMOD), Seiten 313–323, Minneapolis, 1994.

[Cat93] CATELL, R.: The Object Database Standard ODMG-93. Morgan Kaufman, 1993.

[Cat97] CATELL, R.: The Object Database Standard 2.0. Morgan Kaufman, 1997.

[CCM96] CHRISTOPHIDES, V., S. CLUET und G. MOERKOTTE: Evaluating Queries with Generalized Path Expressions. In: ACM International Conference on Management of Data (SIGMOD), Seiten 413–422, 1996.

[CCS00] CHRISTOPHIDES, V., S. CLUET und J. SIMEON: On Wrapping Query Languages and Efficient XML Integration. In: ACM International Conference on Management of Data (SIGMOD), Seiten 141–152, 2000.

[CDG+97] COMON, H., M. DAUCHET, R. GILLERON, F. JACQUEMARD, D. LUGIEZ, S. TISON und M. TIMMASI: Tree Automata - Techniques and Applications. unpublished book, 1997. unter http://www.grappa.univ-lille3.fr/tata/.

[CDSS98] CLUET, S., C. DELOBEL, J. SIMÉON und K. SMAGA: Your Mediators Need Data Conversion! In: ACM International Conference on Management of Data (SIGMOD), Seiten 177–188, Seattle, 1998.

[CG88] CAMPBELL, B. und J. GOODMAN: HAM: A General Purpose Hypertext Abstract Machine. Communications of the ACM, 13(7):856–861, 1988.

[Che76] CHEN, P.: The Entity Relationship Model: Towards a Unified View of Data. ACM Transactions on Database Systems, 1(1):9–36, 1976.

[Chi97] CHIARAMELLA, Y.: Browsing and Querying: two complementary approaches for Multimedia Information Retrieval. In: FUHR, N., G. DITTRICH und K. TOCHTERMANN (Herausgeber): Hypertext - Information Retrieval - Multimedia (HIM), Seiten 9–26. Universitätsverlag Konstanz, 1997.

[CLMY93] COCHINWALA, M., K. LEE, W. MANSFIELD und M. YU: A Distributed Transaction Monitor. In: Research Issues in Data Engineering (RIDE), Seiten 268–272, 1993.

[Clu97] CLUET, S.: Modeling and Querying Semi-structured Data. In: International Summer School on Information Extraction (SCIE), Band 1299 der Reihe Lecture Notes in Computer Science, Seiten 192–213. Springer-Verlag, 1997.

[CM95] CONSENS, M. und T. MILO: Algebras for querying text regions. In: ACM Symposium on Principles of Database Systems (PODS), Seiten 11–22, 1995.

[CMN97] CAREY, M., N. MATTOS und A. NORI: Object-Relational Database Systems: Principles, Products, and Challenges. In: PECKHAM, JOAN (Herausgeber): ACM International Conference on Management of Data (SIGMOD), Seite 502, 1997.

[Cod70] CODD, E. F.: A Relational Model of Data for Large Shared Data Banks. Communications of the ACM, 13(6):377–387, 1970.

[Cor] CORPORATION, MICROSOFT: Microsoft Word. http://www.microsoft.com/ word.

[Cor97] CORPORATION, IBM: DataLinks: Managing External Data with DB2 Universal Database, 1997.

[DBHDV94] DE BRA, P., G.-J. HOUBEN und J. DE VOCHT: Retrieval of Hypertext Structures. Technischer Bericht Eindhoven University of Technology and University of Toronto, 1994.

[DCM99] Dublin Core Metadata Initiative. DCMI Homepage , http://purl.org/dc/,1999.

[DFF+98] DEUTSCH, A., M. FERNANDEZ, D. FLORECSU, A. LEVY und D. SUCIU: XML-QL: A Query Language for XML. In: QL98 - The Query Languages Workshop. W3C, 1998.

[DFF+99] DEUTSCH, A., M. FERNANDEZ, D. FLORESCU, A. LEVY, D. MAIER und D. SUCIU: Querying XML Data. IEEE Data Engineering Bulletin, 22(3):10–18, 1999.

[Dre95] DREGER, M.: MeDoc: Teilprojekt3 - Architektur und Kommunikation. Technischer Bericht MeDoc-Interne Projektbibliothek, 1995.

[eC] CORPORATION EXCELON: eXcelon Data Server. `http://www.exceloncorp.com`.

[Eng97] ENGLANDER, R.: Developing Java Beans. O'Reilly, 1997.

[FB] FREED, N. und N. BORENSTEIN: RFC2046: Multipurpose Internet Mail Extensions (MIME). `http://andrew2.andrew.cmu.edu/rfc/rfc2046.html`.

[Fer96] FERBER, R.: Information Retrieval. `http://www-cui.darmstadt.gmd.de/~ ferber/ir-bb/book_1.html, IPSI, GMD Darmstadt`, 1996.

[FFK+97] FERNANDEZ, M., D. FLORESCU, J. KANG, A. LEVY und D. SUCIU: STRUDEL - A Web site management system. In: ACM International Conference on Management of Data (SIGMOD), Tucson, 1997.

[Fuh96] FUHR, N.: Object-oriented and Database Concepts for the Design of Networked Information Retrieval Systems. Technischer Bericht University of Dortmund, 1996.

[Fuh00] FUHRBERG, K.: Internet-Sicherheit. Carl Hanser Verlag, 2000.

[FW93] FENG, A. und T. WAKAYAMA: SIMON: A grammar-based transformation system for structured documents. In: International Conference on Electronic Publishing (EP), Seiten 361–372, 1993.

[GHJV95] GAMMA, E., R. HELM, R. JOHNSON und J. VLISSIDES: Design Patterns: elements of reusable object-oriented software. Addison-Wesley, 1995.

[GJ79] GAREY, M. und D. JOHNSON: Computers and Intractibility, a Guide to the Theory of NP-Completeness. W.H. Freeman and Co., San Francisco, 1979.

[GLS98] GUAN, T., M. LIU und L.V. SAXTON: Structure-Based Queries over the World Wide Web. In: LING, T.W., S. RAM und M.L. LEE (Herausgeber): Conceptual Modeling - ER'98, Band 1507 der Reihe Lecture Notes in Computer Science, Seiten 107–120. Springer-Verlag, 1998.

[GMQR+97] GARCIA-MOLINA, H., D. QUASS, A. RAJARMAN, Y. SAGIV, J. ULLMAN, V. VASSALOS und J. WIDOM: The TSIMMIS Approach to Mediation: Data Models and Languages. Journal of Intelligent Information Systems, 8:117–132, 1997.

[GMW99] GOLDMAN, R., J. MCHUGH und J. WIDOM: From Semistructured Data to XML: Migrating the Lore Data Model and Query Language. In: International Workshop on the Web and Databases (WebDB), Seiten 25–30, June 1999.

[Gol90] GOLDFARB, C.F.: The SGML Handbook. Oxford University Press, 1990.

[GPVG89] GYSSENS, M., J. PAREDAENS und D. VAN GUCHT: A Grammar-Based Approach Towards Unifying Hierachical Data Models. In: ACM International Conference on Management of Data (SIGMOD), Band 18, Seiten 263–272, Portland, 1989.

[Gro] GROUP, OBJECT MANAGEMENT: Common Object Broking Architecture (CORBA). `http://www.corba.org`.

[GS96] GÉCSEG, F. und M. STEINBY: Handbook of Formal Languages - Beyond Words, Band 3, Seiten 1–61. Springer-Verlag, 1996. In G. Rozenberg and A. Salomaa, editors.

[GT87] GONNET, G. und F. TOMPA: Mind Your Grammar: A new Approach to Modeling Text. International Conference on Very Large Data Bases (VLDB), Seiten 339–346, 1987.

[Gün98] GÜNTHER, T.: Datenmodellierung fü HTML-Dokumente. Institut für Informationssysteme, Medizinische Universität Lübeck, Januar 1998. Studienarbeit.

[GW97] GOLDMAN, R. und J. WIDOM: DataGuides: Enabling Query Formulation and Optimazation in Semistructured Databases. In: International Conference on Very Large Data Bases (VLDB), Seiten 436–445, August 1997.

[Her95] HEROLD, H.: lex und yacc. Addison-Wesley, 2. Auflage, 1995.

[HMF99] HUCK, G., I. MACHERIUS und P. FANKHAUSER: PDOM: Lightweight Persistency Support for the Document Object Model. In: Proceedings of the OOPSLA Workshop "Java and Databases: Persistence Options", 1999.

[HS94] HALASZ, F. und M. SCHWARTZ: The Dexter Hypertext Reference Model. Communications of the ACM, 37(2):30–39, Februar 1994.

[HSP97] Projektantrag HSP3: Schwerpunkt multimediale Kooperation an der Medizinischen Universität zu Lübeck, 1997.

[HU94] HOPCROFT, J. und J. ULLMAN: Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie. Addison-Wesley, 3. Auflage, 1994.

[Inf] INFORMIX. `http://www.informix.com/`.

[ISO86] ISO: Information Processing-Text and Office Systems - Standard Generalized Markup Language (SGML), 1986.

[ISO93] ISO/IEC: Information technology - Open Document Architecture (ODA) and Interchange Format - Temporal relationship and non-linear structures, 1993.

[Java] JAVASOFT: Enterprise Java Beans. `http://www.javasoft.com/ejb`.

[Javb] JAVASOFT: Javasoft Homepage. `http://www.javasoft.com/`.

[Javc] JAVASOFT: JAXP Homepage. `http://www.javasoft.com/xml/`.

[Kem00] KEMPA, M: Dokumentenmodell für XML-basierte World-Wide-Web-Anwendungen: V-DOM. In: 12. GI-Workshop "Grundlagen von Datenbanken", 2000.

[Knu68] KNUTH, D.: Semantics of Context-Free Languages. Mathematical Systems Theory, Seiten 127–145, 1968.

[Knu73] KNUTH, D.: Fundamental Algorithms, Band 1. Addison-Wesley, 2. Auflage, 1973.

[Kra00] KRAMER, J.: Konzeption und Implementierung einer WAP-basierten Benutzerschnittstelle für das Medienarchiv MONTANA. Institut für Informationssysteme, Medizinische Universität Lübeck, 2000. Studienarbeit, in Bearbeitung.

[Kre00] KREUTZBERGER, M.: Konzeption und Implementierung eines Parsers auf Basis einer XML Erweiterung. Institut für Informationssysteme, Medizinische Universität Lübeck, 2000. Studienarbeit.

[KS97] KONOPNICKI, D. und O. SHMUELI: Information Gathering in the World Wide Web: The W3QL Query Language and the W3QS System. In: International Conference on Data Engineering (ICDE), 1997.

[KW95] KADHIM, B. und W. WAITE: Maptool - Mapping Between Concrete and Abstract Syntaxes. Technischer Bericht Unversity of Colorado at Boulder, Februar 1995.

[Lam86] LAMPORT, L.: LaTeX: A Document Preparation System. Addison-Wesley, 1986.

[LB96] LIE, H.W. und B. BOS: Cascading Style Sheets, level 1. `http://www.w3.org/TR/REC-CSS1-96121`, 1996.

[Li98] LI, L.: Java - Data Structures and Programming. Springer-Verlag, 1998.

[Lin79] LINNEMANN, V.: Sprachelemente zur Generierung und Umformung syntaktischer Strukturen auf der Basis von ALGOL - 68 und deren theoretische Untersuchung. Doktorarbeit, Technische Universität Braunschweig, 1979.

[Lin93] LINNEMANN, V.: Grammatiken und Syntaxbä in Datenbanken. In: STUCKY, W. und A. OBERWEIS (Herausgeber): Datenbanksysteme in Büro, Technik und Wissenschaft (BTW), Seiten 403–412. Springer-Verlag, 1993.

[LKD+88] Linnemann, V., K. Küspert, P. Dadam, P. Pistor, R. Erbe, A. Kemper, N. Südkamp, G. Walch und M. Wallrath: Design and Implementation of an Extensible Database Management System Supporting User Defined Data Types and Functions. In: International Conference on Very Large Data Bases (VLDB), Seiten 294–305, 1988.

[LL98] Lecon, C. und V. Linnemann: Klassifikation und Beschreibung von Medienobjekten mit attributierten kontextfreien Grammatiken am Beispiel eines Medienarchivs fü die Lehre. In: Workshop 'Multimedia-Systeme', Juli 1998. http://www.ifis.mu-luebeck.de/public/1998/wp_final.ps.

[LS00] Liefke, H. und D. Suciu: XMILL: An Efficient Compressor for XML Data. In: ACM International Conference on Management of Data (SIGMOD), Seiten 153–164, 2000.

[LSS96] Lakshmanan, L., F. Sadri und I. Subramanian: A declarative language for querying and restructuring the Web. In: IEEE (Herausgeber): Sixth International Workshop on Research Issues in Data Engineering (RIDE), Seiten 12–21, 1996.

[Ltd] Ltd., Thawte Consulting. http://www.thawte.com/.

[Mai98] Maier, D.: Database desiderata for an XML query language. In: QL98 - The Query Languages Workshop. W3C, 1998.

[Mau96] Maurer, H.: Hyper-G now Hyperwave. Addison-Wesley, 1996.

[MESa] Multimedia-Entwicklungszentrum Schleswig-Holstein (MESH). http://www.mesh.de.

[mesb] Medical Subject Headings (MESH). http://www.nlm.nih.gov/pubs/factsheets/mesh.html.

[Met] Metamata: Java Compiler-Compiler (JavaCC). http://www.metamata.com/javacc/.

[MH00] Monson-Haefel, R.: Enterprise JavaBeans. O'Reilly, 2000.

[MMFB97] Mechkour, M., P. Mulhem, F. Fourel und C. Berrut: PRIME-GC: A Medical Information Retrieval Prototype on the Web. In: 7th Int. Workshop on Research Issues in Data Engineering, 1997.

[MMM96] Mendelzon, A.O., G.A. Mihaila und T. Milo: Querying the World Wide Web. In: IEEE Conference on Parallel and Distributed Information Systems, Seiten 80–91. IEEE, 1996.

[MON] Homepage des Medienarchivs MONTANA der Medizinischen Universitä zu Lü. http://www.medienarchiv.mu-luebeck.de.

[MS99] MILO, T. und D. SUCIU: Type Inference for Queries on Semistructured Data. In: ACM Symposium on Principles of Database Systems (PODS), Seiten 215–236, 1999.

[MSA] Microsoft Access Homepage. `http://www.microsoft.com/office/access/`. Microsoft Coporation.

[Mur95] MURATA, M.: Forest-regular Languages and Tree-regular Languages. Working paper, 1995.

[NAM97] NESTOROV, S., S. ABITEBOUL und R. MOTWANI: Inferring Structure in Semistructured Data. In: International Workshop on Management of Semistructured Data, 1997.

[Neta] NETSCAPE: ECMAScript in a nutshell: Our guide to the new specification. `http://www.netscapeworld.com/netscapeworld/nw-07-1997/nw-07-javascript.html`.

[Netb] Netscape Corporation. `http://www.netscape.com/`.

[Nev98] NEVEN, F.: Structured Document Query Languages Based on Attribute Grammars: Locality and Non-Determinism. In: Seventh Workshop on Foundations of Models and Languages for Data and Objects, 1998.

[Nij80] NIJHOLT, A.: Context-Free Grammars: Covers, Normal Forms and Parsing, Band 93. Lecture Notes in Computer Science, 1980.

[NVdB97] NEVEN, F. und J. VAN DEN BUSSCHE: On Implementing structured document facilities on top of a DOOD. In: International Conference on Deductive and Object-Oriented Databases (DOOD), Seiten 351–367, 1997.

[NVdB98] NEVEN, F. und J. VAN DEN BUSSCHE: Expressiveness of structured document query languages based on attribute grammars. In: ACM Symposium on Principles of Database Systems (PODS), Seiten 11–17, 1998.

[O2W] O2WEB: Building a WWW service on top of the O2 database system. `http://www.o2tech.fr`.

[Obj] OBJECTIVITY. `http://www.objectivity.com/`.

[OMG] Object Management Group (OMG) Homepage. `http://www.omg.org`.

[PAGM96] PAPAKONSTANTINOU, Y., S. ABITEBOUL und H. GARCIA-MOLINA: Object Fusion in Mediator Systems. In: International Conference on Very Large Data Bases (VLDB), Bombay, 1996.

[PGMJ95] PAPAKONSTANTINOU, Y., H. GARCIA-MOLINA und J.WIDOM: Object exchange across heterogenous information sources. In: International Conference on Data Engineering (ICDE), Seiten 251–260, Tapei, Taiwan, März 1995.

[Pie99] PIEMONT, C.: Komponenten in Java. DPunkt Verlag, 1999.

[PV99] PAPAKONSTANTINOU, Y. und P. VELIKHOV: Enhancing Semistructured Data Mediators with Document Type Definitions. In: International Conference on Data Engineering (ICDE), Seiten 136–145, 1999.

[QL998] QL'98 - The Query Languages Workshop. `http://www.w3.org/TandS/QL/QL98/Overview.html`, 1998.

[RLS98] ROBIE, J., J. LAPP und D. SCHACH: XML Query Language (XQL). In: QL98 - The Query Languages Workshop. W3C, 1998.

[Rum94] RUMBAUGH, J.: Modeling Models and Viewing Views: A Look at the Model-View-Controller Framework. Journal of Object-Oriented Programming, 7(2):14–20, 1994.

[Sah00] SAHUGET, A.: Everything You Ever Wanted to Know About DTDs, But Were Afraid to Ask. In: International Workshop on the Web and Databases (WebDB), Seiten 69–74, 2000.

[SC98] SIMÉON, J. und S. CLUET: Using YAT to Build a Web-Server. In: International Workshop on the Web and Databases (WebDB), 1998.

[Sch95] SCHMITZ, L.: Syntaxbasierte Programmierwerkzeuge. Teubner Verlag, 1995.

[SL95] STUTSCHKA, U. und V. LINNEMANN: Attributierte Grammatiken als Werkzeug zur Datenmodellierung. In: LAUSEN, G. (Herausgeber): Datenbanksysteme in Büro, Technik und Wissenschaft (BTW), Seiten 160–178. Springer-Verlag, 1995.

[Sof] SOFTWARE, POET. `http://www.poet.com/`.

[SS86] SCHEK, H.-J. und M. SCHOLL: The relatinoal model with relation-valued attributes. Information Systems, 11(2):137–147, 1986.

[Sto96] STONEBRAKER, M.: Object Relational DBMSs. Morgan Kaufman, 1996.

[Str86] STROUSTRUP, B.: The C++ Programming Language. Addison-Wesley, 1986.

[Suc98] SUCIU, D.: Semistructured Data and XML. In: Proceedings of the 5th International Conference on Foundations of Data Organization and Algorithms (FODO), 1998.

[Sül96] SÜLLOW, K.: AMPHORE - Ein Arbeitsplatz zur Filmdokumentation, Band 47, Seiten 67–74. nfd - Nachrichten für Dokumentation, 1996.

[Tec] TECHNOLOGY, GLOBIT GLOBAL INFORMATION: Infonyte XQL-Suite. `http://www.infonyte.com`.

[Ull88] ULLMAN, J.: Principles of Database and Knowledge-Based Systems. Computer Science Press, 1988.

[Uml95] UMLAND, R.: Objektorientierte Datenbanken. Thomson Publishing, 1995.

[Uni] Unicode Homepage. `http://www.unicode.org`.

[VP00] VIANU, V. und Y. PAPAKONSTANTINOU: DTD inference for views of XML data. In: ACM Symposium on Principles of Database Systems (PODS), 2000.

[W3C] The World Wide Web Consortium (W3C). `http://www.w3.org`.

[W3C98a] eXtensible Markup Language (XML) 1.0. W3C Recommendation, `http://www.w3.org/TR/REC-xml/`, 1998.

[W3C98b] eXtensible Stylesheet Language (XSL) Specification. W3C Working draft , `http://www.w3.org/TR/WD-xsl/`, 1998.

[W3C98c] Document Object Model (DOM). W3C Recommendation, `http://www.w3.org/DOM/`, 1998.

[W3C98d] Document Content Description (DCD) for XML. W3C Note, `http://www.w3.org/TR/NOTE-dcd`, 1998.

[W3C98e] XML-Data. W3C Note, `http://www.w3.org/TR/1998/NOTE-XML-data`, 1998.

[W3C99a] XSL Namespace. W3C Recommendation, `http://www.w3.org/TR/REC-xml-names`, 1999.

[W3C99b] Resource Description Framework (RDF). W3C RDF Interest Group , `http://www.w3.org/RDF/`, 1999.

[W3C99c] XML Path Language (XPath). W3C Recommendation, `http://www.w3.org/TR/xpath`, 1999.

[W3C99d] XSL Transformations. W3C Recommendation, `http://www.w3.org/TR/xslt`, 1999.

[W3C99e] XML Schema Part 1: Structures. W3C Working Draft, `http://www.w3.org/TR/xmlschema-1/`, 1999.

[W3C99f] XML Schema Part 2: Datatypes. W3C Working Draft, `http://www.w3.org/TR/xmlschema-2/`, 1999.

[W3C99g] Namespaces in XML. W3C Recommendation, `http://www.w3.org/TR/REC-xml-names/`, 1999.

[W3C99h] Schema for Object-Oriented XML (SOX). W3C Note, `http://www.w3.org/TR/NOTE-SOX/`, 1999.

[W3C00a] The Extensible HypterText Markup Language (XHTML). W3C Recommendation, `http://www.w3.org/TR/xhtml1/`, 2000.

[W3C00b] XML Linking Language (XLink). W3C Recommendation, http://www.w3.org/TR/xlink, 2000.

[W3C00c] XML Pointer Language (XPointer). W3C Recommendation, http://www.w3.org/TR/xptr, 2000.

[Wm] W3-MSQL. http://www.hughes.com.au/software/w3-msql.htm.

Stichwortregister

Lebenslauf

Name:	Ralf Behrens
Geburtstag:	16. Juli 1974 in Peine, Deutschland
Familienstand:	ledig
Staatsangehörigkeit:	deutsch
Mutter:	Waltraud Behrens, geboren 1937
Vater:	Helmut Behrens, geboren 1939
Geschwister:	Harald Behrens, geboren 1959, Axel Behrens, geboren 1969

Werdegang:

1980-1984	Grundschule Hohenhameln
1984-1986	Orientierungsstufe Hohenhameln
1986-1993	Gymnasium Michelsenschule Hildesheim
1993-1997	Studium der Informatik und der Wirtschafts- mathematik an der Universität Hildesheim
1997-2001	Wissenschaftlicher Mitarbeiter am Institut für Informationssysteme der Medizinischen Universität zu Lübeck

Abitur:

Gymnasium Michelsenschule Hildesheim, Juni 1993.

Diplom:

Akademischer Grad:	Diplom-Informatiker (Dipl.-Inform.) Universität Hildesheim, April 1997.
Diplomarbeit:	„Ein Ansatz zur Auswertung von Anfragen in objektorientierten Datenbanken“
Betreuer:	Prof. Dr. J. Biskup (jetzt Universität Dortmund)

Vordiplom:

Vordiplom in Wirtschaftsmathematik
Diplom in Wirtschaftsmathematik bis auf Diplomarbeit abgeschlossen
Universität Hildesheim, April 1997.