

Elektronisches Publizieren

Document Engineering im World-Wide Web

Prof. Dr. Anne Brüggemann-Klein

Unter Mitwirkung von Dipl.-Inf. Univ. Marouane Sayih

Elektronisches Publizieren: Document Engineering im World-Wide Web

von Prof. Dr. Anne Brüggemann-Klein und Unter Mitwirkung von Dipl.-Inf. Univ. Marouane Sayih

Inhaltsverzeichnis

Zusammenfassung	x
I. Modellierung und Kodierung von Dokumenten	1
1. Einleitung	3
1.1 Motivation und Überblick	3
1.2 Lernziele	4
1.3 Literatur	4
2. Modellierung von Dokumenten und Daten im Web	5
2.1 Orientierung	5
2.2 Um was für Dokumente geht es?	5
2.3 Dokumentenbestandteile	7
2.4 Das Modell der strukturierten Dokumente	8
2.5 Vorteile des Dokumentenmodells	9
2.6 Verfeinerungen des Dokumentenmodells	9
2.7 Strukturdarstellungen	10
2.8 Formalisierung des Dokumentenmodells	11
2.9 Diskussion des Dokumentenmodells	11
2.10 Umsetzungen des Dokumentenmodells	12
2.10.1 Umsetzung mit LaTeX	12
2.10.2 Umsetzung mit Microsoft Word	14
2.11 Das Modell der semistrukturierten Daten	15
2.12 Zusammenfassung und Ausblick	15
3. Zeichenkodierungen	17
3.1 Orientierung	17
3.2 Abstrakter Zeichensatz	19
3.2 Kodetabelle	21
3.4 Kodierungsformat	24
3.5 Kodierungsschema	27
3.6 Metainformation über Kodierungen: MIME	27
3.7 Unicode-Werkzeuge	27
3.8 Jenseits von glattem Text	27
3.9 Elementare Typographie	29
3.10 Schrifttechnologie	29
3.11 Zusammenfassung und Ausblick	29
4. XML	30
4.1 Orientierung	30
4.2 Schlüsseltechnologie XML	30
4.3 XML (Extensible Markup Language) am Beispiel	31
4.4 XML-Syntax: Dokument (Instanz)	32
4.5 Wohlgeformtheit und Validität	34
4.6 Parser	34
4.7 XML-Browser	34
4.8 XML-Editoren	35
4.9 Zusammenfassung	35
5. XML Namespaces	37
5.1 Orientierung	37
5.2 (Konstruiertes!) Beispiel mit Namenskonflikten	37
5.3 Syntax von XML Namespaces	39
5.4 Konformität zu XML Namespaces	40
5.5 Namensräume in der DTD	40
5.6 Zusammenfassung	41
II. Schema-Ansätze für XML	42
6. DTD	43
6.1 Orientierung	43
6.2 Konzepte	43
6.2.1 DTD-Deklaration	44

6.2.2 DTD-Bausteine	45
6.2.3 Regeln beim Entwerfen von DTDs	49
6.3 Zusammenfassung	49
7. XML Schema	50
7.1 Orientierung	50
7.2 Konzepte	50
7.2.1 Elemente	51
7.2.2 Typen	52
7.2.3 Einfache Typen (Datentypen)	53
7.2.4 Komplexe Typen (Strukturtypen)	54
7.2.5 Attribute	55
7.2.6 Vererbung	56
7.2.7 Substitutionsgruppen	58
7.2.8 Abstraktion	59
7.2.9 Verwendung und Rollen	60
7.3 Patterns für XML Schema	61
7.3.1 Russian Doll	61
7.3.2 Salami Slice	62
7.3.3 Venetian Blind	63
7.3.4 Garden of Eden	63
III. Datenmodell und Abfragesprachen in XML	65
8. Baummodell vs. XQuery 1.0 und XPath 2.0 Datenmodell (XDM)	67
1.1 Orientierung	67
1.2 XML-Dokumentarten	67
1.3 Ein Baummodell für XML-Dokumente	67
1.4 XQuery 1.0 und XPath 2.0 Datenmodell (XDM)	70
1.4.1 Knoten	73
1.4.2 Atomic Values	75
1.4.3 Sequenzen	76
9. XPath	78
2.1 Orientierung	78
2.2 XPath-Ausdrücke	79
2.2.1 Schritte	79
2.2.2 Prädikate	81
10. XQuery als Abfragesprache für XML	86
3.1 Orientierung	86
3.2 Input Funktionen	86
3.2 Erzeugen von XML Knoten und Attributen	86
3.2.1 Miteinbezogene Elemente und Attribute aus der Input-Dokument	87
3.2.2 Direkte Element-Konstruktoren	87
3.2.3 Computed Constructors	90
3.3 Verbinden und Restrukturieren von Knoten (FLWOR)	92
3.3.1 For und Let Klausel	93
3.3.2 Where Klausel	96
3.3.3 Order by Klausel	97
3.3.4 Return Klausel	97
3.3.5 Joins	98
3.4 Operatoren und Bedingte Ausdrücke	100
3.4.1 Arithmetische Operatoren	101
3.4.2 Vergleichsoperatoren	101
3.4.3 Sequenz Operatoren	103
3.4.4 Bedingte Ausdrücke	103
3.5 Funktionen	104
3.5.1 Built-In Funktionen	104
3.5.2 Benutzerdefinierte Funktionen	105
3.6 Implementierungen	108
3.6.1 Saxon	108
3.6.2 eXist	109

IV. Verarbeitung von XML-Daten	116
11. XML-Dokumente mit XSLT transformieren	118
4.1 Orientierung	118
4.2 Transformation von XML-Dokumente	118
4.2.1 Cascading Style Sheets	120
4.2.2 XML Style Language Transformation	121
4.2.3 Paradigmen der Informatik	133
4.2.4 XSL Formatting Objects	134
12. Zusammenfassung	139
A. Beispiele	140
B. Appendix: Primitive Datentypen	153
C. Appendix: Built-In Funktionen	154
B. Literaturverzeichnis	155

Abbildungsverzeichnis

3.1. Der Zeichensatz ISO 646-US (US-ASCII)	18
3.2. Die deutsche Variante ISO 646-DE von ISO 646	19
3.3. Der Zeichensatz ISO 8859-1 (ISO Latin-1)	19
3.4. Der Windows-Zeichensatz 1252	20
3.5. Der Unicode-Coderaum mit Surrogatpositionen (S) und privatem Bereich (P)	23
4.1. Drehscheibencharakter von XML	30
7.1. Ein einfaches XML-Dokument	50
7.2. Ein XML Schema für einen einfachen Dichternamen	51
7.3. Ein einfaches XML-Dokument mit Schema-Referenz	51
7.4. Eltern-Kind-Beziehung in einem XML-Dokument	52
7.5. Ein einfacher benutzerdefinierter Datentyp in XML Schema	53
7.6. Ein komplexer Typ in XML Schema	54
7.7. Eine Häufigkeitsaussage auf Elementebene in XML Schema	55
7.8. Ein Strukturtyp mit Auswahl in XML Schema	55
7.9. Attributdeklarationen in XML Schema	56
7.10. Vererbung durch Erweiterung in XML Schema	57
7.11. Vererbung durch Restriktion in XML Schema	57
7.12. Typenkennzeichnung bei der Instanziierung in XML Schema	58
7.13. Substitutionsgruppen in XML Schema	59
7.14. Elemente einer Substitutionsgruppe in einer Instanz	59
7.15. Das Russian Doll XML Schema Design Pattern	61
7.16. Das Salami Slice XML Schema Design Pattern	62
7.17. Das Venetian Blind XML Schema Design Pattern	63
7.18. Das Garden of Eden XML Schema Design Pattern	64
8.1. Gedichtsbaummodell	69
8.2. (De)Serialisierung in XML	69
8.3. Datenmodellkomponenten	73
8.4. Knotenhierarchie	74
8.5. Typhierarchie	76
9.1. Das "Mediathek" Dokument	78
10.1. Computed Konstruktor Syntax	90
10.2. bedingter Ausdruck	103
10.3. Syntax der Funktionsdeklaration	106
11.1. Doe.xml in Mozilla Firefox 3.5	119
11.2. Doe.xml im Browser (mittels CSS formatiert)	121
11.3. XSLT-Prozessor	122
11.4. XSLT: Eine Transformationssprache für XML-Dokumente	122
11.5. Doe.xml im Browser (mittels XSLT formatiert)	124
11.6. Das Konzept der Pipes and Filters	133
11.7. Monolithischer Kernel und Mikrokern, vereinfacht	134
11.8. XSL Workflow	134
11.9. XSL Areas	135
11.10. XSL-FO Workflow	135
A.1. Ein formatiertes Dokument	140
A.2. Eine Dokumentdarstellung mit geschachtelten Boxen	142
A.3. Eine Dokumentdarstellung mit eingerückten Ikonen	143
A.4. Eine Dokumentdarstellung als Baum	144
A.5. Eine Verfeinerung durch Einrückungen und typographische Differenzierung	148
A.6. Eine LaTeX-formatierte Version des Dokuments	151
A.7. Ein Beispieldokument im Word-System	152

Tabellenverzeichnis

3.1. Die Ersetzungen der deutschen Variante von ISO 646	17
3.2. Das Verhältnis von abstrakten Zeichen und Glyphen	20
3.3. Die sechzehn Hex-Ziffern und ihre Repräsentationen als Bitmuster	22
3.4. Koderäume für verschiedene Kodierungen	22
3.5. Kodierungsformate für verschiedene Kodierungen	25
3.6. Schema für UTF8-Kodierung	26
4.1. XML-Entities	34
4.2. Bekannte XML-Editoren	35
6.1. Verkettungs- und Wiederholungsoperatoren	47
6.2. Attributtypen in DTDs	47
6.3. Attributsangabe in DTDs	48
7.1. Einige in XML Schema vordefinierte Datentypen	52
7.2. Verschiedene Verwendungsweisen von XML Schemata und zugehörige Rollen	60
7.3. Zusammenfassung der XML Schema Design Patterns	64
9.1. Achsen	80
10.1. Operatoren für Wertevergleich bzw. generelles Vergleich	101
10.2. Knotenvergleich	102
10.3. Reservierte Funktionsnamen	107
11.1. XSLT-Elemente	131

Beispiele

8.1. gedicht.xml	68
8.2. Auszug aus generierte Mediathek.xml	71
8.3. Auszug aus generierte PlaylistMediathek.xml	72
8.4. Knotenarten und Knotenverwandschaft	74
9.1. Einfache Navigation durch den XML-Baum	79
9.2. Anfrage mit Werteprädikat	82
9.3. Anfrage mit Prädikat für Position	82
9.4. Das Kontext-Item	83
9.5. XML Baum nach oben und nach unten durchlaufen	84
9.6. Vergleich zwischen verschiedene Baumzweige	84
9.7. Element anhand seines Namens finden	85
10.1. Elemente aus dem Input-Dokument	87
10.2. Konstruieren von XML-Elemente mit XML-Syntax	88
10.3. Hinzufügen eines Attributs zu einem Elementen (Ergebnis unvollständig aus Platzgründen)	89
10.4. Entfernen von Kinder-Elemente	90
10.5. Einfacher computed Konstruktor	91
10.6. FLWOR	93
10.7. Syntax der FLWOR- Ausdruck	93
10.8. Syntax der for Klausel	94
10.9. Query mit for Klausel	94
10.10. Syntax der let Klausel	94
10.11. Query mit let Klausel	94
10.12. Query mit for und let Klausel	95
10.13. Query mit for und let Klausel1	95
10.14. For Klausel: mehrere Variablen	96
10.15. Query mit where Klausel	96
10.16. Syntax der order by Klausel	97
10.17. Order by Klausel	97
10.18. Order by Klausel 2	97
10.19. Join	98
10.20. Join mit Prädikat	99
10.21. Outer Join	100
10.22. Implizite Typumwandlung	101
10.23. Der eq Operator	102
10.24. Expliziter cast bei untyped Values	102
10.25. Operator für Reihenfolgevergleich	103
10.26. Der union Operator	103
10.27. Der except Operator	103
10.28. Bedingter Ausdruck innerhalb eines FLWORS	104
10.29. Built-in Funktionen	104
10.30. Die Funktion string	105
10.31. Funktionsdeklaration	106
10.32. Rekursive Funktion	107
10.33. Anlegen einer neuen Collection	110
10.34. Speichern von XML-Dokumente in der Datenbank	111
10.35. Einfügen von neuen Knoten	114
10.36. Löschen eines Musikstücks	115
11.1. Doe.xml	119
11.2. style.css	120
11.3. style.xsl	123
11.4. simplePull.xsl	126
11.5. Ergebnisdokument nach Transformation von Doe.xml mittels simplePull.xsl	127
11.6. simplePush.xsl	128
11.7. import-example.xsl	129

11.8. rekursion.xsl	131
11.9. badge.xsl	136
11.10. badge.fo	137
A.1. Ein unformatiertes Dokument mit reinem Text	141
A.2. Repräsentation der logischen Struktur mit Hilfe von eingebettetem Markup	145
A.3. Ein Dokument im LaTeX-Format	149

Zusammenfassung

Elektronisches Publizieren im ursprünglichen Sinne ist der Prozeß, bei dem Dokumente auf digitalem Wege von ihrer Quelle, den Autorinnen und Autoren, zu ihrem Ziel, den Leserinnen und Lesern, gebracht werden. In einem erweiterten Sinne können die Dokumente auch von Programmen generiert oder auf der Empfangsseite von Programmen weiterverwendet werden. Die Anforderungen an den Publikationsprozeß sind vielfältig und anspruchsvoll:

- Die Dokumente können dynamisch, interaktiv und multimedial sein. Sie werden über verschiedene Publikationskanäle und in verschiedenen Präsentationsformaten angeboten. Die Dokumente müssen an die Nutzungsansprüche in der Zielgruppe individuell anpaßbar sein.
- Die Dokumente werden ihrer Zielgruppe in digitalen Bibliotheken oder in Repositories angeboten; d.h., sie müssen für die verschiedensten Recherche- und Information-Retrieval-Methoden sowie für Informationsdienste erschlossen werden.
- Die Dokumente müssen für menschliche Leserinnen und Leser lesbar sein; sie müssen maschinell und computergestützt weiterverarbeitbar sein; und sie müssen so weit mit semantischer Information angereichert sein, daß sie intelligenten Anwendungen als Informationsgrundlage dienen können.
- Der Publikationsprozeß muß schnell, kostengünstig und zuverlässig funktionieren.

Diese Ansprüche an Flexibilität und Effizienz erfordern ein gut durchdachtes Dokumentenmodell und leistungsfähige Werkzeuge zur Verarbeitung von Dokumenten.

In dieser Vorlesung werden der Stand und die Erfordernisse elektronischen Publizierens, insbesondere des wissenschaftlichen Publizierens, dargestellt und analysiert. Die Vorlesung gibt einen Überblick über Basistechnologie, Werkzeuge und Modelle elektronischen Publizierens, erörtert neue Möglichkeiten insbesondere des Online-Publizierens im Internet und zeigt auf, wo die heutigen Methoden und Werkzeuge zu kurz greifen. Im Zentrum der Darstellung stehen die Extensible Markup Language (XML) und ihre Satellitenstandards.

Teil I. Modellierung und Kodierung von Dokumenten

Inhaltsverzeichnis

1. Einleitung	3
1.1 Motivation und Überblick	3
1.2 Lernziele	4
1.3 Literatur	4
2. Modellierung von Dokumenten und Daten im Web	5
2.1 Orientierung	5
2.2 Um was für Dokumente geht es?	5
2.3 Dokumentenbestandteile	7
2.4 Das Modell der strukturierten Dokumente	8
2.5 Vorteile des Dokumentenmodells	9
2.6 Verfeinerungen des Dokumentenmodells	9
2.7 Strukturdarstellungen	10
2.8 Formalisierung des Dokumentenmodells	11
2.9 Diskussion des Dokumentenmodells	11
2.10 Umsetzungen des Dokumentenmodells	12
2.10.1 Umsetzung mit LaTeX	12
2.10.2 Umsetzung mit Microsoft Word	14
2.11 Das Modell der semistrukturierten Daten	15
2.12 Zusammenfassung und Ausblick	15
3. Zeichenkodierungen	17
3.1 Orientierung	17
3.2 Abstrakter Zeichensatz	19
3.2 Kodetabelle	21
3.4 Kodierungsformat	24
3.5 Kodierungsschema	27
3.6 Metainformation über Kodierungen: MIME	27
3.7 Unicode-Werkzeuge	27
3.8 Jenseits von glattem Text	27
3.9 Elementare Typographie	29
3.10 Schrifttechnologie	29
3.11 Zusammenfassung und Ausblick	29
4. XML	30
4.1 Orientierung	30
4.2 Schlüsseltechnologie XML	30
4.3 XML (Extensible Markup Language) am Beispiel	31
4.4 XML-Syntax: Dokument (Instanz)	32
4.5 Wohlgeformtheit und Validität	34
4.6 Parser	34
4.7 XML-Browser	34
4.8 XML-Editoren	35
4.9 Zusammenfassung	35
5. XML Namespaces	37
5.1 Orientierung	37
5.2 (Konstruiertes!) Beispiel mit Namenskonflikten	37
5.3 Syntax von XML Namespaces	39
5.4 Konformität zu XML Namespaces	40
5.5 Namensräume in der DTD	40
5.6 Zusammenfassung	41

Kapitel 1. Einleitung

1.1 Motivation und Überblick

Noch vor wenigen Jahren gab es nur ein relevantes Dokumentenformat im Web, nämlich die Hypertext Markup Language (HTML). HTML baut auf der Standardized General Markup Language (SGML) auf, die seit 1986 als internationaler Standard vorliegt. Inzwischen ist die Extended Markup Language (XML) an die Seite von HTML getreten. Auch XML leitet sich von SGML ab, aber auf eine Weise, die Dokumente flexibel genug für eine semantische Verarbeitung macht. XML wurde bereits im Februar 1998, zwei Jahre nach den ersten Überlegungen, zum W3C-Standard erhoben. Wegen ihrer Flexibilität ist SGML im Publishing Bereich und in der technischen Dokumentation gut etabliert. Es mangelt jedoch an verfügbaren Werkzeugen, um SGML-Dokumente zu erstellen und zu verarbeiten. Deshalb konnte sich SGML nur in Spezialbereichen durchsetzen. Im Gegensatz zu SGML ist XML speziell für das breite Einsatzgebiet des World-Wide Web entwickelt worden. Dass XML wesentlich einfacher aufgebaut ist als SGML, lässt sich schon am Umfang der Spezifikationen beider Sprachen erkennen: Die XML-Spezifikation umfasst etwa 30 Seiten, die SGML-Spezifikation etwa 500 Seiten. Neben den statisch-passiven Textdokumenten, die klassischen, auf Papier gedruckten Dokumenten ähnlich sind, gewinnen im Internet in zunehmendem Maße dynamisch-aktive Textdokumente an Bedeutung. Sie enthalten aktive Komponenten, mit denen Leserinnen und Leser interagieren können, oder sie ändern dynamisch ihren Inhalt oder ihr Format. Eine moderne, umfassende Definition sieht Dokumente als sprechende Gegenstände¹ ("talking things"), fähig zur exakten Reproduktion, unabhängig von Ort und Zeit. In der Begriffswelt von Leving delegieren Menschen das Sprechen an bestimmte Artefakte, die dann Dokumente heißen und, abhängig von ihrem Genre, genau vereinbarte Funktionen übernehmen können. Qualität und Verlässlichkeit von Dokumenten sind zwei wichtige Aspekte beim "Publizieren" von verbürgtem Wissen. In der Vorlesung "Elektronisches Publizieren" geht es um die organisierte Bereitstellung von Dokumenten im Web. Dabei sind die Dokumente immer digital, was aber nicht nur heißt, dass sie elektronisch zur Verfügung stehen, sondern vor allem auch, dass sie in einer bearbeitbaren Version (aktiv, dynamisch und multimedial)², vorhanden sind. Wir lernen im Laufe dieser Vorlesung Methoden, Modelle und Systeme kennen, um Dokumente im Web verfügbar zu machen. Den funktionalen Aspekt von Dokumenten vertiefen wir anhand des Genres des wissenschaftlichen Aufsatzes. Ein Fokus der Lehrveranstaltung "Elektronisches Publizieren" liegt auf dem Aufbau und den Strukturierungsmöglichkeiten von Dokumenten. Diese werden anhand einer systematischen Einführung in Modelle, Sprachen, Werkzeuge und erprobte Techniken für moderne Dokumenten rund um XML und das Publizieren im World-Wide Web im Grenzbereich von Textdokumenten und Daten erläutert (Teil I). Die Vorlesung gibt aber auch eine Einführung in die Basis-Technologien für Publikationssysteme und Publikationsanwendungen in den Fachgebieten Document Engineering³ und Web Engineering⁴ (Teile II-III).

Das Kapitel 2 setzt sich mit dem Dokumentenbegriff auseinander, der dieser Lehrveranstaltung zugrunde liegt. Für statisch-passive Dokumente wurde schon in den achtziger Jahren das Modell der strukturierten Dokumente entwickelt. Es unterscheidet zwischen textuellem Inhalt, logischer Struktur und graphischem Format. Gesteuert von unabhängigen Formatvorgaben, sogenannten Stylesheets, lässt sich aus Inhalt und logischer Struktur ein graphisches Format generieren.

Moderne Systeme zur Erstellung und Verwaltung von Dokumenten und großen Dokumentenbeständen profitieren vom Modell der strukturierten Dokumente, da es ein hohes Maß an Flexibilität bietet: Die logische Strukturierung unterstützt Anwendungen der Dokumentenverwaltung,

¹Definition von Dokument nach Levy (Siehe hierzu das Buch mit dem Titel "Scrolling Forward. Making Sense of Documents in the Digital Age" von David M. Levy).

²"Aktives Dokument": ermöglicht Interaktionen, automatische Berechnungen, "Dynamisches Dokument": Inhalt zum Nutzungszeitpunkt manipulierbar, "Multimediales Dokument": verwendet verschiedene Inhaltsmedien (Text, Bilder, Videos etc.).

³Document engineering is the computer science discipline that investigates systems for documents in any form and in all media. Document engineering is concerned with principles, tools and processes that improve our ability to create, manage, and maintain documents. [Quelle: <http://www.documentengineering.org> (Aufrufsdatum: 04.03.2010)].

⁴Ausgehend vom Software Engineering umfasst Web Engineering die Anwendung systematischer und quantifizierbarer Ansätze, um Spezifikation, Implementierung, Betrieb und Wartung qualitativ hochwertiger Web-Anwendungen durchführen zu können. [Quelle: <http://www.web-engineering.at/de/kapitel1.pdf> (Aufrufsdatum: 04.03.2010)].

wie die automatische Generierung von Katalogeinträgen und Verzeichnissen. Allgemein gesprochen ist das Modell der strukturierten Dokumente die Basis dafür, dass Textdokumente von Anwendungssystemen als Daten behandelt, automatisiert und computergestützt verarbeitet werden können.

Die Unabhängigkeit der Stylesheets von den Dokumenten ermöglicht auf der einen Seite, dass dasselbe Dokument durch Kombination mit verschiedenen Stylesheets passend zum Anwendungszweck formatiert werden kann. Auf der anderen Seite kann man verschiedene Dokumente mit demselben Stylesheet einheitlich formatieren.

Wir sehen anhand verschiedener Beispiele und Systeme, wie das Modell der strukturierten Dokumente mit Zeichenkodierungen, eingebettetem Markup, Dokumentengrammatiken und Stylesheets in der Praxis umgesetzt wird. Kapitel 2 enthält einige Beispiele für Sprachen und Formate sowie für Editier- und Formatiersysteme.

In Kapitel 3 wird der technische Aspekt des Themas Zeichenkodierungen anhand des Unicode-Standards vertieft.

Moderne Dokumente werden durch ein sogenanntes Modell der strukturierten Dokumente definiert. Eine konkrete Verwirklichung eines solchen Modells, dessen Verwendung eine entscheidende Rolle im Prozess des elektronischen Publizierens spielt, ist XML. Die Vorstellung von XML und sogenannten XML Namespaces⁵ bilden die Schwerpunkte der Kapitel 4 und 5.

1.2 Lernziele

Wichtigstes Lernziel dieser Vorlesung ist einerseits, anhand von den vermittelten Grundlagen und den vorgestellten modernen Möglichkeiten des Web Publishing den Stand der Technik einzuschätzen und sich dabei ein Bild von künftigen Entwicklungen zu machen; andererseits steht aber auch der Erwerb von praktischer Kompetenz und Handlungsfähigkeit im Bereich des Web Publishing im Vordergrund.

Die Vorlesung bietet daher einen detaillierten Überblick über Basistechnologie, Werkzeuge und Modelle elektronischen Publizierens, erörtert die neuen Möglichkeiten insbesondere des Online-Publizierens im World-Wide Web und zeigt auf, wo die heutigen Methoden und Werkzeuge zu kurz greifen. Im Zentrum der Darstellung steht die Extensible Markup Language (XML).

Die Thematik dieser Vorlesung steht im Kontext einer Reihe von Informatikdisziplinen; die wichtigsten sind Document Engineering, Web Engineering, Document Management, Content Management, Software Engineering, Digitale Bibliotheken, Datenbanken und Formale Sprachen.

1.3 Literatur

S. Abiteboul, P. Buneman, D. Suciu: Data on the Web. Morgan Kaufmann Publishers 2000.

T. Berners-Lee u.a.: The Semantic Web. Scientific American 2001. Technologies.

J. Bosak: XML, Java and the Future of the Web.

E.R. Harold u.a.: XML in a Nutshell. O'Reilly 2001.

G. Kappel u.a.: Web Engineering. DPunkt 2003.

J.P. Morgenthal: Portable Data / Portable Code: XML & Java.

D. Levy: Scrolling Forward. Making Sense of Documents in the Digital Age. Arcade Publishing 2001.

⁵XML Namespaces ist ein Konzept, das eine einfache Möglichkeit anbietet, Element- und Attributnamen innerhalb von XML-Dokumenten, eindeutig zu benennen.

Kapitel 2. Modellierung von Dokumenten und Daten im Web

2.1 Orientierung

Dokumentenbegriff (Inhalt, Struktur und Format):

Wir setzen uns zunächst mit dem Dokumentenbegriff auseinander, der diesem Buch zugrunde liegt. Schon in den achtziger Jahren wurde für statisch-passive Dokumente das "Modell der strukturierten Dokumente" entwickelt. Es unterscheidet zwischen textuellem "Inhalt", logischer "Struktur" und graphischem "Format". Dabei berücksichtigt die Quellrepräsentation des Dokuments nur textuellen Inhalt und logische Struktur. Gesteuert von unabhängigen "Formatvorgaben", sogenannten "Stylesheets", läßt sich daraus ein graphisches Format generieren.

Wie bereits in der Einleitung dargestellt, profitieren moderne Systeme zur Erstellung und Verwaltung von Dokumenten und großen Dokumentenbeständen vom Modell der strukturierten Dokumente, da es ein hohes Maß an Flexibilität bietet: Die logische Strukturierung unterstützt Anwendungen der Dokumentenverwaltung wie die automatische Generierung von Katalogeinträgen und Verzeichnissen oder das Information Retrieval nach bestimmten logischen Kriterien und erlaubt es ganz allgemein, Dokumente als Daten zu betrachten, die mit Hilfe von Computersystemen verwaltet und manipuliert werden können.

Die Unabhängigkeit der Stylesheets von den Dokumenten ermöglicht auf der einen Seite, daß ein- und dasselbe Dokument durch Kombination mit verschiedenen Stylesheets passend zum Anwendungszweck formatiert werden kann; auf der anderen Seite kann man verschiedene Dokumente durch Kombination mit ein- und demselben Stylesheet einheitlich formatieren. Auch läßt sich die Struktur beim Erstellen und Lesen von Dokumenten nutzen, etwa zur Navigation und zur Definition von Sichten (Outline-Sicht).

Realisierung des Dokumentenmodells:

Wir sehen in diesem Kapitel an Hand verschiedener Beispiele und Systeme, wie das Modell der strukturierten Dokumente mit Zeichenkodierungen, eingebettetem Markup, Dokumentengrammatiken und Stylesheets in der Praxis umgesetzt wird. Das Kapitel enthält einige Beispiele für Sprachen und Formate sowie für Editier- und Formatiersysteme. Darüber hinaus verschaffen wir uns einen ersten Überblick über weitere, inzwischen auch im Internet einsetzbare Dokumentenmodelle und Formate.

Verwandte Modelle:

Im Datenbankbereich hat sich für Daten im Web das sogenannte Modell der semistrukturierten Daten herausgebildet, das dem Modell der strukturierten Dokumente sehr ähnlich ist. Dieses Kapitel enthält am Schluß eine Gegenüberstellung der beiden Modelle.

2.2 Um was für Dokumente geht es?

Können Sie sich noch an das erste Mal erinnern, als Sie bewußt den Begriff "Dokument" gehört haben? Für mich war das 1985, als ich als frisch promovierte Mathematikerin in einem Forschungsprojekt zur "Be- und Verarbeitung von Dokumenten" anfang zu arbeiten. Bis dahin hatte ich mir unter Dokumenten etwas Wichtiges, Offizielles und Einmaliges vorgestellt, z.B. Reisedokumente, Fahrzeugpapiere oder Personalausweise. Die Bezeichnung "Dokument" für digitale Briefe und andere Schriftstücke, die mit Hilfe des Computers beliebig gestaltet und verändert werden können, kam mir fast frivol vor.

Tatsächlich hat mich mein Sprachgefühl da auf eine interessante Bedeutungsentwicklung hingewiesen. In Brockhaus' Konversationslexikon in der vierzehnten Auflage von 1894 findet sich beispielsweise folgendes unter "Dokument":

„Im weiteren Sinne jeder Gegenstand, welcher dazu dient, die Wahrheit einer zu erweisenden Thatsache, besonders einer für ein Rechtsverhältnis erheblichen Thatsache, zu bestätigen. Im engeren Sinne versteht man darunter Urkunden oder Schriftstücke, im Gegensatz zu anderen körperlichen Beweisstücken, wie Grenzsteinen, Wappen, beschädigten Sachen.“

Ursprünglich waren Dokumente also tatsächlich physische Objekte, Gegenstände, die als einmalige Träger eines in menschliche Sprache gekleideten und schriftlich festgehaltenen Gedankens oder Sachverhalts fungierten. Ein solches Dokument hat eine Kontrolleurin im Sinn, wenn sie einen Schwarzfahrer in der Straßenbahn nach seinem Personalausweis fragt. Ein Dokument neuerer Art, etwa ein sauber getippter Bogen Papier mit Name, Adresse, Personalausweisnummer und weiterer Information zur Person erfüllt einfach nicht denselben Zweck, auch wenn er dieselbe sprachliche Information enthält. Die Dauerhaftigkeit der Schriftform gegenüber der gesprochenen Sprache, die Kommunikation unabhängig von Zeit und Raum ermöglicht, und die Authentizität des Trägermediums, die durch Wasserzeichen, Stempel, Siegel, Barcode oder ähnliches erreicht werden kann, machen das Wesen von Dokumenten in diesem ursprünglichen Sinne aus.

In der Brockhaus Enzyklopädie in der siebzehnten Auflage von 1968 findet sich dagegen unter dem Stichwort "Dokumentation":

„Als Dokumente können alle Unterlagen betrachtet werden, die Informationen beinhalten, also nicht nur publiziertes Wissen, sondern auch Briefe, Akten, Urkunden, Bildsammlungen, Filme u.v.a. “

Die Dokumente neuerer Art sind Informationsträger. Die Schriftform ist immer noch wesentlich, aber das Trägermedium und die Art, wie die Information auf das Trägermedium aufgebracht ist, sind nebensächlich. Auf den sprachlichen Ausdruck kommt es an.

Den Unterschied zwischen dem ursprünglichen und dem neueren Dokumentenbegriff können wir uns an zwei Arten von Delikten klarmachen: Fälschen, z.B. eines Personalausweises oder einer Banknote, beinhaltet immer auch eine Fälschung des Trägermediums und betrifft Dokumente im ursprünglichen Sinne. Kopieren, sei es durch Fotokopieren oder durch Abschreiben, generiert, selbst wenn sich die Erscheinungsform dabei total ändert, immer noch ein zweites Exemplar desselben Dokuments im neueren Sinne; die Umstände, unter denen ein solches Kopieren zulässig ist, regelt das Urhebergesetz, das den sprachlichen Ausdruck schützt.

Das Gemeinsame an Dokumenten ursprünglicher und neuerer Art ist, daß sie durch ihre Schriftform eine sprachliche Mitteilung von der Hörbarkeit in die Sichtbarkeit und von der Flüchtigkeit in die Dauerhaftigkeit umsetzen und somit die Grenzen von Zeit und Raum, die sonst der Senderin und dem Empfänger flüchtiger sprachlicher Mitteilungen gesetzt sind, überschreitet. Der Unterschied liegt darin, wie viele Exemplare es von einem Dokument geben kann und wieviele Leserinnen und Leser gleichzeitig Gebrauch von so einem Dokument machen können.

Eine weitere Unterscheidung, die zu einem dritten, dem modernen Begriff des digitalen Dokuments führt, hängt an der Idee vom Dokument als Informationsträger. Sprachlich formulierte Gedankengänge, Ideen, Fakten oder Aussagen werden erst dann zu Information, wenn sie von jemandem aufgenommen und verarbeitet werden. "Information ist Wissen in Aktion" lautet die Kurzformel der Informationswissenschaft. Und wer ist dieser jemand? Bisher haben wir an menschliche Leserinnen und Leser gedacht. Informationsverarbeitung mit Hilfe von Computerprogrammen führt uns zum modernen Dokumentenbegriff.

Computergestützt können wir Gedankengänge organisieren und Dokumente planen; Gedanken verbalisieren und Dokumente erfassen; Dokumente editieren, formatieren und publizieren (der Öffentlichkeit zugänglich machen); Dokumente ablegen, verwalten, abfragen, zusammenfassen, kombinieren, transformieren, extrahieren; Dokumente lesen, verstehen, memorieren, aus ihnen lernen und über sie nachdenken. Moderne Dokumente sind Informationsträger, die alle diese Funktionen effizient unterstützen. Moderne Dokumente müssen deshalb in ihrer Urform digital sein und je nach Anwendungszweck unterschiedliche Präsentationsmedien unterstützen; man spricht in diesem Zusammenhang von Cross-Media Publishing.

Wir haben uns in diesem Abschnitt ein Bild von der Vielzahl der Dokumentenbegriffe gemacht. Wir werden uns von nun an mit Dokumenten der modernen Art beschäftigen. Dokumente sind für uns also

digitale Informationsträger, deren schriftsprachlich festgehaltene Inhalte sowohl von menschlichen Leserinnen und Lesern als auch von Computerprogrammen aufgenommen und weiterverarbeitet werden können. Im nächsten Abschnitt werden wir analysieren, welche Anforderungen sich daraus für Dokumentenformate ergeben und wie diese Anforderungen durch das Modell der strukturierten Dokumente erfüllt werden.

2.3 Dokumentenbestandteile

Wir haben im vorigen Abschnitt herausgearbeitet, daß Dokumente neuerer oder moderner Art als Informationsträger zu sehen sind. Wie aber kommen die Leserinnen und Leser an die in einem Dokument enthaltene Information? Sehen wir uns doch als Einstieg in diese Frage einmal ein konkretes Dokument an, nämlich ein Konzept der Frauenbeauftragten der Technischen Universität München zu einem Computer-Propädeutikum. Sie finden das Konzept im Anhang (Abbildung A.1, „Ein formatiertes Dokument“) und auf dem Webserver <http://sunschlichter0.informatik.tu-muenchen.de/~brueggem/Vorlesungen/> zu diesem Buch unter "compProp/Konzept.htm" Lesen Sie das Dokument bitte jetzt und legen Sie sich zurecht, welche Information darin enthalten ist.

Wahrscheinlich haben Sie bei der Lektüre u.a. festgestellt, daß das Konzept eine Reihe von Themen vorschlägt, die in einem Kurs des Computer-Propädeutikums behandelt werden können, und was diese Themen sind. Wie genau sind Sie aber an diese Information herangekommen?

Die Ausgangsbasis sind sicherlich die in dem Dokument enthaltenen Buchstaben und Interpunktionszeichen, die Sie beim Lesen zu Wörtern und Sätzen zusammensetzen, intellektuell verarbeiten und mit Bedeutung versehen, d.h. also der "Inhalt".

Es sind aber kaum die Buchstaben und Interpunktionszeichen alleine, die diesen Prozeß ermöglichen und ihn Ihnen in der Tat ganz leicht und selbstverständlich erscheinen lassen. Machen wir dazu den Gegenteilstest: Betrachten Sie die unformatierte Version im Anhang (Beispiel A.1, „Ein unformatiertes Dokument mit reinem Text“) oder unter "compProp/Konzept.txt". Versuchen Sie jetzt, aus dieser Version die Gliederung des Konzepts herauszulesen. Das dürfte sich als mühsam bis unmöglich herausstellen.

Offenbar unterstützt die "Formatierung" des Dokuments—etwa die Wahl der Schriften (groß oder klein, fett oder kursiv), die Ausrichtung von Textpassagen (linksbündig, rechtsbündig oder zentriert) und die Abstände zwischen einzelnen Textpassagen—die Aufnahme und Verarbeitung der Information. Gehen Sie zurück zur zuerst betrachteten formatierten Version des Konzepts. Die Gliederung in die Abschnitte "Hintergrund und Ziele", "Das Kursangebot des Computer-Propädeutikums" usw. ist auf einen Blick klar! Die Formatierung der Überschriften in Fettschrift und mit deutlichen Abständen zum umgebenden Text haben Ihnen ausreichende Hinweise gegeben, was die Gliederungselemente sind.

Die Buchstaben, Interpunktionszeichen und sonstigen Symbole, die in einem Dokument enthalten sind, zusammen mit dem Format, also der typographischen Ausprägung der Symbole und der geometrischen Anordnung oder dem Layout, ermöglichen es den menschlichen Leserinnen und Lesern also, den Informationsgehalt zu erfassen.

Was aber ist genauer das Ergebnis dieses menschlichen Verarbeitungsprozesses? Auf der einen Seite wird die "Aussage" des Dokuments verstanden, auf der anderen Seite und unterstützend zum Verstehen der Aussage wird einzelnen Textpassagen eine Rolle im Gesamttext zugeschrieben. So haben Sie wahrscheinlich beim Betrachten der formatierten Version des Konzepts das Textstück "Ein Computer-Propädeutikum von Studentinnen für Studentinnen" als Überschrift des gesamten Dokuments und das Textstück "Hintergrund und Ziele" als Überschrift eines Abschnitts, der bis zur nächsten Überschrift reicht, identifiziert.

Zusätzlich zur Aussage eines Dokuments erkennen menschliche Leserinnen und Leser also aufgrund des Inhalts und des Formats eine Strukturierung in logische Einheiten. In dem Konzept zum Computer-Propädeutikum ließen sich beispielsweise die folgenden "Strukturelemente" identifizieren: Briefkopf mit Namen der Autorinnen und Angaben zur Institution, Hauptüberschrift, Nebenüberschrift mit

Datums- und Versionsangaben, Abschnitte mit Überschriften, Absätzen und Listen, Geldbeträge, Namen der Organisatorinnen und Datumsangaben.

Die von den Leserinnen und Lesern erkannte Struktur ist nicht eindeutig. Hinweise auf die Struktur ergeben sich aus Inhalt und Format aufgrund von dem semantischen Verständnis der Textpassagen und der kulturellen, durch den Umgang mit zahlreichen Dokumenten ähnlicher Art gewonnenen Erfahrung.

Für menschliche Leserinnen und Leser stellt sich also der Prozeß der Informationsaufnahme und -verarbeitung folgendermaßen dar: Das Dokument präsentiert sich mit Inhalt und Format. Menschliche Leserinnen und Leser extrahieren daraus durch gedankliche Arbeit unter Rückgriff auf ihr Sprachverständnis und ihr kulturelles Wissen um die Verwendung von Formatmerkmalen die Aussage des Dokumentes und konstruieren eine Struktur für die einzelnen Textelemente. Aussage und Struktur bilden dann die Grundlage für die weitere intellektuelle Informationsverarbeitung, im Beispiel etwa das Erkennen einer Frist für eine Kursanmeldung und entsprechendes Handeln.

Wie bringen wir diese Abhängigkeit zwischen Inhalt, Format, Aussage und Struktur nun in Zusammenhang mit unserer früheren Feststellung, daß das Format für neuere und moderne Dokumente eigentlich nebensächlich ist und es nur auf die im Dokument enthaltene Information ankommt?

Dieser scheinbare Widerspruch läßt sich leicht auflösen! In der Tat kann ein- und dasselbe Dokument in ganz unterschiedlichen graphischen Aufmachungen daherkommen, angepaßt an das Präsentationsmedium und die Lesesituation. Beispielsweise kann ein Dokument für das ermüdungsfreie Lesen am Bildschirm mit einer anderen Farbpalette und mit anderen Schriften formatiert sein als für das Lesen von einem Papiausdruck; oder Querverweise können für die Onlineversion als Hypertextlinks und für die Druckversion als Seitenzahlen realisiert werden.

Entscheidend für die "Identität" des Dokumentes ist, daß aus den verschiedenen Formaten immer noch dieselbe Aussage und im wesentlichen dieselbe Struktur erkannt wird.

2.4 Das Modell der strukturierten Dokumente

Bisher haben wir das Erfassen der in einem Dokument enthaltenen Information von der Warte der menschlichen Leserinnen und Leser aus betrachtet. Was ist aber, wenn Computerprogramme zur Informationsverarbeitung herangezogen werden sollen? Wollte man den menschlichen Prozeß des Textverstehens nachbilden, der bei einer formatierten und für das menschliche Wahrnehmungssystem "Auge" realisierten Version des Dokuments startet, so müßte man Computerprogramme mit intelligenten Fähigkeiten ausstatten, die Zeichenerkennung, Sprachverstehen und Strukturerkennung einschließen.

In der Tat gibt es Teildisziplinen des Document Engineering, die genau diese Fragestellungen untersuchen, wie etwa Optical Character Recognition oder Document Analysis. Es besteht jedoch Einigkeit, daß diese Verfahren heute bestenfalls auf spezialisierte Domänen anwendbar sind.

Der Ansatz der "strukturierten Dokumente" sieht einen anderen Ausgangspunkt für die computergestützte Informationsverarbeitung vor, nämlich eine explizite Repräsentation von Inhalt und Struktur des Dokumentes. Autorin oder Autor eines strukturierten Dokumentes editieren den eigentlichen Inhalt des Dokumentes und markieren in irgendeiner Weise die notwendigen Strukturelemente. Die so aufbereiteten "Dokumentendaten" lassen sich dann computergestützt weiterverarbeiten.

Um das Format für menschliche Augen brauchen Autor oder Autorin sich dann nicht mehr zu sorgen: Das kann aus einer separaten, vom Dokument unabhängigen und bei Bedarf auswechselbaren "Formatvorlage", auch "Stylesheet" genannt, berechnet werden. Das Stylesheet enthält allgemeine Regeln, wie die einzelnen Strukturelemente zu formatieren sind. Eine Regel kann beispielsweise festlegen, daß Überschriften in einer fetten Kursivschrift mit einem gewissen Abstand zum umgebenden Text formatiert werden sollen oder daß Absätze linksbündig mit Randausgleich zu setzen sind.

Ein "Formatierer" bringt das mit Strukturmarkierungen versehene Quelldokument mit dem Stylesheet zusammen und formatiert das Dokument entsprechend der Vorgaben in dem Stylesheet.

2.5 Vorteile des Dokumentenmodells

Ein solches Vorgehen hat mehrere Vorteile, da es ein hohes Maß an Flexibilität bietet. Wie bereits dargestellt, unterstützt die logische Strukturierung Anwendungen der Dokumentenverwaltung wie die automatische Generierung von Katalogeinträgen und Verzeichnissen oder das Information Retrieval nach bestimmten logischen Kriterien. Die eigentliche Informationsverarbeitung kann sich also direkt auf die logische Struktur stützen und muß sie nicht erst konstruieren. Die Unabhängigkeit der Stylesheets von den Dokumenten ermöglicht auf der einen Seite, daß ein- und dasselbe Dokument durch Kombination mit verschiedenen Stylesheets passend zum Anwendungszweck formatiert werden kann; auf der anderen Seite kann man verschiedene Dokumente durch Kombination mit ein- und demselben Stylesheet einheitlich formatieren. Auch läßt sich die Struktur beim Erstellen und Lesen von Dokumenten nutzen, etwa zur Navigation und zur Definition von Sichten (Outline-Sicht).

Von diesen Vorteilen profitieren besonders die Anwendungen, die große Dokumentenbestände erstellen und verwalten müssen. Stellen wir uns beispielsweise einmal eine große Sammlung von Projektvorschlägen ähnlich unserem Computer-Propädeutikum vor, die alle nach dem gleichen Muster logisch strukturiert und markiert sind. Dann könnten wir die ganze Sammlung systematisch nach anwendungsnahen Kriterien abfragen und uns etwa die im letzten halben Jahr neu hinzugekommenen Vorschläge mit Titel und Datum ausgeben lassen oder nach allen Projekten suchen, die ein bestimmtes gemeinsames Ziel verfolgen. Auch für die Verwaltung des Dokumentenbestandes böte das Modell der strukturierten Dokumente Vorteile. Die mit der Verwaltung befaßten Personen bräuchten für den gesamten Bestand nur einmal die zu verwendenden Strukturelemente festzulegen und sie müßten für jedes Präsentationsmedium nur ein einziges Stylesheet schreiben, das dann auf alle Dokumente des Bestandes anwendbar ist. Für unsere Projektvorschläge würden zwei Stylesheets ausreichen, eines für das Lesen am Bildschirm und eines für das Steuern eines Ausdrucks.

Das Modell der strukturierten Dokumente wurde in den achziger Jahren entwickelt ([MVD82], [FSS82], [AFQ89]); seine Wurzeln gehen sogar bis in die sechziger Jahre zurück [E63]. Wie wir im nächsten Abschnitt sehen werden, unterstützen alle modernen Textverarbeitungssysteme das Modell zumindestens in Ansätzen.

2.6 Verfeinerungen des Dokumentenmodells

Zum vollen Modell der strukturierten Dokumente gehört wesentlich dazu, daß die Namen der Strukturelemente frei wählbar sind. Wenn Sie beispielsweise ein Aufgabenblatt logisch strukturieren möchten, dann müssen Sie Strukturelemente wie Aufgabe, Abgabedatum, Punktezahl, Lösungshinweis oder Schwierigkeitsgrad angeben können. Ist der Sprachvorrat, aus dem Strukturelemente gewählt werden können, fest vorgegeben und beschränkt, kann er nicht alle sich im Laufe der Zeit ergebenden Anwendungen abdecken. Autorin oder Autor stehen dann vor der Situation, ihre von der Anwendung her benötigten Strukturelemente mit den vorgegebenen Elementen mehr recht als schlecht zu simulieren und zu entscheiden, ob beispielsweise eine Aufgabe ein Absatz mit Überschrift oder ein Listeneintrag sein soll. Das verwässert allerdings die Vorteile der strukturierten Dokumente für die computergestützte Informationsverarbeitung, da semantische Information über die Rollen der Strukturelemente verloren geht.

Historisch ist übrigens anzumerken, daß es u.a. im Verlagswesen mehrere Bestrebungen gegeben hat, einen universell einsetzbaren Vorrat an Strukturelementen zu definieren. Für einige Spezialgebiete konnte tatsächlich eine Einigung über einen solchen Vorrat erzielt werden. Das bekannteste Beispiel ist die Sprachdefinition der Text Encoding Initiative (TEI) für den Bereich der textkritischen Apparate im geisteswissenschaftlichen Bereich; der Sprachvorrat, der hier definiert wurde, ist allerdings sehr umfangreich. In der Regel sind solche Versuche jedoch gescheitert und zugunsten von frei wählbaren Strukturelementen aufgegeben worden.

Ein weiterer Gesichtspunkt beim Modell der strukturierten Dokumente ist, ob der zugrundeliegende Vorrat an Strukturelementen, egal ob anwendungsspezifisch definiert oder fest vorgegeben, formal

definiert werden soll. Eine "Strukturvorgabe", also eine mit formalen Methoden festgelegte Definition des Strukturelementevorrats, eventuell zusammen mit Einschränkungen, wie die Elemente verwendet werden dürfen, hat mehrere Vorteile. Beispielsweise können Dokumente automatisch auf ihre strukturelle Korrektheit überprüft werden, wenn solche Vorgaben existieren, oder Editierhilfen können solche Vorgaben auswerten und zur Unterstützung der Autorin oder des Autors einsetzen. Schließlich dokumentiert eine formale Strukturvorgabe auch das Vokabular einer Dokumentenanwendung. Wir werden uns mit dem Thema unter den Stichworten XML-Dokumententypdefinition und XML Schema in einem eigenen Teil noch weiter befassen.

2.7 Strukturdarstellungen

Bisher bin ich bewußt vage geblieben, wie denn die Markierung der Strukturelemente im Dokument erfolgen könnte. Doch auch dazu gibt es schon lange etablierte Konzepte, und zwar im wesentlichen zwei.

Beide Konzepte gehen davon aus, daß die logische Strukturierung streng hierarchisch angelegt ist. Strukturelemente können also andere Strukturelemente vollständig enthalten, sich aber nicht mit ihnen überlappen. Die Strukturelemente, die wir im Beispiel des Computer-Propädeutikums kennengelernt haben, gehorchen dem Gesetz der hierarchischen Schachtelung.

Die erste Methode stellt die hierarchische Struktur graphisch dar. Ich zeige hier drei Varianten. Die erste Variante nutzt ineinander geschachtelte Rechtecke oder Boxen, die mit dem Namen des jeweiligen Elementes gekennzeichnet sind. Der Anhang (Abbildung A.2, „Eine Dokumentdarstellung mit geschachtelten Boxen“) und die Web-Site unter "compProp/KonzeptSpy.gif" enthalten eine solche Darstellung. Eine zweite Variante finden Sie im Anhang (Abbildung A.3, „Eine Dokumentdarstellung mit eingerückten Ikonen“) und unter "compProp/KonzeptIcons.eps"; diese—im übrigen dynamische—Variante, die mit Einrückungen und Ikonen arbeitet, wird Ihnen aus Dateibrowsern bekannt sein. Eine dritte—ebenfalls dynamische—Variante besteht darin, die logische Struktur als einen Baum darzustellen. Der Anhang (Abbildung A.4, „Eine Dokumentdarstellung als Baum“) und "compProp/KonzeptTree.gif" enthalten eine solche Darstellung.

Die graphischen Darstellungen bieten eine gute Übersicht über die Strukturierung des Dokuments; ihre Schwäche liegt in der Präsentation des fortlaufenden Textes an den Blättern der Baumstrukturen.

Die graphischen Möglichkeiten stellen hohe Anforderungen an die Formatierungsmöglichkeiten des darstellenden und ja auch möglichst noch Interaktionen zulassenden Systems. Eine nicht so anspruchsvolle Möglichkeit besteht darin, die hierarchische Struktur mit Hilfe einer Klammerstruktur zu linearisieren. Zu diesem Zweck wird jeder Textbereich, der Inhalt eines logischen Elementes ist, an seinem Anfang und an seinem Ende mit einer öffnenden und einer schließenden Klammer versehen, die den Namen des entsprechenden Strukturelementes enthält. Eine solche Klammer ist einfach Klartext, der sich durch syntaktische Konventionen vom eigentlichen Inhalt abhebt. Der Inhalt ist unformatiert. Das Beispiel A.2, „Repräsentation der logischen Struktur mit Hilfe von eingebettetem Markup“ unter "compProp/Konzept.xml" benutzt die XML-Konvention, die öffnende Klammer für das Element `xxx` durch `<xxx>` und die schließende durch `</xxx>` darzustellen. Der Klartext für die öffnende oder schließende Klammer heißt "Tag" (englisch für Auszeichnung, speziell als prize tag für Preisauszeichnung) und der Oberbegriff zu Tags und eventuellen weiteren Möglichkeiten, nicht zum Inhalt zählende Zeichen in einen Text einzustreuen, ist "Markup". Eine graphische Verfeinerung dieser Darstellung arbeitet, wie in der Abbildung A.5, „Eine Verfeinerung durch Einrückungen und typographische Differenzierung“ unter "compProp/KonzeptMSIExml.eps" dargestellt, mit Einrückungen und mit typographischer Differenzierung von Text und Markup. Da das Markup mit im Text enthalten ist, wenn auch über syntaktische Konventionen vom eigentlichen Inhalt getrennt, spricht man auch von "eingebettetem Markup".

Von einem prinzipiellen Standpunkt aus gesehen, sind alle Darstellungsformen für strukturierte Dokumente äquivalent. Die Repräsentation der hierarchischen Struktur durch eingebettetes Markup ist technisch am einfachsten zu handhaben und ist deshalb derzeit noch die gängigste. Sie hat den Vorteil, daß Dokumente mit eingebettetem Markup mit jedem einfachen Texteditor, der etwa ASCII versteht, editiert und bearbeitet werden können. Die Darstellungsweise ist allerdings unübersichtlich und nicht sehr eingängig. Die anderen Darstellungsformen stoßen an ihre Grenzen, wenn die Schachtelungstiefe

sehr groß wird oder die Granularität der Struktur sehr fein. Das Problem taucht schnell bei der logischen Beschreibung mathematischer Formeln auf.

2.8 Formalisierung des Dokumentenmodells

2.9 Diskussion des Dokumentenmodells

Das Modell der strukturierten Dokumente wird eingesetzt, um Dokumente computerverarbeitbar zu machen. Die zentrale Idee ist, die Kerndaten des Dokuments redundanzfrei in einer Art Reinform vorzuhalten, wie man es auch beim Design relationaler Datenbanken mit Normalformen verfolgt. Daten, die sich aus dieser Quellform berechnen lassen oder die variabel gehalten werden sollen, werden nicht in der Quellform mitkodiert sondern bei Bedarf berechnet und gegebenenfalls als Sichten generiert. Verarbeitungsfunktionen können auf dieser Reinform der Daten aufsetzen und werden nicht belastet durch Idiosynchasien im Datenbestand, die durch einzelne Anwendungen begründet sind.

Im Datenbankbereich dient diese Methode gleichzeitig der logischen "Datenunabhängigkeit"; d.h., Änderungen an der Struktur der Quelldaten können vor Endsysteimen, die über Sichten mit der Datenbasis zu tun haben, verborgen bleiben. Das Potential der Datenunabhängigkeit ist durch das Modell der strukturierten Dokumente und seine Umsetzung mit XML auch im Bereich der Webdokumente gegeben. Allerdings ist die Technik noch nicht so weit fortgeschritten wie im Datenbankbereich, daß etwa mit entsprechenden Zugangsvoraussetzungen ausgestattete Personen einen Bestand von XML-Dokumenten über mit XSLT-generierte Sichten bearbeiten könnten. In diesem Bereich sind noch interessante Forschungsarbeiten zu erledigen. Auch eine Theorie der Normalformen, wie sie im Datenbankbereich etabliert ist, gibt es für den Bereich der strukturierten Dokumente erst in Ansätze.

Das Modell der strukturierten Dokumente sieht konkret vor, den Inhalt und die (logische) Struktur eines Dokumentes, auf denen die automatische Verarbeitung basieren soll, explizit zu repräsentieren. Dabei ist es wichtig, die Strukturelemente je nach Anwendungsfall frei definieren und die entsprechenden Definitionen auch formal festschreiben zu können. Auf dieser Basis, die je nach Anwendungsfall durch zusätzliche und separate Steuerungsdaten ergänzt werden kann, können Verarbeitungsfunktionen aufsetzen und Sichten generiert werden.

Historisch stammt das Modell aus dem Bereich des Document Management. Ein wichtiger Anwendungsfall in diesem Bereich, der das Modell wesentlich motiviert hat, ist die Formatierung. Vorgaben für die Formatierung werden in separaten und auswechselbaren Stylesheets vorgehalten und Formate aus beiden Beschreibungen zusammen berechnet. Über die Generierung formatierter Sichten hinaus sind Systeme des Document Management typischerweise in der Lage, Numerierungen, Verzeichnisse und Querverweise aus der expliziten Repräsentation von Inhalt und Struktur zu erzeugen, unter Hinzunahme von oft ebenfalls in Stylesheets enthaltenen Steuerungsdaten.

Das Modell unterstützt jedoch nicht nur Anwendungsfälle des Document Management im engeren Sinne, sondern trägt auch Anwendungen im weiteren Bereich des Knowledge Management. Dokumente machen mit dem Modell der strukturierten Dokumente als Stütze den Schritt von menschenlesbaren Texten zu gut verwaltbaren menschenlesbaren Texten und weiter zu maschinell verarbeitbaren, als Text kodierten Daten.

Das Modell der strukturierten Dokumente sieht vor, Bereiche im Inhaltstext eines Dokuments mit zusätzlicher semantischer Information zu markieren, indem es die Rolle des entsprechenden Textstücks benennt. Das Modell platziert sich damit in einem Spektrum, das von einer rein graphischen Repräsentation eines Dokuments, etwa als Pixelmuster oder als PostScript-Programm, bis zu einer expliziten Repräsentation der Semantik des Dokumentes, etwa als semantisches Netz reicht. Das eine Ende des Spektrums, die rein graphische Repräsentation, erfordert zur computergestützten Verarbeitung intelligente Software, die derzeit noch nicht zur Verfügung steht. Das andere Ende, eine ausgefeilte Repräsentation der Semantik, erfordert einen hohen Overhead beim Erstellen und Bearbeiten der Dokumente. In der Praxis hat sich der pragmatische Kompromiß, den man bei der Konzeption des Modells eingegangen ist, seit langem bewährt.

Dokumentensysteme, die das Modell der strukturierten Dokumente implementieren, kodieren die Strukturelemente in der Regel mit eingebettetem Markup. Das ist der zweite pragmatische Kompromiß im Bereich des Modells, der sich jetzt auf die Umsetzung des Modells bezieht. Zwar erscheinen immer wieder Positionspapiere mit Titeln wie "Embedded Markup Considered Harmful" ([N00], [RTW93]), aber auch hier scheint der Kompromiß den Test der Zeit zu bestehen.

Was wir in diesem Kapitel "Modell" genannt haben, heißt in der älteren Literatur auch manchmal "Dokumentenarchitektur". Der Begriff der Architektur macht klarer deutlich, worum es in unserer ersten Annäherung an Dokumente und ihre Formate geht, nämlich die relevanten Aspekte von Dokumenten zu identifizieren und ihr Zusammenwirken zu beschreiben.

Wir werden uns in dieser Vorlesung immer wieder fragen, welche abstrakten Modelle den sich im Internet manifestierenden Formaten explizit oder implizit zugrunde liegen, auch ohne daß uns für jede Ebene der Modellierung ein separates Wort wie hier "Architektur" zur Verfügung stünde. Gleich im nächsten Kapitel geht es beispielsweise um ein Modell für Zeichenkodierung. Aus diesem Grund ist in diesem Kapitel die Rede von dem Modell und nicht der Architektur strukturierter Dokumente.

2.10 Umsetzungen des Dokumentenmodells

Die meisten modernen Textverarbeitungssysteme setzen das Modell der strukturierten Dokumente in irgendeiner Form um. Wir sehen uns in diesem Kapitel zwei Systeme daraufhin an, nämlich LaTeX [L86] und Microsoft Word [RW99]. In späteren Kapiteln werden wir dann die Internet-Sprachen HTML und XML im Detail und auch ganz praktisch kennenlernen; dabei wird u.a. auch auf den Tisch kommen, inwieweit diese Sprachen dem Modell der strukturierten Dokumente folgen. Mit LaTeX und Microsoft Word, die ja keine dedizierten Internet-Formate sind, beschäftigen wir uns jedoch nur eingeschränkt unter diesem einem Aspekt des Dokumentenmodells.

Die beiden Systeme LaTeX und Word decken vom Typ her zusammen das Spektrum von Textsystemen ab, die im akademischen und technisch-dokumentatorischen Bereich sowie im Büro zu finden sind. Ich stelle die beiden Systeme also in der Erwartung vor, daß das von Ihnen konkret verwendete Textsystem vom Typ her einem der beiden entspricht. Die beiden Beispielsysteme sollen Sie in die Lage versetzen, für das von Ihnen verwendete Textsystem zu analysieren, in welcher Weise es das Modell der strukturierten Dokumente erfüllt.

Die Diskussion soll Sie auch dafür sensibilisieren, bei der Erstellung Ihrer eigenen Dokumente die angebotenen Mechanismen Ihres Dokumentensystem zur Unterstützung des Modells der strukturierten Dokumente möglichst weitgehend auszunutzen, im Interesse eines effizienteren Document Management und einer flexibleren Weiterverwendung Ihrer Dokumentdaten.

2.10.1 Umsetzung mit LaTeX

Beginnen wir mit LaTeX. Sie finden unser Konzept zum Computer-Propädeutikum im LaTeX-Format im Anhang (Beispiel A.3, „Ein Dokument im LaTeX-Format“) und auf dem Webserver zum Buch unter "compProp/KonzeptTeX.tex". Am besten ist es, wenn Sie sich mit dem möglicherweise fremden Format erst einmal ein bißchen vertraut machen. Machen Sie sich klar, daß der Inhalt der LaTeX-Version mit dem der vorhin schon besprochenen XML-Version übereinstimmt und daß auch eine gewisse Übereinstimmung bei den markierten Strukturen festzustellen ist.

Die in der LaTeX-Version explizit markierten logischen Elemente sind die Angaben über die Autorinnen (`\author{...}`), die Hauptüberschrift (`\title{...}`), die Abschnittüberschriften (`\section{...}`), die Liste (`\begin{itemize} ... \end{itemize}`), die Aufzählungspunkte (`\item ...`), das Dokument als Ganzes (`\begin{document} ... \end{document}`), das Datum (`\Datum`) und die Versionsnummer (`\Version{...}`).

Als erster Eindruck, den wir gleich noch revidieren werden, ergibt sich also, daß LaTeX genau wie XML eingebettetes Markup verwendet, um die logische Struktur eines Dokuments zu beschreiben. Allerdings sind die syntaktischen Formen des Markup in LaTeX vielfältiger als in XML. Eine

direkte Entsprechung zu XMLs Konstrukt `<XXX> . . . </XXX>`, um ein logisches Element XXX zu markieren, ist `\begin{XXX} . . . \end{XXX}`, aber es existieren auch die Formen `\XXX{ . . . }` und sogar `\XXX . . .` ohne explizite Markierung des Elementendes; dies ergibt sich vielmehr aus der nächsten Anfangsmarkierung eines Elements, zusammen mit der Metainformation, ob dieses nächste Element im Element XXX enthalten sein kann. Einen Sonderfall stellt die implizite Abgrenzung von Absätzen durch Leerzeilen dar.

Das zur Verfügung stehende Repertoire von logischen Elementen ergibt sich aus der zugrundeliegenden Dokumentenklasse (`\documentclass{...}`) und den hinzugeladenen Paketen (`\usepackage{...}{...}`). Hier werden externe Dateien referenziert, die i.a. systemweit zur Verfügung stehen und die Sprachelemente definieren, die im Dokument vorkommen können. Am Anfang der LaTeX-Version finden Sie außerdem auch intern definierte Elemente, nämlich `\Datum` und `\Version`.

LaTeX sieht also vor, daß Autorin oder Autor ihr eigenes Vokabular von Elementen definieren und daß eine Gruppe von Autorinnen oder Autoren ein gemeinsames Vokabular benutzt, das dann nur einmal definiert werden muß und von verschiedenen Dokumenten aus referenziert werden kann. Das Vokabular selbst ist formal definiert; weitere syntaktische Einschränkungen sind implizit.

LaTeX erfüllt somit die erste Anforderung des Modells der strukturierten Dokumente, den Inhalt und die logische Struktur eines Dokumentes explizit zu repräsentieren. Die Strukturelemente können je nach Anwendungsfall frei definiert werden; ansatzweise ist es auch möglich, die Strukturelemente formal festzulegen.

Wie sieht es mit der zweiten Anforderung aus, daß Vorgaben für die Formatierung in separaten und auswechselbaren Stylesheets vorzuhalten sind? Hier kommen wieder die Konstrukte `\documentclass{...}` und `\usepackage{...}{...}` in's Spiel, die wir schon bei der Definition der Strukturelemente herangezogen haben. Die hier referenzierten externen Dateien erfüllen nämlich eine Doppelfunktion: Sie legen fest, welche Strukturelemente im Dokument vorkommen können, und geben gleichzeitig vor, wie die Strukturelemente formatiert werden sollen.

Sie können diese Doppelfunktion nachvollziehen, wenn Sie sich die interne Definition von `\Datum` und `\Version` anschauen. Die Definition von `\Version` besagt beispielsweise, daß das nach der Markierung in geschweifte Klammern eingeschlossene Argument, also die Versionsnummer, fett gesetzt werden soll (`fett=boldface`, abgekürzt zu `bf`). Und die Definition von `\Datum` besagt, daß die Systemfunktion `\today` aufgerufen wird, die das aktuelle Datum einsetzt.

Die Forderung nach Stylesheets, die im Falle von LaTeX extern und somit auswechselbar oder intern und somit an das Dokument gebunden sein können, sind also ebenfalls erfüllt.

Ein LaTeX-Prozessor liest nun Dokument und Stylesheets und formatiert die logischen Strukturen entsprechend der Angaben im Stylesheet. Das Ergebnis sehen Sie ebenfalls im Anhang (Abbildung A.6, „Eine LaTeX-formatierte Version des Dokuments“) und auf der Web-Site unter "compProp/KonzeptTeX.eps". Wenn Sie diese formatierte Version mit der früher gezeigten formatierten Version (Abbildung A.1, „Ein formatiertes Dokument“) vergleichen, stellen Sie fest, daß der Informationsgehalt beider Versionen identisch ist, obwohl sich die Formatierungen geringfügig unterscheiden.

LaTeXs Schwachstellen bei der Umsetzung des Modells der strukturierten Dokumente werden offensichtlich, wenn wir uns genauer ansehen, wie LaTeX die logische Markierung mit der Formatvorgabe verbindet und wie das Format definiert ist. LaTeX behandelt nämlich die Markierung als Kommando einer sogenannten Makroprogrammiersprache und die Formatdefinition als Programm in dieser Programmiersprache. Diese Makroprogramme können nun alles mögliche tun, nicht nur die Formatierung festlegen. Beispielsweise speichern die Programme für `\author` und `\title` ihre Argumente nur intern zwischen. Erst das Kommando `\maketitle` im Eingabedokument, das gar keine rechte logische Funktion hat, plazierte die entsprechenden Angaben auf der Seite und legt ihre graphische Erscheinungsform fest. Im Extremfall können Makroprogramme sogar andere Kommandos umdefinieren. Beispielsweise könnten die Leerzeilen, die Absatzgrenzen markieren, so umdefiniert werden, daß das jeweils erste Wort eines Absatzes als Schlüsselwort fett gedruckt wird.

Ebenso bewirkt das Klammerpaar `\begin{verbatim}` und `\end{verbatim}`, daß dazwischen eingeschlossenes Markup im Klartext ausgegeben und nicht interpretiert wird.

Während also eine pure Markupsprache nur die Markierungen bereitstellt, die im Dokument verwendet werden können, koppelt LaTeX die Funktion von Markupsprache, Formatdefinition und Programmierung untrennbar aneinander. Bei weisem Gebrauch muß das nicht unbedingt problematisch sein; es widerspricht allerdings dem allgemeinen und wohlbegründeten Informatikprinzip Separation of Concerns. In der Praxis bedeutet das, daß eine Person, die ein Stylefile für LaTeX erstellen möchte, über ein ganzes Spektrum von Fertigkeiten verfügen muß, nämlich die gewünschte logische Struktur zu planen und zu organisieren, das graphische Design zu entwerfen und die so entwickelten Vorstellungen durch Computerprogramme zu realisieren. Die Einstiegshürde für einen kreativen Gebrauch von LaTeX ist also hoch. Im Gegenzug sozusagen ist LaTeX durch seine Programmierbarkeit ein mächtiges und in seiner Funktionalität bei Bedarf beträchtlich erweiterbares System.

LaTeX ist ursprünglich als Textsatzsystem entwickelt worden. Seine Funktionen beschränken sich also im wesentlichen auf den Bereich des Document Management. Da jedoch LaTeX die Technik des eingebetteten Markups zur Repräsentation von Strukturelementen verwendet, können Fremdanwendungen im Prinzip LaTeX-Quelldokumente als Text lesen und automatisch weiterverarbeiten. Damit ist es möglich, LaTeX-Dokumente als Textdaten anzusehen und computergestützt zu verarbeiten.

Die LaTeX-Software ist übrigens frei verfügbar; die deutsche TeX Users Group DANTE e.V. stellt LaTeX für verschiedene Plattformen unter <http://www.dante.de/> zur Verfügung.

Zusammenfassend läßt sich also feststellen, daß LaTeX das Modell der strukturierten Dokumente weitgehend erfüllt, nur daß die Markupsprache nicht einfach deklarativ Markierungen bereitstellt sondern eine volle Programmiersprache ist.

Nach ähnlichen Prinzipien wie LaTeX operieren die älteren und nicht mehr allzu gängigen Systeme TROFF und GML.

2.10.2 Umsetzung mit Microsoft Word

Das zweite System, das wir näher betrachten wollen, nämlich Microsoft Word, steht LaTeX fast diametral entgegen. Einfachheit der Benutzung als Bürosystem ist oberstes Ziel, für das Abstriche bei der Mächtigkeit, Flexibilität und Erweiterbarkeit in Kauf genommen werden.

Ein Word-Dokument editiert man mit einer formatierten Sicht. Markup wird nicht explizit angezeigt, wie in einem LaTeX-Quelldokument, sondern die einem logischen Element mit einer Formatvorgabe zugeordnete Formatierung wird gleich umgesetzt und am Bildschirm angezeigt (WYSIWYG—What you see is what you get). Online liegt Ihnen das Konzept zum Computer-Propädeutikum als Word-Dokument vor unter "compProp/Konzept.doc"; der Anhang (Abbildung A.7, „Ein Beispieldokument im Word-System“) enthält ein Bildschirmfoto desselben Dokuments. Wenn Sie das Word-System installiert haben, können Sie nachvollziehen, welche Strukturelemente ich verwendet habe, nämlich Autorin, Titel, Hauptüberschrift, Nebenüberschrift, Datum, Version, Abschnitt Überschrift, Absatz und Aufzählungspunkt. Öffnen Sie dazu das Dokument, positionieren Sie den Cursor in die entsprechenden Elemente und lesen Sie ihren Namen in der Menüzeile ab.

Word ermöglicht es Autorinnen und Autoren, Zeichenketten oder ganzen Absätzen eine Druckformatvorlage zuzuweisen. Eine Druckformatvorlage ist ein Satz von Formatierparametern mit ihren Werten, die im Dokument über einen Namen referenziert werden können. Für die Word-Version des Computer-Propädeutikums habe ich beispielsweise eine Druckformatvorlage mit Namen "Abschnittüberschrift" definiert und den einzelnen Abschnittüberschriften zugewiesen.

Vom Modell der strukturierten Dokumente aus gesehen entsprechen also die Namen der Druckformatvorlagen dem Vokabular an Strukturelementen, die Druckformatvorlage selber entspricht einer Regel im Stylesheet und die Zuweisungen der Druckformatvorlagen an die Absätze und Zeichenketten im Dokument entsprechen der Markierung mit logischen Strukturelementen. Word

ermöglicht damit die explizite Repräsentation von Inhalt und logischer Struktur eines Dokumentes. Die Strukturelemente können je nach Anwendungsfall frei definiert werden; ansatzweise ist es auch möglich, die Strukturelemente formal festzulegen.

Darüber hinaus ist es in Word möglich, in sogenannten "Document Templates", die als Stylesheets fungieren, Sammlungen von Druckformatvorlagen anzulegen, sie zu verwalten und sie Dokumenten zuzuordnen. Stylesheets in Word sind auf diese Weise auswechselbar.

Trotzdem erfüllt Word das Modell der strukturierten Dokumente nicht in zufriedenstellendem Maße. Der Grund liegt darin, daß außer für Absätze und darin enthaltene Zeichenketten keine logischen Markierungen im Dokument angebracht werden können. Beispielsweise können wir die einzelnen Aufzählungspunkte der Liste nicht zu einem logischen Element Liste zusammenfassen. Genauso wenig können wir eine Abschnittüberschrift mit den zugehörigen Absätzen zu einem Abschnitt bündeln und entsprechend markieren. Die Schachtelungstiefe von Strukturelementen in Word ist also empfindlich limitiert.

Word verwendet kein eingebettetes Markup für die Repräsentation von Strukturelementen im Dokument. Word-Dokumente liegen vielmehr in einem proprietären, nicht dokumentierten und sich häufig ändernden Format vor. Möchte man ein Word-Dokument automatisch weiterverarbeiten und dabei auch die Struktur der Formatvorlage auswerten, gibt es zwei Möglichkeiten: Microsoft hat eine stabile Zugangsschnittstelle (API) für Fremdanwendungen definiert, die Word-Dokumente aus Programmen heraus verarbeiten möchten. Diese API ist von einigen Programmiersprachen (Visual Basic, C++ aus nutzbar. Das ist die erste Möglichkeit. Die zweite Möglichkeit besteht darin, das Word-Dokument in das stabile und dokumentierte Format RTF zu exportieren und auf die RTF-Daten zuzugreifen. Der Haken bei diesem Ansatz ist allerdings, daß die Abbildung der Strukturelemente aus Word nach RTF instabil und mit Informationsverlust behaftet ist. Bei Word sind also erheblich höhere Hürden damit verbunden, die Dokumentdaten einer Fremdanwendung zugänglich zu machen als bei LaTeX.

Nach ähnlichen Prinzipien wie Word operieren Word Perfect, FrameMaker, Microstar Office und Quark Express.

2.11 Das Modell der semistrukturierten Daten

2.12 Zusammenfassung und Ausblick

Moderne Dokumente sind Informationsträger, die von menschlichen Leserinnen und Lesern in unterschiedlichen Kontexten und mit unterschiedlichen Zielen verwendet werden, die aber auch computergestützt oder programmgesteuert weiterverarbeitet werden. Ein modernes Dokumentenmodell muß das Spektrum von Dokumentenanwendungen in seiner ganzen Breite unterstützen. Da Computer natürliche Sprache in einem für die automatische Bearbeitung von Dokumenten erforderlichen Umfang auch in absehbarer Zeit nicht verstehen werden, müssen moderne Dokumente semantische Information, die menschliche Leserinnen und Leser durch gedankliche Arbeit erschließen könnten, für die programmgestützte Weiterverarbeitung in expliziter Form enthalten.

In den achtziger Jahren wurde das Modell der strukturierten Dokumente entwickelt, um diesen Anforderungen zu entsprechen. Ein strukturiertes Dokument enthält dabei eine explizite Repräsentation sowohl des eigentlichen textuellen Inhalts als auch der semantischen Rollen von Textbereichen. Diese Information von Text und Strukturelementen wird hierarchisch strukturiert dargeboten. Ein strukturiertes Dokument kann mit einer separaten und auswechselbaren Formatvorlage (Stylesheet) kombiniert werden, um eine formatierte Sicht auf das Dokument zu berechnen. Die für moderne Dokumentenanwendungen essentielle Flexibilität läßt sich jedoch nur erreichen, wenn das Vokabular von Strukturelementen frei konfigurierbar ist.

Das Modell der strukturierten Dokumente stellt in zweifacher Hinsicht einen pragmatischen Kompromiß dar. Der erste Kompromiß bezieht sich auf die semantische Information, mit der der reine Text eines strukturierten Dokuments angereichert ist. Diese semantische Information besteht

in einer simplen Benennung der semantischen Rollen von Textbereichen. Der zweite pragmatische Kompromiß betrifft die Methode, mit der die Strukturmarkierungen in einem strukturierten Dokument in der Regel angebracht sind, nämlich in Form von eingebettetem Markup.

Das Modell der strukturierten Dokumente bewährt sich seit zwanzig Jahren in umfangreichen technischen Dokumentationen (z.B. im Flugzeugbau, in dem die gedruckte Dokumentation eines Airbus schwerer ist als das Flugzeug selber) und bei Projekten zum elektronischen Publizieren. Es ist -zumindestens in Anklängen- in allen aktuellen Text-, Dokumenten- und Dokumentationssystemen umgesetzt. Und es bildet die Grundlage der beiden Websprachen HTML und XML.

Drei wichtige Faktoren machen digitale Dokumente zu modernen Informationsträgern, die vielfach verwendbar und der automatischen Verarbeitung zugänglich sind. Der erste Faktor ist das Modell der strukturierten Dokumente, dem solche digitalen Dokumente entsprechen sollen. Die weiteren Faktoren sind die Standardisierung der Textkodierung, die Textinhalte austauschbar macht, und die Standardisierung der Strukturinformation. Mit der Standardisierung der Textkodierung befassen wir uns im nächsten Kapitel; mit der Standardisierung der Strukturinformation werden wir in einem späteren Kapitel im Zusammenhang mit XML zu tun haben. Vorher werden wir uns jedoch mit der "kleinen Lösung" HTML und der zugehörigen Stylesheet-Sprache CSS befassen.

Wir haben in diesem Kapitel eine Reihe allgemeiner Informatik-Prinzipien angewendet gesehen. Ich möchte diese Prinzipien zum Abschluß des Kapitels noch kurz diskutieren.

1. Late Binding.

Late Binding bedeutet, salopp gesagt, Entscheidungen so lange hinauszuzögern, bis sie unumgänglich sind. Das Prinzip des Late Binding bringt oft einen Gewinn an Flexibilität und wird in ganz unterschiedlichen Teilgebieten der Informatik angewendet. Polymorphie in objektorientierten Programmiersprachen ist ein Beispiel. Polymorphie bedeutet, daß in einer Klassenhierarchie spezialisierte Klassen die Methoden allgemeinerer Klassen, von denen sie erben, umdefinieren dürfen. Erst zur Laufzeit steht für ein jedes Objekt fest, wie es in der Klassenhierarchie einzuordnen ist und welche der polymorph definierten Methoden für das Objekt Gültigkeit haben.

In diesem Kapitel haben wir das Prinzip des Late Binding im Zusammenhang mit Stylesheets kennengelernt. Erst wenn ein Dokument präsentiert werden soll, wird ein zum Anwendungszweck passendes Stylesheet zugeschaltet und das graphische Aussehen des Dokuments festgelegt.

2. Separation of Concerns.

Separation of Concerns bedeutet, ein System so zu gestalten, daß unterschiedliche Aufgaben unabhängig voneinander durchgeführt werden können. Separation of Concerns erfordert entsprechende Modelle und Architekturen. Ein klassischer Fall von Separation of Concerns sind Datenbankanwendungen. Durch die Trennung in eine Anwendungsschicht, eine konzeptionelle Schicht und eine physikalische Schicht kann unabhängig voneinander an der Anwendungssoftware und an den Speicher- und Zugriffsstrukturen gearbeitet werden. Dies garantiert Datenunabhängigkeit.

In diesem Kapitel haben wir Separation of Concerns durch das Dokumentenmodell erzielt: Unabhängig voneinander können entsprechende Fachleute an der inhaltlichen Gestaltung und an der Präsentation eines Dokuments arbeiten.

Kapitel 3. Zeichenkodierungen

3.1 Orientierung

Wir befassen uns in diesem Kapitel mit dem grundlegenden Bestandteil eines strukturierten Dokuments, nämlich seinem Inhalt, und damit, wie dieser Inhalt im Computer repräsentiert -man sagt auch "kodiert"- werden kann. Wir gehen davon aus, daß der Inhalt aus "glattem" Text besteht, also aus einer Folge von elementaren Texteinheiten wie Buchstaben, Ziffern, Interpunktionszeichen und anderen Zeichen. Diese Zeichen müssen letztendlich in Bits und Bytes kodiert werden. Eine "Zeichenkodierung" gibt dazu eine Vorschrift an.

Die bekannteste Zeichenkodierung ist "US-ASCII", standardisiert als ANSI X3.4 (1968) und als die Variante ISO 646-US von "ISO 646" (1972). Der Zeichensatz von US-ASCII (Abbildung 3.1, „Der Zeichensatz ISO 646-US (US-ASCII)“) reicht lediglich, um lateinische und US-amerikanische Texte sowie Texte in einigen wenigen weiteren Sprachen zu kodieren. Für das Deutsche, das Dänische und vierzehn weitere Sprachen sieht ISO 646 deshalb nationale Varianten vor, die die in Abbildung 3.1 fett umrandeten US-ASCII-Zeichen "#" (Hash), "\$" (Dollar), "@" (At), "[" (eckige Klammer auf), "\" (Backslash), "]" (eckige Klammer zu), "^" (Caret), "" (Grave), "{" (geschweifte Klammer auf), "|" (Strich), "}" (geschweifte Klammer zu) und "~" (Tilde) durch nationale Zeichen ersetzen. Die deutsche Variante ISO 646-DE (Abbildung 3.2, „Die deutsche Variante ISO 646-DE von ISO 646“) nimmt beispielsweise die in Tabelle 3.1, „Die Ersetzungen der deutschen Variante von ISO 646“ angegebenen Ersetzungen vor.

Tabelle 3.1. Die Ersetzungen der deutschen Variante von ISO 646

ISO 646-US	ISO 646-DE
[Ä
\	Ö
]	Ü
{	ä
	ö
}	ü
~	ß
@	§

Aus diesem Grund erscheint ein Absender "Technische Universität München, Arcisstraße 21" gelegentlich auch heute noch in amerikanischen E-Mail-Programmen in der Form "Technische Universit{t M}nchen, Arcisstra~e 21"; d.h., die ursprünglich mit ISO 646-DE angelegte Kodierung der Adresse wurde mit Hilfe von ISO 646-US ausgegeben. Sie können sich vorstellen, daß im Jahre 1984, als ich es das erste Mal versuchte, das Schreiben von LaTeX-Texten auf einer deutschen Tastatur wegen der vielen Vorkommen der Steuerzeichen "{", "}", "\", "[" und "]" einige Probleme aufwarf.

Mitte der achziger Jahre erweiterte die European Computer Manufacturer's Association (ECMA) den Zeichensatz US-ASCII zu einer Familie von Zeichensätzen, mit denen die alphabetischen Schriftsysteme kodiert werden können. Die inzwischen von der ISO unter dem Namen "ISO 8859" kodierte Zeichensatzfamilie besteht aus den Zeichensätzen ISO 8859-1 bis ISO 8859-15. Jeder Zeichensatz ISO 8859-X umfaßt die 128 US-ASCII-Zeichen und ergänzt sie um weitere 128 Zeichen. Für die meisten westeuropäischen Sprachen, z.B. das Deutsche, Dänische oder Französische, ist ISO 8859-1, auch ISO Latin-1 genannt, (Abbildung 3.3, „Der Zeichensatz ISO 8859-1 (ISO Latin-1)“) relevant. ISO 8859-7 deckt das griechische Alphabet ab und ISO-8859-15 ersetzt im wesentlichen das internationale Währungssymbol mit dem Eurozeichen. Die sogenannte "Code Page 1252" (Abbildung 3.4, „Der Windows-Zeichensatz 1252“) von Microsoft Windows ist in weiten Teilen identisch mit ISO 8859-1, ersetzt aber einige Kontrollzeichen von ISO 8859-1 durch druckbare Zeichen, u.a. das Eurozeichen.

Abbildung 3.1. Der Zeichensatz ISO 646-US (US-ASCII)

20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D	2E	2F
	!	"	#	\$	%	&	'	()	*	+	,	-	.	/
30	31	32	33	34	35	36	37	38	39	3A	3B	3C	3D	3E	3F
0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F
@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
50	51	52	53	54	55	56	57	58	59	5A	5B	5C	5D	5E	5F
P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
60	61	62	63	64	65	66	67	68	69	6A	6B	6C	6D	6E	6F
`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
70	71	72	73	74	75	76	77	78	79	7A	7B	7C	7D	7E	
p	q	r	s	t	u	v	w	x	y	z	{		}	~	

Anfang der neunziger Jahre kamen "Unicode" und "ISO/IEC 10646" heraus. Das Ziel dieser beiden in einem Sinne, den wir noch genauer kennenlernen werden, äquivalenten Zeichenkodierungen ist Universalität, also Texte aus sämtlichen Sprachen der Welt eindeutig zu kodieren. Die aktuellen Versionen Unicode 3.0 und ISO/IEC 10646-1:2000 kodieren 49.194 Zeichen, die alle modernen und viele klassischen Sprachen abdecken. Unicode und ISO/IEC 10646 werden laufend um weitere historisch bedeutsame Zeichen und Sprachen ergänzt.

Die Kodierungen ISO 8859 und Unicode sind von zentraler Bedeutung für die Internetformate HTML und XML: Ohne internationale Standards für die Kodierung universeller Zeichensätze wäre der grenzüberschreitende Dokumentenaustausch im Internet und im Web zum Scheitern verurteilt. Wir werden diese Kodierungen deshalb in diesem Kapitel genauer besprechen, und zwar im Rahmen eines abstrakten "Kodierungsmodells" für Zeichen ([DW00], [DYIWFT02], [VV02]).

Das Kodierungsmodell gibt uns einen konzeptuellen Rahmen, innerhalb dessen wir die Kodierung von potentiell 2^{31} , d.h. einigen Milliarden Zeichen in Bitmuster strukturieren und so besser verstehen können.

Wir werden in den folgenden Abschnitten die einzelnen Bestandteile des Kodierungsmodells besprechen und auf konkrete Zeichenkodierungen anwenden, wobei Unicode die zentrale Rolle spielt. Die einzelnen Bestandteile sind:

1. Ein abstrakter Zeichensatz.
2. Eine Kodetabelle.
3. Ein Kodierungsformat.
4. Ein Kodierungsschema.
5. Eine Übertragungssyntax.

Die einzelnen Bestandteile des Kodierungsmodells entsprechen zunehmend konkreteren Repräsentationen von Zeichen, von einem abstrakten Begriff hin zu Bitmustern; zwischen einer abstrakten Repräsentationsebene und der nächstliegenden konkreteren Repräsentationsebene besteht eine Abbildung. Um über mehrere Ebenen hinweg von einer Zeichenposition in einer Kodetabelle direkt zu seiner Kodierung in einem Kodierungsschema zu gelangen, können wir die verschiedenen Abbildungen hintereinanderausführen und erhalten damit eine sogenannte "Zeichenkarte" ("Character Map").

3.2 Abstrakter Zeichensatz

Die oberste, abstrakteste Ebene des Kodierungsmodells ist der "abstrakte Zeichensatz", der aus einem Satz, d.h. einer Menge, sogenannter "abstrakter Zeichen" besteht. Ein abstraktes Zeichen ist eine Informationseinheit, die zur Repräsentation, Organisation oder Kontrolle von Text dient, also beispielsweise Buchstaben, Ziffern, Interpunktionszeichen, Akzente, graphische Symbole, ideographische Zeichen, Leerzeichen, Tabulatoren, "Line Feed" und "Carriage Return", Kontrollcodes für "Start of Selected Area" und "End of Selected Area" etc. Der Zeichensatz von US-ASCII beispielsweise besteht aus 33 Kontrollzeichen und 95 druckbaren Zeichen.

Ein abstrakter Zeichensatz beinhaltet noch keine Ordnung oder Nummerierung der im Zeichensatz enthaltenen Zeichen; dieser Aspekt kommt in der nächsten Stufe des Kodierungsmodells in Form einer Kodetabelle hinzu.

Abstrakte Zeichensätze wurden in der Regel festgelegt, um alle Texte einer bestimmten Sprache oder Sprachfamilie angemessen zu kodieren. Der Anspruch, alle lebenden und alle historisch bedeutsamen Sprachen mit einem einzigen Zeichensatz kodieren zu können, führte zu den (identischen) Zeichensätzen von Unicode und ISO/IEC 10646, dem sogenannten "Universal Character Set" ("UCS").

Abbildung 3.2. Die deutsche Variante ISO 646-DE von ISO 646

20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D	2E	2F
	!	"	#	\$	%	&	'	()	*	+	,	-	.	/
30	31	32	33	34	35	36	37	38	39	3A	3B	3C	3D	3E	3F
0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F
S	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
50	51	52	53	54	55	56	57	58	59	5A	5B	5C	5D	5E	5F
P	Q	R	S	T	U	V	W	X	Y	Z	Ä	Ö	Ü	^	_
60	61	62	63	64	65	66	67	68	69	6A	6B	6C	6D	6E	6F
,	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
70	71	72	73	74	75	76	77	78	79	7A	7B	7C	7D	7E	
p	q	r	s	t	u	v	w	x	y	z	ä	ö	ü	ß	

Abbildung 3.3. Der Zeichensatz ISO 8859-1 (ISO Latin-1)

A0	A1	A2	A3	A4	A5	A6	A7	A8	A9	AA	AB	AC	AD	AE	AF
	í	φ	£	¥	¥	!	§	..	©	≡	«	¬	-	®	-
B0	±	²	³	⁴	⁵	⁶	⁷	⁸	¹	º	»	¼	½	¾	¿
C0	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î
D0	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ
E0	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î
F0	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ

Abbildung 3.4. Der Windows-Zeichensatz 1252

80	€		82	,	83	f	84	,,	85	...	86	†	87	‡	88	~	89	%	9A	Š	9B	<	9C	œ		9E	Ž				
	91	‘	92	’	93	“	94	”	95	•	96	-	97	-	98	~	99	™	9A	Š	9B	>	9C	œ		9E	Ž	9F	ÿ		
A0	A1	i	A2	φ	A3	£	A4	¤	A5	¥	A6	ı	A7	§	A8	..	A9	©	AA	≡	AB	«	AC	¬	AD	-	AE	®	AF	-	
B0	°	B1	±	B2	²	B3	³	B4	ˆ	B5	μ	B6	¶	B7	·	B8	¸	B9	¹	BA	º	BB	»	BC	¼	BD	½	BE	¾	BF	¿
C0	À	C1	Á	C2	Â	C3	Ã	C4	Ä	C5	Å	C6	Æ	C7	Ç	C8	È	C9	É	CA	Ê	CB	Ë	CC	Ì	CD	Í	CE	Î	CF	Ï
D0	Ð	D1	Ñ	D2	Ò	D3	Ó	D4	Ô	D5	Õ	D6	Ö	D7	×	D8	Ø	D9	Ù	DA	Ú	DB	Û	DC	Ü	DD	Ý	DE	Þ	DF	ß
E0	à	E1	á	E2	â	E3	ã	E4	ä	E5	å	E6	æ	E7	ç	E8	è	E9	é	EA	ê	EB	ë	EC	ì	ED	í	EE	î	EF	ï
F0	ä	F1	ñ	F2	ò	F3	ó	F4	ô	F5	õ	F6	ö	F7	÷	F8	ø	F9	ù	FA	ú	FB	û	FC	ü	FD	ý	FE	þ	FF	ÿ

Tabelle 3.2. Das Verhältnis von abstrakten Zeichen und Glyphen

Abstrakte Zeichen	Glyphen
A	À
f+i	fi
f+f+l	ffl
ö	o+¨
→	→ oder →

Zur visuellen Präsentation von Text dienen abstrakte graphische Einheiten, die "Glyphen" genannt werden. Glyphen entsprechen auf der Präsentationsebene den Zeichen auf der Kodierungsebene. Die Liste

À, Á, Â, Ã, Ä, Å

enthält graphische Ausprägungen des Glyphen für den Buchstaben "A", sogenannte Glyphenbilder. Nach Design, Größe, Gewicht und anderen Charakteristika zusammengehörige Glyphenbilder bilden einen Font.

Die Beziehung zwischen abstrakten Zeichen und Glyphen ist komplex. Tabelle 3.2 gibt eine Gegenüberstellung. Im einfachsten Fall präsentiert sich ein einzelnes Zeichen als genau ein Glyph. Einzelne Glyphen können jedoch auch ganze Folgen von abstrakten Zeichen darstellen, wie etwa bei den Ligaturen "fi", "fl", "ff" oder "ffl". Ein einziges abstraktes Zeichen mit Akzent, wie das Zeichen "ö", kann durch zwei Glyphen dargestellt sein, nämlich den Glyphen für das Basiszeichen "o" und den darüber positionierten Glyphen für den Akzent. Im Tamilischen gibt es Buchstaben, die durch ein Paar von Glyphen repräsentiert werden; dieses Glyphenpaar umrahmt dann in der formatierten Sicht einen weiteren Glyphen, der den Nachbarbuchstaben im Text darstellt. Im Arabischen schließlich gibt es sogenannte kontextuelle Formen; d.h., ein Zeichen wird je nach dem Kontext seiner Nachbarzeichen durch Glyphen ganz verschiedener graphischer Form dargestellt. Ähnliche Phänomene sind aus der mittelalterlichen Satztechnik bekannt: Gutenberg beispielsweise verwendete für ein- und denselben Buchstaben verschieden breite Glyphen oder abkürzende Symbole für Wörter oder Silben, um einen gleichmäßigen rechten Rand zu erzeugen; hier gibt also die Formatierung den Kontext für die Wahl eines von mehreren Glyphen. In mathematischen Ausdrücken stellt die Semantik einen Kontext für die Auswahl von Glyphen dar. Beispielsweise kann ein langer oder kurzer Pfeil gewählt werden. Typfestlegung einer Funktion:

$(f: A \longrightarrow B)$

Produktion in einer Grammatik:

$(S \rightarrow [S])$.

.

In vielen Fällen sind die elementaren Texteinheiten, die als abstrakte Zeichen Eingang in einen Zeichensatz finden sollen, nicht offensichtlich. Sowohl aus den verschiedenen Sprachen, deren Texte kodierbar sein sollen, als auch aus den verschiedenen Verarbeitungsfunktionen, die auf die Texte Anwendung finden sollen, können Anforderungen und Konflikte erwachsen.

Im Schwedischen beispielsweise sind "ä" und "ö" Buchstaben für sich, die hinter "z" im Alphabet angeordnet sind. Im Französischen dagegen gibt die Diaräse über einem Vokal lediglich einen Hinweis, daß der Vokal separat zu sprechen ist, etwa in "duët". Anstelle eines einzigen abstrakten Zeichens "e mit Diaräse" wären also die beiden abstrakten Zeichen "e" und "''" mit zwei getrennten, aber kombinierbaren Glyphen natürlicher.

Für die Darstellung eines deutschen Textes in gedruckter Form ist die Unterscheidung von Groß- und Kleinbuchstaben im Zeichensatz äußerst bequem; für Sortierfunktionen oder Vergleiche dagegen stellt sie eine (wenn auch kleine) Hürde dar. Im Spanischen gilt "ll" für das alphabetische Sortieren als ein einziger Buchstabe, für die Formatierung wird "ll" als Folge von zwei Buchstaben behandelt.

Die Festlegung eines abstrakten Zeichensatzes erfordert also eine sorgfältige Abwägung von verschiedenen Alternativen, wobei Sprachen, Schriften und Bearbeitungsfunktionen zu berücksichtigen sind. Das zugegebenermaßen sehr allgemein formulierte Ziel bei der Entwicklung des Unicode-Zeichensatzes und der gesamten Unicode-Kodierung war es, die Implementierung nützlicher Verarbeitungsprozesse für Textdaten zu ermöglichen.

Unicode 3.0 formuliert zehn Designprinzipien für den Unicode-Standard. Zwei davon lassen sich bereits mit den Begriffen auf der Ebene des Zeichensatzes formulieren:

1. Unicode kodiert Zeichen, nicht Glyphen.
2. Unicode kodiert glatten Text, keine Formatinformation.

Wir beschäftigen uns in diesem Buch fast ausschließlich mit abstrakten Zeichen als den elementaren zu kodierenden Einheiten eines Schriftsystems. Glyphen spielen bei der Formatierung von Texten eine Rolle und liegen außerhalb unseres Themenspektrums.

3.2 Kodetabelle

Auf der nächsten Stufe des Kodierungsmodells steht die "Kodetabelle", die den abstrakten Zeichen im Zeichensatz eine "Kodeposition" zuweist. Eine Kodeposition ist immer eine natürliche Zahl größer oder gleich null.

Der "Koderaum" einer Zeichenkodierung ist immer ein Abschnitt der nicht-negativen natürlichen Zahlen, der die Null und alle Kodepositionen enthält. Der Koderaum kann jedoch auch Zahlen enthalten, die keine zulässigen Kodepositionen sind.

Kodepositionen werden üblicherweise in Hexadezimalnotation angegeben, wodurch auch eine natürliche Beziehung zu Bitmustern hergestellt wird. Diese Beziehung kommt auf der nächsten Stufe des Kodierungsmodells, dem Kodierungsformat, zum Tragen. Wir wiederholen hier kurz die wesentlichen Fakten der Binärkodierung und der Hexadezimalnotation.

Ein "Bit" ist eine der beiden Ziffern 0 oder 1. Ein "Byte", manchmal auch Oktett genannt, ist eine Sequenz von 8 Bits. Ein "Wyde" ist eine Sequenz von zwei Bytes oder sechzehn Bits.

Die sechzehn möglichen Sequenzen von vier Bits bezeichnen wir in der üblichen Weise mit den Ziffern 0,...,9 und den Buchstaben A,...,F, auch "Hex-Ziffern" genannt. Wir drucken die Bits 0 und 1 fett, um

sie von den Hex-Ziffern 0 und 1, also 4-Bit-Folgen, zu unterscheiden. Die Hex-Ziffern 0,...,9,A,...,F haben die natürlich-zahligen Werte von null bis fünfzehn. Damit kann dann auch jede Hex-Zahl, also jede Sequenz von Hex-Ziffern als natürliche Zahl im Hexadezimalsystem aufgefaßt werden, nämlich als:

$$h_0 + h_1 16 + h_2 16^2 + \dots + h_n 16^n$$

Wobei die Zeichenfolge "16" für die natürliche Zahl sechzehn in Dezimalschreibweise steht. Tabelle 3.3 drückt diese Beziehung zwischen numerischen Werten und Bitmustern aus.

Wir differenzieren bei einer Hex-Zahl nicht zwischen dem numerischen Wert und dem Bitmuster, das sie repräsentiert. Es sollte jeweils aus dem Kontext klarwerden, welche Interpretation gemeint ist.

Tabelle 3.3. Die sechzehn Hex-Ziffern und ihre Repräsentationen als Bitmuster

Hex-Ziffer	Bitmuster	Wert in Dezimalschreibweise
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

Bytes oder Zahlen von 0 bis 255 (dezimal) können wir somit durch Sequenzen von zwei Hex-Ziffern darstellen und Wydes oder Zahlen von 0 bis 65.535 (dezimal) durch Sequenzen von vier Hex-Ziffern.

Der fortlaufende Text enthält sowohl Dezimaldarstellungen als auch Hexadezimaldarstellungen von Zahlen parallel, ohne syntaktisch zwischen ihnen zu unterscheiden. Um Missverständnisse zu vermeiden, erscheint an einigen Stellen das Wort "dezimal" in Klammern hinter einer Dezimaldarstellung; an anderen Stellen erscheint der Wert der Zahl in Worten (z.B. "sechzehn").

Tabelle 3.4. Koderäume für verschiedene Kodierungen

Kodierung	Koderaum	Zahl der Kodepositionen (dezimal)
US-ASCII	0-7F	128
ISO 8859-X	0-FF	256
Unicode	0-10FFFF	65.536+1.048.576
ISO/IEC 10646	0-7FFFFFFF	32.147.483.648

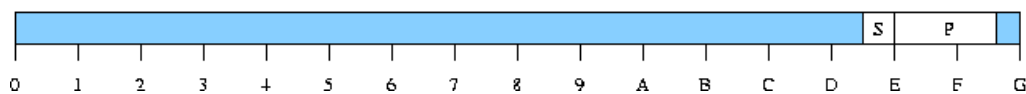
Wie Tabelle 3.4, „Koderäume für verschiedene Kodierungen“ auflistet, umfaßt der Koderaum für US-ASCII die 128 (dezimal) Positionen von 0 bis 7F, der für ISO 8859-X die 256 (dezimal) Positionen

von 0 bis FF, der für Unicode die 65.536 (dezimal) Positionen von 0 bis FFFF plus die 1.048.576 (dezimal) Positionen von 10000 bis 10FFFF und der für ISO/IEC 10646 die 2.147.483.648 (dezimal) Positionen von 0 bis 7FFFFFFF.

Die Koderäume können weiter strukturiert sein: Die Positionen von 0 bis 1F und die Position 7F von US-ASCII sind beispielsweise Kontrollzwecken vorbehalten, ebenso die Positionen 80 bis 9F von ISO 8859-X. Beide Koderäume können durch ein Byte repräsentiert werden.

Der Unicode-Koderaum (Abbildung 3.5, „Der Unicode-Coderaum mit Surrogatpositionen (S) und privatem Bereich (P)“) enthält Lücken an den 2.048 (dezimal) sogenannten Surrogatpositionen von D800 bis DFFF und an den Positionen FFFE sowie FFFF. Diese Positionen sind für spezielle Funktionen vorgesehen, die wir unter den Aspekten Kodierungsformat und Kodierungsschema noch genauer besprechen werden; ihnen werden auch künftig keine Unicode-Zeichen zugeordnet werden.

Abbildung 3.5. Der Unicode-Coderaum mit Surrogatpositionen (S) und privatem Bereich (P)



Die 6.400 Unicode-Kodepositionen von E000 bis F8FF, knapp 10 Prozent des Koderaums von 0 bis FFFF, sind für private Zwecke reserviert. Die Positionen sind etwa für Logos, für Icons oder für in speziellen Notationen (Musik, Tanz) benötigte Zeichen außerhalb des Unicode-Zeichensatzes freigegeben. Die Kodepositionen in diesem Bereich werden auch in späteren Unicode-Ergänzungen für private Verwendung freigehalten.

Von den übrigen 57.086 (dezimal) Unicode-Positionen im Bereich von 0 bis FFFF sind 65 (dezimal) mit Kontrollzeichen belegt, ist 49.194 (dezimal) Positionen ein druckbares Zeichen zugeordnet und sind 7.827 (dezimal) Positionen noch frei für Erweiterungen von Unicode. Wir werden im nächsten Abschnitt bei der Diskussion von Kodierungsformaten sehen, wie der Unicode-Koderaum im Bereich von 0 bis FFFF durch zwei Bytes und im Bereich von 10000 bis 10FFFF durch vier Bytes repräsentiert wird. Derzeit, d.h. bis zur Version 3.0, enthält die Unicode-Kodetabelle nur Werte im Bereich von 0 bis FFFF.

Der Koderaum von ISO/IEC 10646 ist konzeptuell unterteilt in 128 Gruppen von 256 Ebenen, wobei jede Ebene 256 Spalten mit je 256 Seiten, also insgesamt 65.536 Zeichen, enthält. Derzeit weist ISO/IEC 10646-1:2000 nur den Kodepositionen von 0 bis FFFF, die in Ebene 0 von Gruppe 0 liegen, Werte zu. Diese Ebene heißt auch "Basic Multilingual Plane" ("BMP"). Die Kodetabellen von Unicode Version 3.0 und ISO/IEC 10646-1:2000 sind Position für Position identisch. Das Unicode Konsortium und die ISO haben sich verpflichtet, diese Übereinstimmung auch bei späteren Ergänzungen der Kodetabelle aufrechtzuerhalten; sie haben eine Organisationsform wechselseitiger Liaisons geschaffen, die dies garantieren kann.

Unicode beinhaltet für jedes in seiner Kodetabelle kodierte Zeichen die folgenden Daten:

- Die Kodeposition des Zeichens.
- Ein typisches Glyphenbild für das Zeichen.Einen Namen.
- Einen Namen.
- Semantische Information.

Die semantische Information ist eine Spezialität von Unicode. ISO/IEC 10646 stellt diese Information nicht zur Verfügung.

Für die Kodeposition 00AF steht beispielsweise in Unicode die folgende Information zur Verfügung: Das Zeichen hat (1) den Unicode-Namen "Macron", (2) alternative Namen "Overline" und "APL

Overline" und ist (3) graphisch ein hochgestellter waagerechter Strich. Das Macron bewirkt (4) einen horizontalen Vorschub ("Spacing Character") und steht (5) in Beziehung zu weiteren Unicode-Zeichen, u.a. dem Zeichen 0304, "Combining Macron", das genauso aussieht wie das Macron selbst, aber beim Formatieren keinen Vorschub bewirkt ("Nonspacing Character") sondern sich über das vorangehende Basiszeichen positioniert. Das Makron kann (6) zerlegt werden in das Leerzeichen 0020, gefolgt von dem Combining Macron 0304. Das Macron 00AF ist (7) im Unterschied zum Combining Macron kein "Kombinationszeichen" sondern ein "Basiszeichen".

Das Macron wurde, wie viele andere Unicode-Zeichen, aus "Kompatibilitätsgründen" in die Kodetabelle aufgenommen. Hier wirkt sich das Designziel der Konvertierbarkeit von existierenden Standards nach Unicode und zurück aus. Ein Zeichen, das nur aus Kompatibilitätsgründen in der Kodetabelle steht, läßt sich gemäß einer Unicode-Vorgabe immer zerlegen in eine Sequenz aus sogenannten "Primärzeichen".

Für Basiszeichen, die von Kombinationszeichen (Akzente und andere Modifikatoren) gefolgt sind, definiert Unicode einen Äquivalenzbegriff. Grob gesagt können Kombinationszeichen, die sich graphisch an unterschiedlichen Stellen am Basiszeichen positionieren, miteinander ausgetauscht werden. Beispielsweise können die beiden Kombinationszeichen 0307 ("Combining Dot Above") und 0323 ("Combining Dot Below") ihre Plätze tauschen, so daß die beiden Kodierungen 006F 0307 0323 und 0067 0323 0307 für ein "o" mit einem Punkt über sich und einem Punkt unter sich miteinander äquivalent sind.

Auf dem Äquivalenzbegriff für Zeichenfolgen und der Zerlegung von Kompatibilitätszeichen in Folgen von Primärzeichen und umgekehrt der Komposition von Kompatibilitätszeichen aus Folgen von Primärzeichen beruhen auch zwei Normalisierungen von Unicode Zeichenketten, nämlich die "Precomposed Form" und die "Decomposed Form".

Webtexte sollten stets Zeichen benutzen, die schon soweit wie möglich zusammengesetzt sind; verbleibende Kombinationszeichen sollten in normierter Reihenfolge angegeben werden. Webanwendungen sollten so liberal wie möglich sein bei Texten, die sie einlesen, und so konservativ wie möglich bei Texten, die sie ausgeben, um Kompatibilitätsprobleme zu vermeiden.

Von den zehn Designprinzipien, die Unicode anführt, lassen sich fünf auf der Ebene der Kodetabelle erklären:

1. Unifikation: Unicode repräsentiert Zeichen, die in verschiedenen Alphabeten vorkommen, aber vom Aussehen her ähnlich sind, nur einmal.
2. Konvertierbarkeit zwischen etablierten Standards: Eine eindeutige Abbildung aus einem etablierten Standard nach Unicode soll ermöglicht werden. Zeichenpositionen aus einem einzelnen international verbreiteten Zeichensatz, die nach dem Designprinzip der Unifikation eigentlich miteinander identifiziert werden müßten, erhalten trotzdem getrennte Unicode-Kodepositionen.
3. Semantik: Unicode definiert semantische Eigenschaften für Zeichen.
4. Dynamische Komposition: Jedes Basiszeichen kann mit beliebig vielen Kombinationszeichen gleichzeitig gepaart werden.
5. Charakterisierung äquivalenter Kodierungen: Für die verschiedenen Kodierungen eines Basiszeichens mit Kombinationszeichen oder eines primären und eines aus Kompatibilitätsgründen in den Zeichensatz aufgenommenen Zeichens kann eine normalisierte Kodierung gefunden werden.

3.4 Kodierungsformat

Ein "Kodierungsformat" für eine Kodetabelle legt Bitrepräsentationen für die Kodeposition fest. Dazu bedient es sich einer "Kodeeinheit", die in der Regel aus acht oder sechzehn Bits besteht. Ein Kodierungsformat bildet dann die Positionen eines Koderaums in Sequenzen von Kodeeinheiten und somit in Bitmuster ab. Wird jede Kodeposition einer Kodetabelle auf die gleiche Anzahl von

Kodeeinheiten abgebildet, sprechen wir von einem Kodierungsformat "fester Länge"; andernfalls hat das Kodierungsformat "variable Länge".

Ein kanonisches Kodierungsformat fester Länge ergibt sich aus der Dualzahldarstellung von Kodepositionen. Die Anzahl der jeweils benötigten Kodeeinheiten ergibt sich aus der Größe des Koderaums und der Zahl der Bits in einer Kodeeinheit.

Tabelle 3.5, „Kodierungsformate für verschiedene Kodierungen“ gibt eine Übersicht über die verschiedenen Kodierungsformate.

Tabelle 3.5. Kodierungsformate für verschiedene Kodierungen

Kodierung/Kodierungsformat	Koderaum/Kodeeinheit/Länge
US-ASCII/kanonisch	0-7F/1 Byte/fest (1)
ISO 8859-X/kanonisch	0-FF/1 Byte/fest (1)
ISO/IEC 10646, Ebene 0/UCS-2	0-FFFF/1 Wyde/fest (1)
ISO/IEC 10646/UCS-4	0-7FFFFFFF/2 Wyde/fest (1)
Unicode 3.0/UTF8	0-10FFFF/1 Byte/variabel (1-4)
ISO/IEC 10646/UTF8	0-7FFFFFFF/2 Byte/variabel (1-6)
Unicode 3.0/UTF16	0-10FFFF/1 Wyde/variabel (1-2)

Für US-ASCII bzw. seine ISO 646-Varianten sowie für die ISO 8859-X-Kodetabellen ist das kanonische Kodierungsformat das einzig gebräuchliche. Für die höchstens 256 (dezimal) vielen Positionen genügen eine Kodeeinheit von acht Bits sowie eine Kodeeinheit pro Zeichenposition.

Die beiden kanonischen Kodierungsformate für die Koderräume von 0 bis FFFF bzw. von 0 bis 7FFFFFFF heißen UCS2 und UCS4. Beide Kodierungsformate repräsentieren eine Kodeposition mit genau einer Kodeeinheit; im Falle von UCS2 ist die Kodeeinheit 16 Bits lang und im Falle von UCS4 ist sie 32 Bits lang.

Prinzipiell kann eine Kodierung mehrere Kodierungsformate definieren und das kanonische Kodierungsformat muß nicht darunter sein. Unicode 3.0 beispielsweise definiert nur zwei Kodierungsformate, nämlich UTF8 und UTF16, wobei UTF für "Unicode Transfer Format" steht. UTF8 hat eine Kodeeinheit von 8 Bits Länge und repräsentiert Kodepositionen mit einer bis vier Kodeeinheiten. UTF16 hat eine Kodeeinheit von 16 Bits Länge und repräsentiert Kodepositionen mit einer bis zwei Kodeeinheiten.

Wir werden uns zunächst mit UTF16 beschäftigen. Ursprünglich war Unicode für einen Koderaum von 0 bis FFFF ausgelegt, so daß sechzehn Bits genügt hätten, um Kodepositionen darzustellen. Tatsächlich hätte dieser Koderaum ja auch mindestens bis Version 3.0 ausgereicht. Als jedoch klar wurde, daß zumindestens für alle historisch interessanten Sprachen und Alphabete der Koderaum nicht ausreichen würde, und weil man mit ISO/IEC 10646 und seinem größeren Koderaum kompatibel bleiben wollte, wurde das Konzept der Surrogat-Kodepositionen eingeführt und der Koderaum von Unicode bis zur Position 10FFFF erweitert. Das Kodierungsformat UTF16 benutzt jeweils ein Paar von Surrogatpositionen, um Positionen jenseits von FFFF darzustellen.

UTF16 stellt wie die kanonische Kodierung jede gültige Kodeposition im Bereich von 0 bis FFFF durch ein einziges Wyde dar. Die ungültigen Kodepositionen in dem sogenannten hohen Surrogatbereich von D800 bis DBFF und in dem niedrigen Surrogatbereich von DC00 bis DFFF entsprechen den hohen und niedrigen Surrogat-Wydes der Form 110110xxxxxxxxxx und 110111xxxxxxxxxx, in denen jeweils zehn Bitpositionen für 2^{10} verschiedene Werte frei wählbar sind. Die Gegenrechnungen $DBFF+1-D800=DC00-D800=400=4 \times 16^2$ (dezimal) $=2^{10}$ (dezimal) und $DFFF+1-DC00=E000-DC00=400=2^{10}$ (dezimal) bestätigen, daß in jedem der beiden Surrogatbereiche 2^{10} , also 1024 (dezimal) ungültige Kodepositionen liegen. Ein Paar von Surrogatpositionen, von denen die erste im hohen und die zweite im niedrigen Bereich liegen, ist somit eindeutig charakterisiert durch zwanzig Bits, die wiederum die 2^{20} Werte von 10000 bis 10FFFF repräsentieren können: $10FFFF+1-10000=110000-10000=100000=2^{20}$ (dezimal).

Es ist deshalb stimmig, wenn UTF16 eine Kodeposition P im Bereich von 10000 bis 10FFFF durch die kanonische Bitdarstellung der beiden hohen und niedrigen Surrogatpositionen "H" und "L" repräsentiert, wobei sich "H" und "L" wie folgt berechnen:

Tabelle 3.6. Schema für UTF8-Kodierung

Einheiten	Darstellungsform	freie Bits	Maximum
1	0xxxxxxx	7	7F
2	110xxxxx 10xxxxxx	11	7FF
3	1110xxxx (10xxxxxx) ²	16	FFFF
4	11110xxx (10xxxxxx) ³	21	1FFFFF
5	111110xx (10xxxxxx) ⁴	26	3FFFFFFF
6	1111110x (10xxxxxx) ⁵	31	7FFFFFFF

$$H = (P - 10000) \text{DIV} 400 + D800, L = (P - 10000) \text{MOD} 400 + DC00.$$

Umgekehrt bestimmen eine hohe und eine niedrige Surrogatposition "H" und "L" eine Unicode-Position "P" im Bereich von 10000 bis 10FFFF wie folgt:

$$P = (H - D800) \cdot 400 + (L - DC00) + 10000.$$

Das zweite Unicode-Kodierungsformat, nämlich UTF8, hat die besondere Eigenschaft, die ersten 128 Kodepositionen kanonisch mit einem Byte darzustellen. UTF8 ist damit US-ASCII-transparent.

Die Kodierungseinheit von UTF8 ist 8 Bits lang. UTF8 stellt jede Position im Bereich von 0 bis FFFF mit ein bis drei Kodierungseinheiten, im Bereich von 0 bis 10FFFF mit ein bis vier Kodierungseinheiten und im Bereich von 0 bis 7FFFFFFF mit ein bis sechs Kodierungseinheiten dar. UTF8 kann somit im Unterschied zu UTF16 den vollen Koderaum von ISO/IEC 10646 kodieren.

Die UTF8-Darstellung einer Position mit einer einzigen Kodierungseinheit hat die Form 0xxxxxxx mit einer führenden 0 und beliebigen Bits an den mit x markierten Stellen. Die UTF8-Darstellung einer Position mit n Kodierungseinheiten, $1 < n < 6$, hat die Form:

$1^n 0x^{7-n}$, gefolgt von $n - 1$ Kodierungseinheiten der Form 10xxxxxx.

Bei drei Kodierungseinheiten sind also genau sechzehn Bits frei wählbar, so daß mit drei Kodierungseinheiten unter UTF8 Positionen bis FFFF darstellbar sind.

Die mit n Kodierungseinheiten darstellbaren Positionsbereiche lassen sich aus der Tabelle 3.6, „Schema für UTF8-Kodierung“ entnehmen:

Für die Darstellung einer Kodeposition unter UTF8 muß immer die kleinstmögliche Zahl von Kodierungseinheiten gewählt werden. Eine Kodeposition im Bereich von 800 bis FFFF muß also mit drei Kodierungseinheiten und darf nicht -obwohl das technisch möglich wäre- mit vier Kodierungseinheiten dargestellt werden.

Die Umkehrfunktion von UTF8, die aus einer Folge von Kodeeinheiten eine Kodeposition berechnet, darf jedoch auch zu lange Darstellungen korrekt umrechnen. Die zwei Bytes C1BF dürfen also unter UTF8 in die Position 7F zurückgerechnet werden, obwohl die UTF8-Darstellung der Position 7F das Byte 7F ist.

Das Unicode-Designprinzip der Effizienz läßt sich auf Ebene von Kodierungsformaten erläutern. Für jedes der beiden Unicode-Kodierungsformate UTF8 und UTF16 läßt sich in einem Strom von Positionsdarstellungen von jeder Kodierungseinheit aus der Anfang der zugehörigen Positionsdarstellung mit beschränktem Backup finden. Im Falle von UTF16 muß höchstens um eine Kodierungseinheit zurückgegangen werden, im Falle von UTF8 höchstens um drei bzw. fünf Positionen, je nach Größe des zu kodierenden Koderaums.

3.5 Kodierungsschema

Damit Daten über Netzwerke zuverlässig ausgetauscht werden können, müssen sie in eine Folge von Bytes serialisiert werden. Mit Hilfe eines Kodierungsformats ist es bis jetzt gelungen, einen Text als Folge von Kodierungseinheiten darzustellen. Ein Kodierungsschema hat nun die Aufgabe, zu einem Kodierungsformat zusätzlich festzulegen, wie die Kodierungseinheiten in Bytefolgen zu serialisieren sind.

Ist die Kodierungseinheit eines Kodierungsformats selbst schon acht Bits lang, so ist nichts mehr festzulegen und das Kodierungsschema ist mit dem Kodierungsformat identisch. Wir können also UTF8 sowohl als Kodierungsformat als auch als Kodierungsschema ansehen.

Ist die Kodierungseinheit ein Byte, wie bei UTF16, so gibt es zwei Möglichkeiten der Serialisierung: "big endian" (das höherwertige Byte kommt zuerst) und "little endian". Dementsprechend gibt es zu UTF16 zwei Kodierungsschemata, genannt UTF16-BE und UTF16-LE. Analoges gilt für UCS2 und UCS4.

Mit dem "Byte Order Mark" ("BOM") an Position FEFF kann ein UTF16-repräsentierter Datenstrom signalisieren, welche der beiden Serialisierungen, "big endian" oder "little endian", vorliegt. Da FFFE keine zulässige Kodeposition in Unicode ist, läßt sich, wenn als die ersten zwei Bytes FF und FE gelesen werden, schließen, daß das Kodierungsschema UTF16-LE vorliegt. Ein BOM am Anfang eines UTF16-kodierten Datenstroms ist nicht Bestandteil der Daten und wird -abgesehen von seiner Signalwirkung für die Serialisierung- ignoriert. Tritt die Kodeposition FEFF dagegen an anderer Stelle im Datenstrom auf, so wird sie als das entsprechende Unicode-Zeichen "Zero Width No-Break Space" interpretiert, das dieser Position zugewiesen ist.

Generell darf ein Unicode-Zeichenstrom am Anfang mit dem zusätzlichen Zeichen BOM versehen werden, um Anwendungen, die über keine Metainformation über die Natur des Datenstroms verfügen, zu signalisieren, daß es sich um Unicode-Daten handelt und welches Kodierungsschema vorliegt. Das erste BOM eines Zeichenstroms bildet dann keinen Teil der eigentlichen Daten.

3.6 Metainformation über Kodierungen: MIME

3.7 Unicode-Werkzeuge

Bei der Diskussion von Werkzeugen zur Bearbeitung von Unicode-Texten unterscheiden wir zwischen der Eingabe und der Darstellung von Unicode-Zeichen.

Wenn Sie einen genuinen Unicode-Editor wünschen, ist die Auswahl derzeit noch nicht allzu groß. Eine der wenigen Optionen ist ein Forschungsprototyp von der Duke University (<http://www.lang.duke.edu/unidemo/unidemo.html>) namens UniEdit, der eine Vielzahl von Kodierungsschemata und Eingabeformaten unterstützt.

Am anderen Ende des Spektrums, was Unicode-Unterstützung angeht, können Sie jeden beliebigen Editor für 7-Bit-ASCII hernehmen und Ihre damit erstellten Texte als Unicode-Texte in UTF8-Kodierung ausgeben. Damit können Sie direkt natürlich auch nur die 7-Bit-ASCII-Zeichen in Ihren Texten verwenden. Indirekt haben Sie über eine weitere Kodierungsstufe jedoch auch Zugriff zu dem vollen Unicode-Zeichenrepertoire. Im Falle der Java-Kodierung steht Ihnen auch ein Werkzeug zur Verfügung, mit dem Sie die Java-Notation wieder dekodieren und in verschiedene Kodierungsformate übersetzen können. Das Werkzeug heißt native2ascii und ist Bestandteil des Java Software Development Kit.

3.8 Jenseits von glattem Text

Wir haben uns in den bisherigen Abschnitten zu Zeichenkodierungen darauf konzentriert, wie wir glatten Text als Folge von Unicode-Zeichen kodieren können. Damit können wir den reinen Inhalt von

strukturierten Dokumenten handhaben. Da auch eingebettetes Markup, z.B. die in dem Abschnitt 2.7 und in dem Beispiel A.2, „Repräsentation der logischen Struktur mit Hilfe von eingebettetem Markup“ eingeführten Tags der Form `<xxx>` und `</xxx>`, eine Folge von Zeichen ist, scheint es auf den ersten Blick, als hätten wir das Problem, strukturierte Dokumente zu kodieren, bereits vollständig gelöst.

Es ergibt sich jedoch eine zusätzliche Komplikation, da Markuptext von Inhaltstext syntaktisch unterscheidbar sein muß. Zur Abgrenzung von Markup und Inhalt werden bestimmte Funktionszeichen eingesetzt, die dann weder im Inhalt noch, außerhalb ihrer syntaktischen Rolle, im Markup im Klartext vorkommen dürfen. In XML ist `"<"` ein solches Funktionszeichen, das den Anfang eines Tags charakterisiert. Das Zeichen `"<"` darf im Inhaltstext eines XML-Dokuments nicht vorkommen.

Die klassische Methode, Funktionszeichen in glattem Text unterzubringen, ist die Verwendung von sogenannten "Escape"-Zeichen, die selbst nicht wörtlich als Bestandteil des Textes verstanden werden sondern die Präsenz eines Zeichens signalisieren, das nicht direkt im Text vorkommen darf. Im Falle von XML bezeichnet die Formel `&#x<Hexziffern>`; im Inhaltstext das Unicode-Zeichen an Position `Hexziffern`. Wir können also ein `"<"` im Inhaltstext als `<` notieren und das zusätzliche Funktionszeichen `"&"` durch `&`.

Im allgemeinen wird uns also glatter Text nicht direkt als Folge von Zeichen begegnen. Vielmehr werden wir die Zeichenfolge, die unseren Text ausmacht, konstruieren aus einer weiteren Folge von sogenannten Eingabezeichen, von denen ganze Teilsequenzen einzelne Zeichen unseres Textes repräsentieren. In der Praxis haben wir es also mit einer zweistufigen Kodierung von Text zu tun.

Die hier dargestellte Technik, Zeichen entweder durch sich selbst oder durch spezielle Zeichenfolgen wie `&#x<Hexziffern>`; zu kodieren, löst insgesamt drei bekannte Probleme:

Das erste Problem, wie sich in glattem Text Funktionszeichen an Stellen unterbringen lassen, an denen sie nicht im Klartext vorkommen dürfen, haben wir bereits angesprochen.

Dieselbe Technik ermöglicht es zweitens auch, Zeichen in einen Text einzufügen, die das verwendete Eingabewerkzeug nicht unterstützt. Sie können also beispielsweise in ein XML-Dokument mit einer amerikanischen Tastatur den dort nicht unterstützten Umlaut `"ä"` als `ä` eingeben.

Drittens ist es üblich, Paare aus den beiden ASCII-Steuerzeichen LF ("linefeed", "newline") und CR ("carriage return") sowie jedes der beiden Zeichen für sich allein als ein- und dasselbe Zeilenendezeichen zu interpretieren. Auf diese Weise werden die beiden gängigen verschiedenen Konventionen, im Text ein Zeilenende zu signalisieren, vereinheitlicht. Werkzeuge, die die Texte verarbeiten, verfügen damit über eine plattformunabhängige Methode, Zeilen zu nummerieren.

Wir haben bisher die Technik, Zeichen durch Zeichenfolgen spezieller Syntax zu kodieren, am Beispiel von XML illustriert. Die Technik ist aber Standard in allen Bereichen, in denen Texte erstellt werden, speziell auch bei der Erstellung von Programmtexten.

Java-Programme beispielsweise dürfen beliebige Unicode-Zeichen der "Basic Multilingual Plane", also im Codebereich von 0 bis FFFF, enthalten, z.B. in Namen, in Literalen für Zeichenketten oder in Kommentaren. Grundsätzlich kann im Programmtext jedes Zeichen durch sich selbst oder durch eine Sequenz `\u<xxxx>` mit einer ungeraden Anzahl von Zeichen `"\"`, einer positiven Anzahl von Zeichen `"u"` und einer Sequenz `<xxxx>` von genau vier Hexziffern dargestellt werden. Die einzige Ausnahme betrifft das Zeichen `"\"`, das nur dann für sich selbst stehen darf, wenn es nicht in einer Sequenz der Form `\u<xxxx>` vorkommt und somit misinterpretiert werden kann.

Das klassische Java-Programm HelloWorld

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}
```

dürfte also auch in folgender Form erscheinen:

```
\u0070ublic class HelloWorld {  
public static void main(String[] args) {  
System.out.println("\uuuu0048ello World");  
\\\u007D  
}
```

Die Java-Notation macht es also möglich, einen beliebigen Unicode-Text mit Zeichen im Bereich zwischen 0 und FFFF als einen 7-Bit-ASCII-Text zu kodieren.

3.9 Elementare Typographie

3.10 Schrifttechnologie

3.11 Zusammenfassung und Ausblick

Wir haben in diesem Kapitel gesehen, wie sich glatter Text ("plain text") mit Unicode standardisiert kodieren läßt. Bei Markupsprachen kodiert Unicode auch die in den glatten Text eingebetteten Markierungen, also die Tags, Referenzen und Deklarationen. "Fancy text", wie es im Unicode-Standard heißt, ist im Modell der strukturierten Dokumente das Ergebnis eines Formatierprozesses, der, von einem Stylesheet gesteuert, den Inhaltstext mit typographischen Attributen versieht, und zwar in Abhängigkeit von den semantischen Rollen, die die Struktur für die entsprechenden Textstücke festlegt; dabei erfolgt u.a. eine Umsetzung von abstrakten Zeichen in Glyphenbilder.

Wir haben in diesem Kapitel zwei weitere Informatik-Prinzipien angewendet gesehen, die ich zum Abschluß noch kurz diskutieren möchte.

1. "Levels of Abstraction." "Levels of Abstraction" einzuführen bedeutet, einen komplexen Sachverhalt ausgehend von einer hohen Abstraktionsstufe auf immer niedrigeren Abstraktionsstufen zu beschreiben und auf diese Weise einen konzeptuellen Rahmen herzustellen, in dem der Sachverhalt verstanden werden kann. Eine typische Anwendung des Prinzips ist die Rechnerarchitektur, in der ein Rechensystem auf den Ebenen Hardware, Maschinensprache, Betriebssystem und Anwendungssystem beschrieben wird.

In diesem Kapitel haben wir das Prinzip der "Levels of Abstraction" im Zusammenhang mit Kodierungen kennengelernt, die im Rahmen des Kodierungsmodells auf immer konkreterer Ebene beschrieben wurden.

2. Notwendigkeit von Interpretation. Zu jeder Kodierungsmethode von Daten oder Information gehört eine Interpretation, die jedem Kodifikat seine Bedeutung zuweist. Aus einer Folge von Bits ist ohne Metainformation, nach welcher Methode kodiert wurde, nicht abzulesen, was die Bedeutung der Bitfolge ist. Die Bitfolge FF kann zum Beispiel in Dualzahldarstellung die Zahl 256 und in Zweikomplementdarstellung die Zahl -1 kodieren. Oder sie kodiert, in einem anderen Kontext, das Zeichen "#" an Position 255 im Zeichensatz ISO Latin-1. Negroponte spricht in diesem Zusammenhang von der Universalität des Bit.

In diesem Kapitel haben wir verschiedene Regeln kennengelernt, nach denen sich Zeichen und glatte Texte als Bitfolgen kodieren lassen. Die Metainformation für die Interpretation eines solchen Kodifikats ist u.a. mit MIME ausdrückbar.

Kapitel 4. XML

4.1 Orientierung

In dem vergangenen Kapitel 2 haben wir das Konzept und die vielfältige Welt moderner Dokumente mit dem sogenannten Modell der strukturierten Dokumente kennengelernt. Der Kern dieses Kapitels wird die Möglichkeit sein, wie man so ein modernes strukturiertes Dokument tatsächlich konkret kodieren kann und wie man es in einer Computerdatei hinschreibt. Dabei werden wir uns etwas genauer mit dem Basisstandard dieser Vorlesung (XML) beschäftigen.

4.2 Schlüsseltechnologie XML

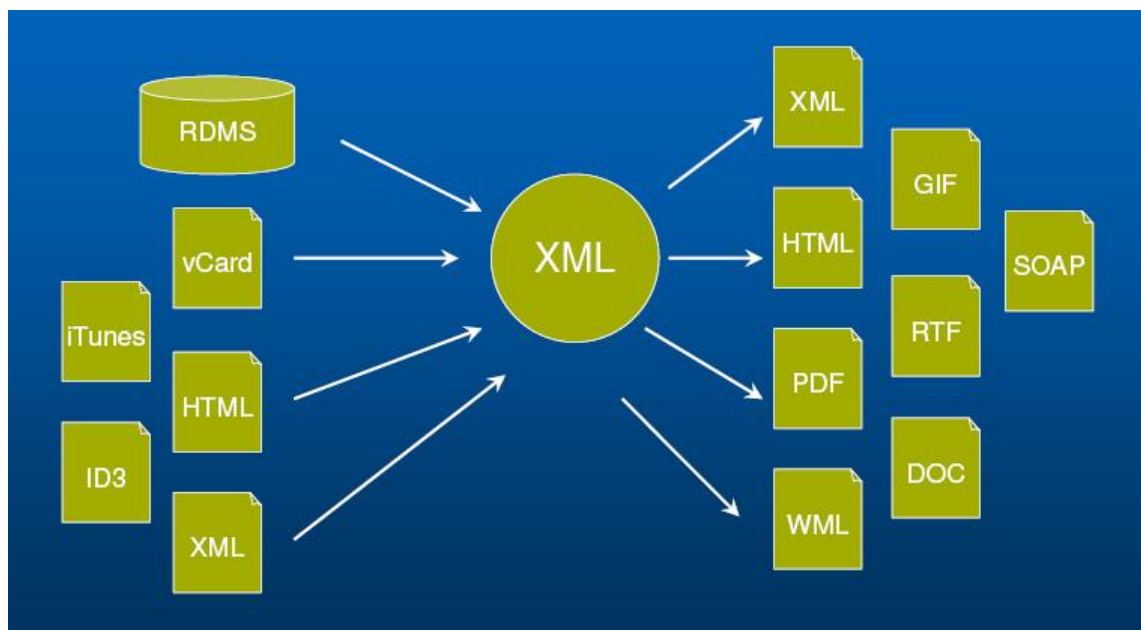
Zunächst ist XML eine Metasprache für Markup Sprachen. Dies bedeutet, dass sie eine Sprache ist, mit der sogenannte Markup Sprachen beschrieben werden können. Technisch gesehen bildet XML eine Untermenge der seit 1986 ISO standardisierten Sprache SGML.

XML definiert ein Format¹, mit dem man strukturierte Dokumente kodieren kann. Das besondere an XML ist nicht die Art, wie es gemacht ist, sondern die Tatsache, dass diese Sprache zu dem vom W3C standardisierten Austauschformat für Dokumente geworden ist, das eine sehr große Verbreitung gefunden hat.

Als standardisiertes Austauschformat für Dokumente hat XML das Potential als Drehscheibe zu fungieren (Siehe Abbildung 4.1, „Drehscheibencharakter von XML“). Das heißt, dass das XML-Format als Quellformat für das Single Source Publishing angesehen wird, einerseits weil es die Möglichkeit bietet, interne Daten aus verschiedenen Systemen zu importieren und diese dann durch verschiedenen Satelliten-Technologien, beispielsweise Transformieren mit XSLT oder Abfragen mit XQuery, zu verarbeiten, andererseits sind auch Werkzeuge, mit denen die XML-Daten in verschiedenen Präsentationsformate, beispielsweise PDF oder HTML, übersetzt werden können, vorhanden.

Somit hat man für Publikationsanwendungen und für moderne Dokumente im Grenzbereich von Daten und Dokumenten eine geeignete Basis, die dieses Dokumentenmodell der strukturierten Dokumente umsetzt und für die notwendige Flexibilität sorgt.

Abbildung 4.1. Drehscheibencharakter von XML



¹Dateiformat, das den Aufbau einer Datei beschreibt.

4.3 XML (Extensible Markup Language) am Beispiel

```
<?xml version="1.0" encoding="UTF-8"?>
  <biographie id="Mozart">
    <name>Mozart</name>
    <vorname>Joannes Chrysostomus Wolfgangus Theophilus</vorname>
    <geburtsdatum>27. Januar 1756</geburtsdatum>
    <geburtsort>Salzburg</geburtsort>
    <beruf>Komponist</beruf>
    <werkliste>
      <oper id="35">
        <titel>Die Schuldigkeit des ersten Gebots</titel>
        <entstehungszeit>1767</entstehungszeit>
        <köchelverzeichnis>KV 35</köchelverzeichnis>
      </oper>
      <oper id="38">
        <titel>Apollo und Hyacinth</titel>
        <entstehungszeit>1767</entstehungszeit>
        <köchelverzeichnis>Kv 38</köchelverzeichnis>
      </oper>
      <kirchenmusik id="220">
        <titel>Spatzenmesse</titel>
        <entstehungszeit>1776</entstehungszeit>
        <köchelverzeichnis>KV 220</köchelverzeichnis>
      </kirchenmusik>
      <kirchenmusik id="317">
        <titel>Krönungsmesse</titel>
        <entstehungszeit>1779</entstehungszeit>
        <köchelverzeichnis>KV 317</köchelverzeichnis>
      </kirchenmusik>
      <orchesterwerke>
        <sinfonie id="17">
          <tonart>B-Dur</tonart>
          <entstehungszeit>1765</entstehungszeit>
          <köchelverzeichnis>KV 17</köchelverzeichnis>
        </sinfonie>
        <sinfonie id="18">
          <tonart>Es-Dur</tonart>
          <entstehungszeit>1765</entstehungszeit>
          <köchelverzeichnis>KV 18</köchelverzeichnis>
        </sinfonie>
        <klavierkonzert id="37">
          <tonart>F-Dur</tonart>
          <entstehungszeit>1767</entstehungszeit>
          <köchelverzeichnis>KV 37</köchelverzeichnis>
        </klavierkonzert>
        <violinkonzert id="261">
          <tonart>Adagio E-Dur </tonart>
          <entstehungszeit>1776</entstehungszeit>
          <köchelverzeichnis>KV 261</köchelverzeichnis>
        </violinkonzert>
      </orchesterwerke>
    </werkliste>
  </biographie>
</xml>
```

```
</biographie>
```

Wie im Quellcodebeispiel "biographie.xml" zu sehen ist, sieht XML der Sprache HTML sehr ähnlich. XML ist keine Markup-Sprache in dem Sinne, dass die Menge der Elemente (sogenannte "Tags") und Attribute vorgegeben und vordefiniert ist, wie z.B. bei HTML. Die Namen der Tags können von XML-Anwender frei festgelegt werden. Die Tag-Namen könnten am besten so gewählt werden, daß sie die Bedeutung des Inhalts ausdrücken. Dabei bleibt die Tag-Syntax wie bei HTML erhalten:

```
<ElementName AttributeName="Wert">Inhalt</ElementName>
```

Wie bereits erwähnt, erlaubt XML ein freies Vokabular, also die freie Wahl der Element- und Attributnamen. Somit lässt sich XML als erweiterbare Markup-Sprache, woraus das Kürzel "X" für eXtensible seine Begründung erhält.

Tipp

Ein leeres Element-Tag kann für jedes Element benutzt werden, das keinen Inhalt hat.

Beispiele für leere Elemente:

```
<ElementName></ElementName>  
<ElementName/>
```

Mag sein, dass das Ausdenken von einem eigenen Vokabular (Element- und Attributnamen) schön und flexibel sein kann. Dieser Vorteil kann sich aber sehr schnell in einen Albtraum umwandeln, wenn die XML-Anwender dieses nicht abstimmen. Aus diesem Grund gibt es die sogenannten Dokumenttypdefinitionen (DTDs), die unter anderem die Element- und Attributnamen festlegen (Siehe Kapitel 6 Schemasprachen (DTD)).

Analog zu HTML können XML-Dokumente auch mit Cascading Style Sheets (CSS) kombiniert werden. Ein normaler Webbrowser kann also ein XML-Dokument mit einem CSS-Stylesheet interpretieren und entsprechend umsetzen.

Fazit

Fasst man die vergangenen Überlegungen kurz zusammen, so lässt sich als Fazit ziehen: "XML, ohne DTDs zu berücksichtigen, ist im Grunde genommen nicht anders als HTML mit freien Elementen, Attributen und Referenzen". Es ist wohl auch deutlich, dass XML als Syntax für strukturierte Dokumente zu bezeichnen ist. Dabei wird XML auch als Metasprache für Markup Sprachen definiert. Die mit XML definierten Markup-Sprachen, wie z.B. XHTML oder DocBook, werden auch als XML-Anwendungen² bezeichnet.

4.4 XML-Syntax: Dokument (Instanz)

In diesem Abschnitt werden einige Syntax-Regeln beschrieben:

- In der Regel wird die XML-Deklaration in der ersten Zeile eines XML-Dokuments angezeigt. Sie lässt sich wie folgt darstellen:

```
<?xml version="1.0" encoding="" standalone=""?>
```

Während das Attribut "Version" die Versionsnummer der XML-Spezifikation definiert, bestimmt "encoding" die Kodierung der XML-Datei. Der Default-Wert lautet "UTF-8". "standalone" wird

²Bei einer XML-Anwendung handelt es nicht um eine konkrete Software, sondern um ein festgelegtes Vokabular für einen spezifischen Anwendungskontext.

selten angegeben und kann nur die Werte "no" und "yes" (Yes, für den Fall, dass das Dokument über keine DTD verfügt) belegen.

Hinweis

Die Groß- und Kleinschreibung im Wert für die Kodierung wird nicht berücksichtigt. „UTF-8“ entspricht „utf-8“.

- Die meisten Tags in XML, wie auch in HTML, treten paarweise als Start- und End-Tags auf:

```
<ElementName AttributeName="Wert">Inhalt</ElementName>
```

- Im XML müssen alle Tags der Form <tag> immer einen End-Tag der Form </tag> haben. Die XML-Tags, zu denen es keinen End-Tag gibt, müssen wie folgt dargestellt werden:

```
<tag/>
```

Diese Art von Tags bezeichnen dann immer ein leeres Element.

Hinweis

Bei XML ist im Gegensatz zu HTML die Groß- und Kleinschreibung zu beachten:

```
<tag> ist ungleich <tAg>.
```

- Schachtelungen: In XML müssen die Start- und Endtags auch richtig geschachtelt werden.

Erlaubt ist:

```
<sinfonien>
  <tonart></tonart>
  <entstehungszeit></entstehungszeit>
  <köchelverzeichnis></köchelverzeichnis>
  ...
</sinfonien>
```

Nicht erlaubt ist:

```
<sinfonien>
  <tonart></sinfonien></tonart>
<entstehungszeit></entstehungszeit>
<köchelverzeichnis></köchelverzeichnis>
...
```

- Parameter: Alle Attribut-Werte sollen immer in Quotes-Zeichen "..." oder '...' eingeschlossen werden.
- Entities: In XML können auch Entities definiert werden, bei denen einem Namen ein bestimmter Text zugeordnet wird. Bekannte und vordefinierte Beispiele sind:

Tabelle 4.1. XML-Entities

Name	Ersetzungstext
<	<
>	>
&	&
"	"
'	'

4.5 Wohlgeformtheit und Validität

Die Wohlgeformtheit stellt nur einen speziellen Aspekt der XML-Syntax dar. In XML sollen die Strukturelemente eine hierarchische Struktur folgen, das heißt entweder sollen die Elemente ganz ineinander enthalten sein oder sie müssen nebeneinander stehen und dabei keine Überlappungen zulassen.

Hinweis

Zur Wohlgeformtheit gehört weiterhin, dass in Elementen keine mehrfachen Attributnamen vorkommen.

Nicht erlaubt wäre beispielsweise:

```
<ElementName AttributeName1="Wert1" AttributeName1="Wert2">
```

Das Gegenstück zu der Wohlgeformtheit ist die Validität, die besagt, dass ein XML-Dokument die, in einer zugehörigen Dokumenttypdefinition oder allgemein in einer Schemasprache, festgelegten Regeln einhalten muss.

4.6 Parser

Anhand eines Parsers oder sogenannten XML-Prozessors kann überprüft werden, ob das XML-Dokument den syntaktischen Regeln des XML-Standards gehorcht und insbesondere, ob die Tag-Klammerung richtig gesetzt ist.

Die wichtigste Aufgabe eines Parsers ist die Überprüfung der Syntax und der Wohlgeformtheit. Ein validierender Parser überprüft zusätzlich die Richtigkeit der Daten anhand einer Schemasprache (DTD, W3C-Schema etc.). Einer der bekanntesten Parser ist "Xerces", ein validierender XML-Parser für C++, Java und Perl für eine große Anzahl an Plattformen aus dem Projekt ApacheXML.

Es gibt zwei wichtige APIs, die von fast allen Parsern implementiert sind. Diese lauten DOM und SAX und werden später näher betrachtet.

Im Web sind auch Online-Parser zu finden. Ein Beispiel kann unter dem folgenden Link näher betrachtet:

<http://www.stg.brown.edu/service/xmlvalid/>

Eine weitere Alternative bieten die Browser anhand der integrierten Parser, die aber nur die Wohlgeformtheit überprüfen können (keine Validität).

4.7 XML-Browser

Die meisten Browser, wie beispielsweise Microsoft Internet Explorer oder Netscape Navigator, haben die Funktionalität, sowohl das XML-Dokument pur als eine hierarchische Struktur, meistens

mit einem Einklapp/Ausklapp Mechanismus, darstellen zu können als auch die Möglichkeit, das Dokument mit einem CSS-Stylesheet versehen zu können (Microsoft Internet Explorer ab Version 5 und Netscape Navigator ab Version 6).

4.8 XML-Editoren

Auf dem Markt werden zahlreiche kommerzielle als auch freie Lösungen solcher Editoren angeboten. Die auf Java basierende oXygen-Software ist ein weitverbreiteter kommerzieller XML-Editor, der als Standalone, sowie im Verbund mit Eclipse als ein mächtiger Editor und Entwicklungsumgebung verfügbar ist.

In der nachfolgenden Tabelle 4.2, „Bekannte XML-Editoren“ ist eine Auflistung bekannter XML-Editoren zu finden:

Tabelle 4.2. Bekannte XML-Editoren^a

XML-Editor	URL-Adresse
oXygen	http://www.oxygenxml.com/
Xmlspy	http://www.altova.com/xmlspy.html
XML Buddy	http://www.xmlbuddy.com/
XMLmind XML Edit	http://www.xmlmind.com/xmleditor/

^a(Aufrufsdatum der Links in der Tabelle:22.12.09)

XML-Editoren bieten unter anderem die folgenden Funktionalitäten an:

- Syntax-Highlighting, Syntaxvervollständigung,
- Navigation in Element-Hierarchie,
- nach Wahl formatierte Darstellung,
- Überprüfen der Wohlgeformtheit, Validierung,
- Integration von Werkzeugen (XSLT, XQuery) und Debuggern,
- spezielle Visualisierungen:
 - tabellenähnliche Dokumente,
 - DTDs und Schema-Dokumente.

4.9 Zusammenfassung

Sowohl SGML als auch XML sind zwei Metasprachen zur Definition von Markup-Sprachen und dienen zur syntaktischen Repräsentation von strukturierten Dokumenten und ihren Strukturvorgaben. Während SGML im Verlagswesen und in der Dokumentation großer Systeme von Bedeutung ist, bietet XML eine vereinfachte Fassung von SGML für das Web an.

Schließlich unterscheidet sich XML sehr stark von HTML, in dem sie an kein festes Vokabular für Elemente, Attribute und Entitäten gebunden ist. Dadurch hat XML jedoch auch keine definierte Semantik für Sprachelemente.

Literatur zu XML:

- Kurs XML der TEIA-AG (<http://www.teialehrbuch.de/XML/>).
- Online-Tutorial von W3Schools (<http://www.w3schools.com/xml/>).

- XML-Portal von O'Reilly (<http://www.xml.com/>).
- XML-Seite des W3C (<http://www.w3.org/XML/>).
- E.R. Harold, W.S. Means: XML in a Nutshell. O'Reilly 2001.
- M. Johnson: XML for the Absolute Beginner (http://www.javaworld.com/javaworld/jw-04-1999/jw-04-xml_p.html).

Kapitel 5. XML Namespaces

5.1 Orientierung

Seit Einführung von XML sind Dutzende von XML-Anwendungen entstanden, also von XML Vokabularen, die teilweise sehr breit genutzt werden. Der Natur des Web entsprechend finden solche Entwicklungen unabhängig voneinander statt. Auf diese Weise entstehen auf natürliche Weise Namenskonflikte; d.h., ein- und derselbe Name hat je nach Verwendungszusammenhang unterschiedliche Bedeutung. Das wird problematisch, wenn verschiedene Vokabularen miteinander kombiniert werden sollen, und eine solche Komponierbarkeit ist ja sehr wünschenswert.

Ein berühmtes Beispiel ist ein Vokabular für Einrichtungshäuser, das XML-Elemente wie `table`, `chair` und `cupboard` einführt und dann für die Präsentation in einem Katalog HTML-Elemente wie `h1`, `p` und `table` wiederverwenden möchte. Ein Browser ist dann vor dem Konflikt gestellt, ob ein Element `table` jetzt für einen Tisch oder für eine Tabelle steht.

Namenskonflikte, wie sie oben beschrieben wurden, tauchen nicht nur im Zusammenhang mit XML auf, und es sind Strategien bekannt, wie man mit ihnen umgehen kann. Dazu gehören Standardisierung von Namen, Namensverwaltung durch eine zentrale Instanz oder einfache Umbenennung von Bezeichnern, die in Konflikt stehen. Gerade im verteilten Web mit seinem vielfältigen und zahlreichen autonomen Akteuren sind diese Ansätze jedoch nicht anwendbar; sie stehen in eklatanten Widerspruch zu Intention und Architektur des Webs.

Ziel dieses Kapitels ist es, mit XML Namespaces eine Möglichkeit zur Behandlung von Namenskonflikten in XML aufzuzeigen, dass im Web skaliert und die Komponierbarkeit von unabhängig voneinander entwickelten Vokabularen unterstützt. Wir führen XML Namespaces als eine hilfreiche Erweiterung von XML ein, die eine friedliche Koexistenz von XML-Vokabularen ermöglicht.

Anhand eines konstruierten Beispiels (mit Namenskonflikten) wird vermittelt, wie identische Namen mit verschiedenen Semantiken in Konflikt stehen, und wie Namesräume die unterschiedlichen Semantiken identischer Elemente markieren können. Der Ansatz von XML Namespaces ist dabei sehr ähnlich zu Paketnamen in Java. Wir erläutern auch, wann ein XML-Dokument zu XML Namespaces konform ist. Abschließend weisen wir—im Vorgriff auf den nächsten Teil—auf die Unverträglichkeit von XML Namespaces und Dokumenttypdefinition hin.

5.2 (Konstruiertes!) Beispiel mit Namenskonflikten

Das folgende Beispiel verdeutlicht die Problematik der Namenskonflikte innerhalb einer XML-Datei:

```
<?xml version="1.0" encoding="UTF-8"?>
<werke id="Penderecki">
  <titel>Prof. Dr.</titel>
  <name>Penderecki</name>
  <vorname>Krzysztof</vorname>
  <datum>23. November 1933</datum>
  <geburtsort>Debica (Polen)</geburtsort>
  <werkliste>
    <werk id="1">
      <titel>The Devils of Loudun</titel>
      <datum>1968</datum>
    </werk>
  </werkliste>
</werke>
```

```
<werk id="2">
  <titel>Paradise Lost</titel>
  <datum>1976</datum>
</werk>
<werk id="3">
  <titel>Die schwarze Maske </titel>
  <datum>1984</datum>
</werk>
</werkliste>
</werke>
```

Das obige Beispiel, das einige werke des berühmten polnischen Komponisten Krzysztof Penderecki auflistet, weist einige Namenskonflikte auf, zum Beispiel die beiden Elementnamen `titel` und `datum` mit zwei verschiedenen Verwendungen.

Zunächst soll angesichts der Mehrdeutigkeit von Namen festgelegt werden, von welcher Art ein benanntes Objekt ist. In dem Beispiel ist es klar erkennbar, dass es sich bei der ersten Verwendung des Elementnamens `titel` um den akademischen Titel handelt. Jedes Werk Pendereckis hat allerdings auch einen Elementnamen `titel`, der den Werknamen darstellt.

Analog wurde der Elementname `datum` sowohl für das Geburtsdatum von Penderecki als auch für das Erscheinungsdatum eines werks verwendet.

XML Namespaces bietet nun die Möglichkeit, Namen in XML-Dokumenten mit einer URI zu verknüpfen, die dann als Name oder ID für einen Namensraum fungiert.

Aus Sicht von XML Namespaces wird ein XML-Namensraum (vergleichbar mit einem virtuellen Bereich) durch ein URI versehen, der ausschließlich der Benennung dient, d.h. nicht auf eine gültige Ressource verweisen muss.

Es ist zu berücksichtigen, dass es keine global verbindliche Definition von dem Inhalt eines Namensraums gibt. Die Namensräume werden daher als offene Räume betrachtet, die nur über ihren Namen mit möglichst logischen URIs¹ versehen sind.

Schließlich gehört jedes Element, das nicht explizit mit einem Namensraum versehen ist, zu dem sogenannten anonymen oder universellen Namensraum.

Innerhalb eines Namensraums kann zwischen folgenden Arten von Elementen und Attributen unterschieden werden:

globale Elemente:

- Zuordnung zum Namensraum über Markierung mit Namensraumnamen.

globale Attribute:

- Eigenständige Zuordnung zum Namensraum über Markierung mit Namensraumnamen, unabhängig von Element, bei dem das Attribut vorkommt. Die Kombination des Namensraumnamens mit dem Attributnamen lässt das globale Attribut eindeutig identifizieren (z.B. `xml:id`, `xml:lang` und `xml:type`).

lokale Attribute für jedes globale Element im Namensraum

- Attribute des Elements ohne eigene Markierung mit Namensraumnamen. Wenn für ein Attribut kein Namespaces-Kürzel angegeben ist, bezieht es sich immer auf den Element-Typ des Tags (lokales Attribut).

¹Ein Uniform Resource Identifier (URI) (deut. „einheitlicher Bezeichner für Ressourcen“).

5.3 Syntax von XML Namespaces

Wie kann man jetzt rein syntaktisch einen Element- oder Attributnamen mit einem Namensraum bzw. einer URI markieren? Dazu sieht XML Namespaces sogenannte Präfixe (einfache Strings) vor, die als Abkürzungen für URI's definiert werden und den Namen vorangestellt werden. Jedes Präfix gilt innerhalb des Elements, in dem es definiert ist, einschließlich des Elements selbst. Somit gilt

```
<?xml version="1.0" encoding="UTF-8"?>
<werke id="Penderecki" xmlns:pn='http://www.person.com'>
  <pn:titel>Prof. Dr.</pn:titel>
  <pn:name>Penderecki</pn:name>
  <pn:vorname>Krzysztof</pn:vorname>
  <pn:datum>23. November 1933</pn:datum>
  <pn:geburtsort>Debica (Polen)</pn:geburtsort>
  <wk:werkliste xmlns:wk='http://www.werk.com'>
    <wk:werk id="1">
      <wk:titel>The Devils of Loudun</wk:titel>
      <wk:datum>1968</wk:datum>
    </wk:werk>
    <wk:werk id="2">
      <wk:titel>Paradise Lost</wk:titel>
      <wk:datum>1976</wk:datum>
    </wk:werk>
    <wk:werk id="3">
      <wk:titel>Die schwarze Maske </wk:titel>
      <wk:datum>1984</wk:datum>
    </wk:werk>
  </wk:werkliste>
</werke>
```

Wie bereits erwähnt gilt das deklarierte Namespaces-Kürzel für das Element selbst, sowie alle Kind-Knoten, sofern es nicht in einem Nachfahren umdefiniert wird. Somit gilt das Kürzel `pn` in unserem vorigen Beispiel für alle Elemente sowie Attribute des XML-Dokuments "werke.xml". Dagegen gilt das Kürzel `wk` nur innerhalb des Element `werkliste`, sowie seine Kind-Knoten.

Es ist aber auch rein syntaktisch möglich die Präfixdefinition innerhalb der Elementhierarchie zu überschreiben (Für die Dokumentsemantik ist diese Variante nicht empfehlenswert). Umgesetzt auf das obige Beispiel in diesem Kapitel wäre also folgende Änderung möglich:

```
<?xml version="1.0" encoding="UTF-8"?>
<werke id="Penderecki" xmlns:st='http://www.person.com'>
  <st:titel>Prof. Dr.</st:titel>
  <st:name>Penderecki</st:name>
  <st:vorname>Krzysztof</st:vorname>
  <st:datum>23. November 1933</st:datum>
  <st:geburtsort>Debica (Polen)</st:geburtsort>
  <st:werkliste xmlns:st='http://www.werk.com'>
    <st:werk id="1">
      <st:titel>The Devils of Loudun</st:titel>
      <st:datum>1968</st:datum>
    </st:werk>
    <st:werk id="2">
      <st:titel>Paradise Lost</st:titel>
      <st:datum>1976</st:datum>
    </st:werk>
  </st:werkliste>
</werke>
```

```
<st:werk id="3">
  <st:titel>Die schwarze Maske </st:titel>
  <st:datum>1984</st:datum>
</st:werk>
</st:werkliste>
</werke>
```

Wir haben also das Präfix `st`, der als Abkürzung für "`http://www.person.com`" galt, überschrieben. Somit ist das Kürzel `st` ab dem Element `werkliste` mit der folgenden URI "`http://www.werk.com`" versehen.

Falls gewünscht, kann man einen Default-Namespace deklarieren, der für alle Element-Typ-Namen gilt, für die nicht explizit ein Namespace-Kürzel angegeben ist (die also keinen ":" enthalten). Ein Beispiel einer solchen Deklaration sieht folgendermaßen aus:

```
<werke id="Penderecki" xmlns:st='http://www.person.com'
  xmlns='http://www.default-namespaces.com'>
```

In diesem Fall steht das Element `werke` in einem sogenannten Default-Namensraum, also nicht im universellen Namensraum. Wenn ein Element in einem Default-Namensraum steht und ein Attribut ohne Präfix trägt, gibt es ein Problem mit der Namensraumzugehörigkeit des Attributes. Ist es als lokales Attribut des Elements im Default-Namensraum zu verstehen oder als globales Element steht. Der Standard XML Namespaces legt die erste Bedeutung fest; d.h., das Attribut ist lokal. Als logische Schlussfolgerung kann der universelle anonyme Namensraum keine globale Attribute enthalten.

5.4 Konformität zu XML Namespaces

Wann ist ein XML-Dokument konform zu XML Namespaces? Dahinter stecken einige Syntaxvorgaben, wie beispielsweise dass die Element- und Attributnamen höchstens einen ":" enthalten dürfen. Der (optionale) Teil vor dem ":" ist das Präfix. Präfixe müssen in einem übergeordneten oder in demselben Element deklariert sein. Alle anderen Namen (z.B. ID-Werte) dürfen keinen ":" enthalten.

Für XML Namespaces gilt eine Nachrangigkeit gegenüber XML. Mit anderen Worten: Namespaces-konforme Dokumente müssen XML-konform sein.

5.5 Namensräume in der DTD

In einer idealen Welt ist die folgende Separation of Concerns sinnvoll: Das Schema bzw. die Strukturvorgabe legt für Elemente und Attribute fest, zu welchem Namensraum sie gehören sollen; in der Instanz, also das konkrete Dokument, wird ein passendes Präfix definiert, mit dem ein Element bzw. Attribut mit dem vorgeschriebenen Namensraum markiert wird. Wir sprechen hier von freier Wahl des Präfixes auf Instanzebene. Eine DTD² schreibt aber nicht nur vor, in welchem Namensraum ein Element stehen soll, sondern auch, mit welchem Präfix die Instanz diese Zuordnung bewerkstelligen muss. Die freie Wahl des Präfixes auf Instanzebene ist nicht gegeben. Das Konfliktpotential verschiebt sich von Namen auf Präfixe.

Eine Behelfslösung besteht darin, in der DTD die Präfixe indirekt, über Entitäten, festzulegen.

```
<!ENTITY % dc-prefix "dc">
<!ENTITY % dc-colon ":">
<!ENTITY % dc-title "%dc-prefix;%dc-colon;title">
<!ELEMENT %dc-title; (#PCDATA)>
```

²DTD's werden im Teil 2 genauer betrachtet

Dann genügt es, eine Entity umzudefinieren, wenn man ein anderes Präfix einstellen möchte. Sobald die DTD eingelesen ist, steht das Präfix jedoch fest; es kann auf der Instanz-Ebene nicht mehr geändert werden.

Zum Abschluss möchten wir noch auf die praktische Möglichkeit hinweisen, mit Hilfe von Deklarationen für Default-Attribute in DTDs in den Instanzen die Tatsache, dass Namensräume verwendet werden, ganz zu verbergen, wie das folgende Beispiel zeigt:

```
<!ATTLIST math xmlns CDATA #FIXED "...">  
Verwendung <math>...</math>
```

Diese Technik ist für sogenannte Dateninseln anwendbar, also Elemente, deren Unterelemente und Attribute alle aus demselben Vokabular stammen.

Dennoch bleiben XML Namespaces in Verbindung mit DTDs nicht ganz frei von Problemen. Eine zufriedenstellende Lösung im Sinne des Prinzips Separation of Concerns bietet der XML Schema Ansatz.

5.6 Zusammenfassung

XML Namespaces stellen eine XML-basierte Syntax zur Verfügung, um Element- und Attributnamen eines Vokabulars eindeutig zu identifizieren und dabei Namenskonflikte auszuschließen. XML Namespaces ermöglicht damit die Komponierbarkeit von unabhängig voneinander entwickelten XML-Vokabularen, die sonst an potentiellen Namenskonflikte scheitern kann. Bei der Entwicklung von XML-Anwendungen wird deshalb die Festlegung eines Namensraumes dringend empfohlen.

Unter Kompatibilität einer Schemasprache mit XML Namespaces versteht man, dass die Strukturvorgabe festlegt, in welchen Namensräumen Elemente und Attribute liegen sollen, und dass Instanzen diese Vorgabe mit selbst-gewählten Präfixen erfüllen. Daraus kann man schließen, dass XML Namespaces und DTDs nicht kompatibel sind.

Wir werden später sehen, dass XML-Schema mit XML Namespaces kompatibel ist

Literatur

- Kurs XML der TEIA-AG (<http://www.teialehrbuch.de/XML/>).
- XML-Portal von O'Reilly (<http://www.xml.com/>).
- XML-Seite des W3C (<http://www.w3.org/XML/>).
- E.R. Harold, W.S. Means: XML in a Nutshell. O'Reilly 2001.
- T. Bray: XML Namespaces by Example(<http://www.xml.com/lpt/a/569>).

Teil II. Schema-Ansätze für XML

In den vorhergehenden Kapiteln haben wir grundlegende Eigenschaften von XML (XML-Instanzen) kennengelernt. XML ist eine Auszeichnungssprache zur Darstellung hierarchisch strukturierter Datensätze in Form von Textdaten. Die Syntax von XML ist standardisiert durch Regeln, wie die Auszeichnung vorgenommen werden muss, wie beispielsweise die Zeichen zur Markierung von Anfangs- und Schließtags "<, >".

Einigen, wenn nicht sogar den meisten, von Ihnen müsste es bekannt sein, dass die Tags, die Sie verwenden können, um ein HTML-Dokument zu schreiben und zu strukturieren, vorgegeben sind. Als Autor eines HTML-Dokuments können Sie also nur solche Tags verwenden, die im HTML-Standard enthalten sind. Die Struktur eines HTML-Dokuments kann daher nicht von einer festen Strukturvorgabe, der sogenannten HTML-Dokumenttypdefinition (HTML-DTD), abweichen.

XML ist im Gegensatz zu HTML nicht auf vordefinierte Tags angewiesen. In XML können Sie Ihre eigenen Tags erstellen und ihnen dann die Bedeutung zuweisen, die Sie für das Element vorgesehen haben. Des Weiteren können Sie in XML eine eigene Dokumentstruktur und das damit in Beziehung stehende XML-Schema für jede neue Art von XML-Dokumente definieren. Dabei sind XML-Schemas Dokumente, die zum Definieren und Überprüfen des Inhalts und der Struktur von XML-Daten verwendet werden, analog zum Definieren und Überprüfen der Tabellen, Spalten und Datentypen einer Datenbank mit einem Datenbankschema.

Schließlich und um das XML-Vokabular sinnvoll nutzen zu können, muss nur noch überprüft werden, ob ein Dokument seiner Strukturvorgabe entspricht oder nicht.

Kapitel 6. DTD

6.1 Orientierung

In diesem Kapitel beschäftigen wir uns mit Dokumenttypdefinitionen, d. h. den Beschränkungen über Strukturen, die formuliert werden können, und der Syntax, mit deren Hilfe dies geschieht.

Zunächst werden wir die grundlegenden Konzepte von DTDs, die als Bestandteile des XML-Standards gelten, obwohl die DTD-Syntax selber kein XML ist, vorstellen. Anhand eines Beispiels wird die Deklaration einer DTD in einem Dokument veranschaulicht. Den Schwerpunkt dieses Kapitels nimmt die Beschreibung der DTD-Konzepte ein, mit der Elemente, Attribute, Entitäten und Notationskonventionen deklariert werden.

6.2 Konzepte

Mittels DTDs können verschiedene Formen von Anforderungen oder Bedingungen über Dokumente festgelegt werden. Zum einen wird ein Dokument durch die DTD in einer Strukturvorgabe beschränkt, d. h. das Vokabular von Element- und Attributnamen und ihren Verwendungsweisen wird festgelegt sowie die Regeln für den Aufbau der logischen Struktur von XML-Dokumenten mit Elementen und Attributen (Instanzen) werden definiert. Zum anderen können DTDs dazu dienen, physische Einheiten festzulegen, aus denen ein Dokument gebildet wird. Diese Funktionen werden wir nun anhand von Beispieldokumenten vorstellen.

Strukturvorgabe:

Eine DTD kann die logische Struktur eines XML-Dokuments definieren. Das folgende Beispiel verdeutlicht, was wir damit meinen:

```
<gedicht>
  <kopf>
    <titel>Als ich die Universität bezog</titel>
    <autor>Rainer Maria Rilke</autor>
    <jahr>1895</jahr>
  </kopf>
  <strophe xml:id="erste-strophe">
    <zeile>Ich seh zurück, wie Jahr um Jahr</zeile>
    <zeile>so müheschwer vorüberrollte;</zeile>
    <zeile>nun endlich bin ich, was ich wollte</zeile>
    <zeile>und</zeile>
  </strophe>
  <strophe>
    ...
  </strophe>
</gedicht>
```

Als erstes können wir einige Regeln aufstellen, die uns bei dem Aufbau einer möglichen DTD hilfreich sein können. Jedes Gedicht hat einen Titel, ein Erscheinungsjahr und einen oder mehreren Autor(en). Zudem besteht es aus mindestens einer Strophe. Die Strophe steht im Dokument direkt hinter dem Titel.

Der Strophe untergeordnet sind die einzelnen Zeilen, von denen, analog zu den Strophen, mindestens eine vorhanden sein muss und beliebig viele stehen können.

Dieses Beispiel stellt drei wichtige Konzepte der Strukturvorgabe vor:

1. Die Beschreibung von Reihenfolge (In unserem Beispiel kommt das Element `strophe` nach dem Element `titel`).
2. Die Angabe der Häufigkeit von den Informationseinheiten (In unserem Beispiel müssen die beiden Elemente `zeile` sowie `strophe` mindestens einmal vorhanden sein, können aber beliebig wiederholt werden).
3. Die Beschreibung von den hierarchischen Beziehungen zwischen den Informationseinheiten (In unserem Beispiel sind die `zeilen` den `strophen` untergeordnet).

Neben Elemente spielen auch die Attribute eine wichtige Rolle. Sie dienen dazu, zusätzliche Informationen über die Elemente darzustellen. In unserem vorigen Beispiel haben wir bereits einen Attribut namens `xml:id` mit dem Attributwert `erste-strophe` kennengelernt.

Physische Struktur:

Während Elemente und Attribute zur logischen Strukturierung von Dokumenten verwendet werden, können in XML-DTDs mittels Entitäten Dokumente physisch segmentiert werden. Somit werden die Komponenten eines XML-Dokuments, aus denen es physisch zusammengesetzt ist, festgelegt.

6.2.1 DTD-Deklaration

Als erstes schauen wir uns die Deklaration einer DTD an. Eine DTD wird am Anfang eines XML-Dokuments in der sogenannten „Document Type Declaration“ untergebracht. Sie kann außerhalb des Dokuments liegen. Alternativ oder zusätzlich dazu können weitere DTD-Deklarationen im XML-Dokument stehen.

Beispiel für Dokumenttyp-Deklaration mit interner DTD:

- ```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Gedicht [<!--Deklaration der internen DTD -->]>
<Gedicht>
</Gedicht>
```

#### **Beispiele für Dokumenttyp-Deklaration mit externer DTD:**

- ```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Gedicht System "../src/gedicht.dtd">
<Gedicht>
</Gedicht>
```
- ```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Gedicht System "http://ep.in.tum.de/xmlvorlagen/gedicht.dtd">
<Gedicht>
</Gedicht>
```
- ```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Gedicht Public "-//Lehrstuhl Angewandte Informatik//DTD Gedicht V 1.0//DE"
System "http://ep.in.tum.de/xmlvorlagen/gedicht.dtd">
<Gedicht>
</Gedicht>
```

Beispiel für Dokumenttyp-Deklaration mit interner und externer DTD:

- ```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Gedicht System "http://ep.in.tum.de/xmlvorlagen/gedicht.dtd"
[<!--Deklaration der internen DTD -->]>
<Gedicht>
</Gedicht>
```

Wie den Beispielen zu entnehmen, beginnt die Dokumenttyp-Deklaration mit `<!DOCTYPE`. Dahinter folgt der Name des Dokumenttyps—in unseren Beispielen `Gedicht`. Die nächste Angabe lautet `SYSTEM`, dazu kann `PUBLIC` hinzugefügt, wenn sich die DTD-Definition in einer separaten Datei befindet, ansonsten entfällt diese letzte Angabe und wird durch Eckige Klammern `[<!-- Deklaration der internen DTD -->]` ersetzt.

**Hinweis:**

`SYSTEM` wird verwendet, wenn der Speicherort der DTD explizit angegeben wird, sei es auf dem eigenen Rechnersystem oder auf einem entfernten Rechner im Internet. `PUBLIC` kennzeichnet eine globale DTD, welche (ähnlich der HTML-DTD) allgemein gültig ist quasi für jeden nutzbar ist. XML-Parser unterstützen häufig Kataloge, in denen ein global gültiger Name auf einen Dateiname oder eine URL abgebildet wird. Trotzdem muss bei der Verwendung von `PUBLIC` als Fallback der `SYSTEM`-Teil auch immer mit angegeben werden.

## 6.2.2 DTD-Bausteine

Eine DTD enthält Festlegungen für:

- mögliche Elemente (sogenannte "Tags"),
- deren Attribute,
- Notationen,
- Textbausteine (sogenannte Entities).

Wie diese Bausteine für den Aufbau von DTDs verwendet werden können, werden wir anhand eines Beispiels veranschaulichen.

Zunächst schauen wir uns einmal ein Gedicht in Form eines XML-Dokuments an:

```

<gedicht>
 <kopf>
 <titel>Als ich die Universität bezog</titel>
 <autor>Rainer Maria Rilke</autor>
 <jahr>1895</jahr>
 </kopf>
 <strophe xml:id="erste-strophe">
 <zeile>Ich seh zurück, wie Jahr um Jahr</zeile>
 <zeile>so müheschwer vorüberrollte;</zeile>
 <zeile>nun endlich bin ich, was ich wollte</zeile>
 <zeile>und was ich strebte: ein Skolar.</zeile>
 </strophe>
 <strophe>
 <zeile>Erst 'Recht' studieren war mein Plan;</zeile>
 <zeile>doch meine leichte Laune schreckten</zeile>
 <zeile>die strengen, staubigen Pandekten,</zeile>
 <zeile>und also ward der Plan zum Wahn.</zeile>
 </strophe>
 <strophe>
 <zeile>Theologie verbot mein Lieb,</zeile>
 <zeile>konnt mich auf Medizin nicht werfen,</zeile>
 <zeile>so daß für meine schwachen Nerven</zeile>
 <zeile>nichts als - Philosophieren blieb.</zeile>
 </strophe>
 <strophe>
 <zeile>Die Alma mater reicht mir dar</zeile>
 <zeile>der freien Künste Prachtregister,-</zeile>
 <zeile>und bring ichs nie auch zum Magister,</zeile>
 <zeile>bin was ich strebte: ein Skolar.</zeile>
 </strophe>
</gedicht>

```

Die Form einer fertigen DTD für unser Beispieldokument "gedicht.xml" kann zum Beispiel wie folgt repräsentiert werden:

```

<!ELEMENT gedicht (kopf,strophe+)>
<!ELEMENT kopf (titel, autor, jahr)>
<!ELEMENT strophe (zeile+)>
<!ATTLIST gedicht xml:id ID #IMPLIED>
<!ELEMENT titel (#PCDATA)>
<!ELEMENT autor (#PCDATA)>
<!ELEMENT jahr (#PCDATA)>
<!ELEMENT zeile (#PCDATA)>

```

Die Bestandteile im Einzelnen:

### Elementdeklaration

Eine Elementdeklaration, wie es unserem Beispiel zu entnehmen, sieht folgendermaßen aus:

Form: `<!ELEMENT Elementname Inhaltsvorgabe>`  
 DTD-Beispiel: `<!ELEMENT gedicht (kopf,strophe+)>`

Der Inhalt eines Elements kann entweder textuelle Daten enthalten, somit wird nur das Schlüsselwort `#PCDATA` als *Inhaltsvorgabe* verwendet. Oder die *Inhaltsvorgabe* dient dazu, die Struktur derjenigen Elemente festzulegen, die in dem definierten Element vorkommen dürfen. In unserer Beispiel-DTD



darf ein Gedicht aus den beiden Elementen `kopf` und `strophe` bestehen. Um die mögliche Struktur festzulegen können aus Elementnamen mit Hilfe von Operatoren Ausdrücke gebildet werden, die die erlaubte Struktur beschreiben (Siehe Tabelle 6.1, „Verkettungs- und Wiederholungsoperatoren“).

Des Weiteren können Daten und Elemente, die in beliebiger Reihenfolge und beliebig häufig auftreten dürfen, gemeinsam in einer Inhaltsvorgabe vorkommen. In diesem Fall sprechen wir von einem gemischten Inhalt "mixed content".

Allgemein sieht eine Deklaration von gemischtem Inhalt folgendermaßen aus:

```
<!ELEMENT Elementname (#PCDATA | Element 1 | ... | Element n)*>
```

Neben dem Schlüsselwort `#PCDATA` sind `ANY` und `EMPTY` weitere reservierte Namen für den Inhalt eines Elements. `ANY` definiert jeden beliebigen Inhalt, in dem Text oder andere Elemente vorkommen können. `EMPTY` deklariert ein leeres Element.

**Tabelle 6.1. Verkettungs- und Wiederholungsoperatoren**

Operator	Beschreibung
?	einmal oder keinmal
+	einmal oder mehrmal
*	keinmal, einmal oder mehrmal
,	Sequenz (gefolgt von)
	Alternative (oder)

### Attributdeklaration

Eine Attributdeklaration, wie es unserem Beispiel zu entnehmen, sieht folgendermaßen aus:

Form: `<!ATTLIST Elementname Attributname Attributtyp StandardVorgabewert>`  
 DTD-Beispiel: `<!ATTLIST gedicht xml:id ID #IMPLIED>`

Der Attributtyp beschreibt welche Werte das Attribut annehmen kann.

Was die verschiedenen *Attributtypen* zu bieten haben, sehen Sie übersichtlich in Tabelle 6.2, „Attributtypen in DTDs“.

**Tabelle 6.2. Attributtypen in DTDs**

Typ	Beschreibung	Beispiel
CDATA	Zeichenkette	"abc", "123"
Aufzählung	Liste möglicher Namen	("Name1"   "Name2"   "Name3")
ID	eindeutiger Bezeichner für dieses Element	"ABC123", "isbn1234"
IDREF	eindeutiger Bezeichner eines bestimmten Elements, gedacht als Referenz	"ABC123", "isbn1234"
IDREFS	Liste von Bezeichnern	"isbn123 isbn456 isbn 789"
NMTOKEN	gültiger XML-Name	"a-b-c", "01.01.2010"
NMTOKENS	Liste von gültigen XML-Namen	"Wert1 Wert2 Wert3"
ENTITY	Name einer in der DTD definierten Entität, auf die verwiesen wird.	<code>&lt;figure src="logo"/&gt;</code> <code>&lt;!ENTITY logo SYSTEM "tum.gif" NOTATION "gif"&gt;</code>

Typ	Beschreibung	Beispiel
NOTATION	Notationsname	<!NOTATION Datentyp SYSTEM "URL">
xml:	vordefinierter XML-Wert	"xml:lang"

Die letzte Angabe (Der sogenannte *StandardVorgabewert*) innerhalb einer Attributdefinition gibt an, ob das Attribut erforderlich ist oder nicht, und besagt, ob es einen Standardwert für dieses Attribut gibt. Die folgende Tabelle 6.3, „Attributsangabe in DTDs“ verdeutlicht die vier möglichen Werte:

**Tabelle 6.3. Attributsangabe in DTDs**

Wert	Bedeutung
"#REQUIRED"	Element muss Attribut enthalten
"#IMPLIED"	Attribut ist optional
"#FIXED"	Attribut muss den hinter #FIXED angegebenen Standardwert annehmen
Literal	Standardwert des Attributs

### Entity-Deklaration

Eine Entity-Deklaration sieht folgendermaßen aus:

Form: <!ENTITY *Entity-Name* *Entity-Definition*>

Des Weiteren unterscheiden wir zwischen zwei Typen von Entities:

#### 1. Generelle Entities:

Generelle Entities werden in der DTD deklariert und in der Instanz verwendet.

Beispiele für die Deklaration interner genereller Entity:

```
<!ENTITY GL 'Gedichte und Lieder 2010'>
<!ENTITY GL 'Gedichte & Lieder 2010'>
```

Beispiele für die Deklaration externer genereller Entity:

```
<!ENTITY GL SYSTEM 'gl.ent'>
<!ENTITY GL PUBLIC '-//TSV//Veranstaltung GL//DE' 'gl.ent'>
```

Verwendung in Dokumenteninstanz (Text und Attributwerte):

&GL;

#### 2. Parameter-Entities

Parameter-Entities werden in der DTD deklariert und verwendet.

Beispiel für die Deklaration interner Parameter-Entity:

```
<!ENTITY % gedichtstext '(#PCDATA | emph)*'>
```

Beispiele für die Deklaration externer Parameter-Entity:

```
<!ENTITY % gedichtstext SYSTEM 'gedichtstext.dtd'>
<!ENTITY % gedichtstext PUBLIC "-//Springer//Gedichtstext DTD//DE"
'gedichtstext.dtd'>
```

Verwendung in DTD:

```
<!ELEMENT titel %gedichtstext;>
<!ELEMENT Autor %gedichtstext;>
```

## 6.2.3 Regeln beim Entwerfen von DTDs

Im letzten Abschnitt haben wir eine Vielzahl von DTD-Bausteinen kennengelernt. Wir werden jetzt lernen, was man bei der DTD-Erstellung beachten muss.

### Attribute vs. Elemente

```
<gedicht titel="Als ich die Universität bezog"
 strophe-zeile1="Ich seh zurück, wie Jahr um Jahr"
 strophe-zeile2="so müheschwer vorüberrollte;"
 strophe-zeile3="nun endlich bin ich, was ich wollte"
 strophe-zeile4="und was ich strebte: ein Skolar....">
```

Der vorige Beispielscode veranschaulicht eine ungeeignete Möglichkeit für den Aufbau einer DTD. Hier sind alle Informationen über das Gedicht als Attribute dargestellt, somit geht die Struktur des Dokuments verloren. Es ist daher zu empfehlen, dass Sie bei der Entscheidung für Elemente oder Attribute einige Eigenschaften beachten:

- Elemente können hinsichtlich eines Datentyps nicht spezifiziert werden, Attribute schon.
- Attribute können keine geordneten Unterstrukturen enthalten, Elemente können dagegen beispielsweise über Unterelemente verfügen.

### Modularisierung

- Bei einem modularisierten Aufbau werden Gesamtdokumente aus standardisierten Einzelbauteilen zusammengesetzt, wie beispielsweise das Aufteilen eines Buches in Kapiteln.

### Wiederverwendbarkeit

- Sie können sehr viel Zeit sparen, wenn Sie Abkürzungen (Entities) verwenden.

## 6.3 Zusammenfassung

In diesem Kapitel haben wir uns mit folgenden Themen, im Zusammenhang mit Dokumenttyp-Definitionen (DTDs) beschäftigt:

Zunächst haben wir gesehen, dass mittels Element- und Attributdefinitionen die logische Struktur von Dokumenten festgelegt werden kann. Darüber hinaus können Entitäten die physikalische Dokumentstruktur verändern.

Wir haben anhand eines Beispiels die Deklaration einer DTD in einem Dokument veranschaulicht.

Die Beschreibung der DTD-Syntax, mit der Elemente, Attribute, Entitäten und Notationskonventionen deklariert werden, hat den Schwerpunkt dieses Kapitels eingenommen.

Zum Schluss haben wir gelernt, was man bei der DTD-Erstellung beachten muss.

---

# Kapitel 7. XML Schema

## 7.1 Orientierung

*"Das Material in diesem Abschnitt ist zu großen Teilen der Diplomarbeit von Dennis Pagano [P08] entnommen."*

Die wohl bedeutendste Schemasprache für XML ist momentan XML Schema (XML Schema Language), welche seit 2. Mai 2001 den Status einer W3C-Empfehlung hat. Nach der Einführung von XML wurden zunächst nur DTDs als Schemata verwendet, doch stellte sich schon bald heraus, dass die DTD-Spezifikation einige Mängel aufwies, die letztendlich zur Entwicklung von XML Schema geführt haben. Die bemängelten Defizite der DTD-Spezifikation betrafen vor allem die Ausdrucksstärke und das Fehlen eines Typsystems, jedoch auch einfachere Tatsachen, wie die Formulierung von DTDs in einem Nicht-XML-Format.

Die Ausdrucksschwäche von DTDs beruht vor allem darauf, dass in DTDs unterschiedliche Inhaltsmodelle stets auch unterschiedliche Elementnamen erfordern. Die kontextabhängige Unterscheidung verschiedener Elementtypen bei gleichen Elementnamen ist in einer DTD nicht möglich, weshalb sich nur lokale Sprachen mit DTDs beschreiben lassen (d. h. Sprachen, bei denen es eine eindeutige Beziehung zwischen Elementname und Elementtyp gibt).

Das meiste Augenmerk richteten die Entwickler von XML Schema auf das Typsystem. Nicht nur stellt XML Schema die in den gängigen Programmiersprachen verwendeten einfachen Typen zur Verfügung, um beispielsweise Attribute spezifizieren zu können, die nur eine Ganzzahl enthalten dürfen. Vielmehr beruht die gesamte Spezifikation auf einem objektorientierten Ansatz, der Typen als Grundlage verwendet. So lassen sich beliebige neue Typen erstellen, bereits existierende erweitern oder einschränken, und auch Vererbung, Polymorphie und Abstraktion finden sich in der XML Schema-Spezifikation wieder. Viele dieser Neuerungen ergaben sich aus Anforderungen, die bei der Verwendung von XML als Grundlage für den Datenaustausch bei verteilten Anwendungen entdeckt wurden. Betrachtet man heutzutage die enorme Fülle von Web-Services oder Applikationen, die AJAX verwenden, so lässt sich die Wichtigkeit dieser Erweiterung gar nicht hoch genug einschätzen.

Im Gegensatz zu DTDs sind XML Schemata selbst in der XML-Syntax notiert, was eine leichtere Parsebarkeit und Validierbarkeit ermöglicht als es bei DTDs der Fall ist.

## 7.2 Konzepte

Wir wollen nun anhand einiger einfachen Szenarien, bei denen wir uns z. B. mit Gedichten beschäftigen, die grundlegenden Konzepte von XML Schema verdeutlichen, auf die wir uns im weiteren Verlauf dieses Skriptes beziehen. Ausgehend von sehr einfachen XML-Dokumenten, entwickeln wir dazu XML Schemata, welche die von uns gewünschte Semantik gewährleisten, und erweitern diese schrittweise. Aktuelle Textbücher ([VV02], [HM04], [MS06]) bieten eine detaillierte Einführung in XML Schema.

Betrachten wir in Abbildung 7.1, „Ein einfaches XML-Dokument“ zunächst ein sehr einfaches wohlgeformtes XML-Dokument:

### Abbildung 7.1. Ein einfaches XML-Dokument

```
<?xml version="1.0" encoding="UTF-8"?>
<dichtername>Rainer Maria Rilke</dichtername>
```

In diesem XML-Dokument speichern wir einen Dichternamen *Rainer Maria Rilke*. Dabei nehmen wir in diesem konkreten Fall an, dass der Name des Dichters stets durch einen Tag<sup>1</sup> `dichtername` gekennzeichnet ist, und der Inhalt des Elements aus einer einfachen Zeichenkette besteht, die den Name des Dichters repräsentiert. Ein zu diesen Annahmen passendes XML Schema sieht wie folgt aus:

### Abbildung 7.2. Ein XML Schema für einen einfachen Dichternamen

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="dichtername" type="xs:string"/>
</xs:schema>
```

Zunächst stellen wir fest, dass es sich bei dem Dokument in Abbildung 7.2, „Ein XML Schema für einen einfachen Dichternamen“ tatsächlich um ein wohlgeformtes XML-Dokument handelt. Weiter ist der Namespace<sup>2</sup> für XML Schema `"http://www.w3.org/2001/XMLSchema"`<sup>3</sup> erkennbar. Um nun zu kennzeichnen, dass das betrachtete XML-Dokument ein Element unserer einfachen XML-Sprache für Dichternamen sein soll, müssen wir eine Referenz auf obiges XML Schema in der Instanz angeben. Wir erhalten somit folgendes XML-Dokument:

### Abbildung 7.3. Ein einfaches XML-Dokument mit Schema-Referenz

```
<?xml version="1.0" encoding="UTF-8"?>
<dichtername xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation="name1.xsd">
 Rainer Maria Rilke
</dichtername>
```

Mit dem Attribut `xsi:noNamespaceSchemaLocation`<sup>4</sup> referenzieren wir die XML Schema-Datei, welche in unserem Beispiel `name1.xsd` heißt. Erst dadurch ermöglichen wir es einem validierenden Parser<sup>5</sup>, das Schema zur Überprüfung heranzuziehen.

Mit diesem XML-Dokument und dem zugehörigen Schema aus Abbildung 7.2, „Ein XML Schema für einen einfachen Dichternamen“ ist der Grundstein für die Diskussion der wichtigsten Konzepte von XML Schema gelegt. In den nachfolgenden Abschnitten betrachten wir diese sukzessive etwas genauer.

## 7.2.1 Elemente

Das erste Konzept, welches wir betrachten sind Elemente. Ein Element wird in XML Schema mit dem Tag `xs:element` deklariert, welches als zwingendes Attribut den Namen des Elements, welches in gültigen Instanzen vorkommen soll, erhalten muss. In Abbildung 7.2, „Ein XML Schema für einen einfachen Dichternamen“ haben wir bereits ein Element namens `dichtername` deklariert.

Elemente, die als direkte Kinder des Wurzelements `xs:schema` in einem XML Schema-Dokument deklariert werden, sind globale Elemente, was die Instanzen des Schemas betrifft. Das bedeutet, dass auf diese Weise deklarierte Elemente – und nur diese – als Wurzelemente in Instanzen verwendet werden dürfen.

<sup>1</sup>Als Tag bezeichnet man im XML-Kontext die Repräsentation eines Elements auf der Markup-Ebene. Dabei wird ein Element immer durch ein öffnendes und ein schließendes Tag gekennzeichnet. Im Falle von leeren Elementen, können diese beiden Tags zu einem einzigen Tag verschmelzen. Siehe auch ([HM04], [MS06]) für eine grundlegende Einführung.

<sup>2</sup>Namespaces werden in XML verwendet, um die Eindeutigkeit von Element- und Attributnamen zu gewährleisten (Siehe hierzu S.64 ff [V05]).

<sup>3</sup>In diesem Skript wird der Namespace für XML Schema stets an das Präfix „xs“ gebunden, eine in der Praxis weit verbreitete Konvention. Grundsätzlich ließe sich jedoch natürlich jedes Präfix verwenden.

<sup>4</sup>Das Attribut `xsi:noNamespaceSchemaLocation` wird für Elemente der Schema-Instanz verwendet, die selbst zu keinem expliziten Namensraum gehören. Andernfalls gebraucht man entsprechend das Attribut `schemaLocation`. Näheres dazu in ([HM04], [MS06]) Für den Namespace `"http://www.w3.org/2001/XMLSchema-instance"` verwenden wir in diesem Skript das Präfix `xsi`.

<sup>5</sup>Ein Parser für XML-Dokumente überprüft nur auf Wohlgeformtheit und erkennt Sprachelemente wie Elemente oder Attribute. Ein validierender Parser kann zusätzlich ein XML-Dokument gegen das darin spezifizierte Schema validieren, also überprüfen, ob das Dokument den im Schema spezifizierten Einschränkungen gerecht wird.

Je umfangreicher das betreffende Schema ist, desto problematischer werden globale Elementdeklarationen. Auf der einen Seite bringen sie Variabilität in die Instanzen, so dass bei vielen globalen Elementen nicht mehr von vorne herein klar ist, welches Wurzelement in einem Dokument erwartet werden kann. Auf der anderen Seite können bei vielen globalen Elementen schneller Namenskonflikte auftreten.

Wie wir bereits erwähnt haben, unterscheidet XML Schema anders als eine DTD zwischen Elementen mit demselben Namen in unterschiedlichen Kontexten, so dass es allein durch die Veränderung des Kontextes möglich wird, denselben Elementnamen anderweitig wieder zu verwenden. Was jedoch ist ein Kontextwechsel in diesem Zusammenhang? Diese Frage führt uns zum nächsten Aspekt des Element-Konzepts, nämlich zur Eltern-Kind-Beziehung zwischen Elementen:

Um in XML-Dokumenten eine Struktur über den Elementen aufzubauen, bedient man sich der natürlichen Eltern-Kind-Beziehung. Darunter versteht man das Verschachteln von Elementen, so dass das öffnende und das schließende Tag eines Elements andere Elemente vollständig umschließen. Das äußere Element nennt man in diesem Fall in Bezug auf die inneren Elemente das Elternelement, die umschlossenen Elemente sind entsprechend Kindelemente des äußeren Elements.

#### Abbildung 7.4. Eltern-Kind-Beziehung in einem XML-Dokument

```
<?xml version="1.0" encoding="UTF-8"?>
<dichtername>
 <vorname>Rainer Maria</vorname>
 <nachname>Rilke</nachname>
</dichtername>
```

In Abbildung 7.4, „Eltern-Kind-Beziehung in einem XML-Dokument“ ist beispielsweise das Element `dichtername` ein Elternelement für die Elemente `vorname` und `nachname`, welche beide jeweils entsprechend wieder Kindelemente von `dichtername` sind.

Um solche Verschachtelungen in XML Schema umsetzen zu können, benötigen wir allerdings noch ein weiteres Konzept, nämlich Typen, die eigenständig, außerhalb von Element- und Attributdeklaration definiert werden können. Eine solche Verschachtlung ist in XML gang und gäbe. Im Gegensatz zu DTDs bietet XML Schema jedoch die Möglichkeit, die Deklaration von `vorname` und `nachname` lokal innerhalb der Deklaration von `dichtername` vorzunehmen, so dass sie in anderen Kontexten wieder frei zur Verfügung stehen.

## 7.2.2 Typen

Spezifiziert man mit XML Schema Elemente oder Attribute, so erhalten diese stets einen Typ. Dieser kann entweder explizit definiert werden, erhält dann einen Namen und wird als globaler Typ bezeichnet, oder er entsteht implizit bei der Deklaration eines Elements oder Attributs, und man nennt ihn lokalen Typ. Globale Typen sind stets über ihren Namen referenzierbar, wohingegen lokale Typen nur im Kontext ihrer Definition sichtbar sind.

**Tabelle 7.1. Einige in XML Schema vordefinierte Datentypen**

Datentyp	Beschreibung
boolean	Boolescher Wert
date	Datum
decimal	decimal
dateTime	Datum und Uhrzeit
float	Gleitkommazahl
integer	Ganzzahl
string	Zeichenkette
time	Uhrzeit

Das bedeutet, dass wir den Inhalt eines Elements oder Attributs in der Weise weiter einschränken können, wie wir es von der objektorientierten Programmierung her kennen. Dabei unterscheidet XML Schema zwischen einfachen Typen und komplexen Typen. Einem Attribut kann mit XML Schema nur ein einfacher Typ zugewiesen werden, ein Element hingegen kann auch einen komplexen Typen haben.

## 7.2.3 Einfache Typen (Datentypen)

Ein Beispiel für einen einfachen Typ haben wir bereits in unserem ersten XML Schema kennen gelernt, als wir gefordert haben, dass der Name eines Dichters aus einer Zeichenkette bestehen soll:

```
<xs:element name="dichtername" type="xs:string"/>
```

Das Attribut `type` der Elementdeklaration enthält den vordefinierten Datentyp `string`, und legt somit fest, dass das Element `dichtername` in jeder gültigen Instanz dieses Schemas eine Zeichenkette als Inhalt haben muss. In der XML Schema-Spezifikation wurden viele einfache Typen, wie sie in den meisten gängigen Programmiersprachen vorhanden sind, bereits vordefiniert. In Tabelle 7.1, „Einige in XML Schema vordefinierte Datentypen“ sind diejenigen dieser Datentypen aufgeführt, die wir im weiteren Verlauf noch benötigen werden. Wollen wir beispielsweise ein Element deklarieren, welches die Gedichte-anzahl eines Dichters bezeichnen soll, so könnten wir dies unter Zuhilfenahme des Datentyps `integer` in folgender Weise ganz einfach fordern:

```
<xs:element name="gedichte-anzahl" type="xs:integer"/>
```

In Bezug auf dieses Schema gültige XML-Dokumente dürften dann nur ganze Zahlen als Inhalt für das Element `gedichte-anzahl` verwenden.<sup>6</sup>

Ein Problem mit unserer Deklaration ist der auf der semantischen Ebene zu großzügig gewählte gültige Bereich für die `gedichte-anzahl`. Sinnvoller als beliebige positive und negative Ganzzahlen zuzulassen wäre es, die Gedichte-Anzahl z. B. auf einen Bereich zwischen 1 und 1000 zu beschränken.<sup>7</sup> XML Schema bietet dafür die Möglichkeit an, neue einfache Typen auf der Basis von bereits existierenden zu definieren, wie wir im folgenden Beispiel sehen:

### Abbildung 7.5. Ein einfacher benutzerdefinierter Datentyp in XML Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="gedichte-anzahl" type="gedichte-anzahlType"/>
 <xs:simpleType name="gedichte-anzahlType">
 <xs:restriction base="xs:integer">
 <xs:minInclusive value="1"/>
 <xs:maxInclusive value="1000"/>
 </xs:restriction>
 </xs:simpleType>
</xs:schema>
```

Zunächst erkennen wir die Referenzierung des von uns definierten Typen `gedichte-anzahlType` im `type`-Attribut des `element`-Tags. Die Definition des Datentypen wird durch das Element `simpleType` bewerkstelligt, das über das Attribut `name` den Namen des definierten Typen festlegt. Das Kindelement `restriction` ist eine Methode in XML Schema, aus bereits existierenden Datentypen einen neuen Datentypen abzuleiten.<sup>8</sup> Es bedeutet, dass der zugrundeliegende Datentyp, der über das Attribut `base` spezifiziert wird, hinsichtlich eines bestimmten Aspekts eingeschränkt wird. In XML Schema werden diese Aspekte Facetten genannt. Wir verwenden die beiden

<sup>6</sup>Betrachtet man ein XML-Dokument als solches, so ist offenkundig, dass zunächst jedes Element oder Attribut Zeichenketten enthält. Erst die Interpretation dieser Zeichenketten im Kontext eines Datentypen macht aus der Zeichenkette einen anderen Datentyp, beispielsweise eine Ganzzahl. In diesem Zusammenhang spricht man ebenfalls vom Parsen der Zeichenkette und von der Validierung des Inhalts in Bezug auf den erwarteten Datentyp. Bei den einfachen Typen in XML Schema wird hier hauptsächlich von Regulären Ausdrücken Gebrauch gemacht, um diese Validierung durchzuführen.

<sup>7</sup>Die Frage nach einer sinnvollen Begrenzung beruht auf der jeweiligen Anwendungsdomäne.

<sup>8</sup>Weitere Methoden sind `list` und `union`.



Facetten `minInclusive` und `maxInclusive`, um die Menge der gültigen Ganzzahlen auf das geschlossene Intervall zwischen 1 und 1000 zu beschränken.

Je nach erweitertem Datentyp können wir mit den verschiedenen Facetten auf diese Weise z. B. reguläre Ausdrücke verwenden, um nur ganz bestimmte Zeichenketten zuzulassen, oder die Anzahl der Nachkommastellen einer Gleitkommazahl beschränken.

## 7.2.4 Komplexe Typen (Strukturtypen)

Einfache Typen sind grundsätzlich so geartet, dass sie von Attributen aufgenommen werden können, die – wie bereits erwähnt – nur einfache Typen haben dürfen. Insbesondere können Datentypen keine Elemente enthalten oder andere XML-spezifische Strukturen abbilden. Ein Typ für den Dichternamen aus Abbildung 7.4, „Eltern-Kind-Beziehung in einem XML-Dokument“ lässt sich somit nicht als einfacher Typ realisieren, XML Schema stellt dafür sogenannte komplexe Typen (auch Strukturtypen genannt) zur Verfügung. Elemente, die einen komplexen Typen haben, können weitere, möglicherweise verschachtelte Elemente enthalten oder über Attribute verfügen.

Analog zur Definition eines einfachen Typen, wird ein komplexer Typ über das Element `complexType` definiert, wie wir im folgenden Beispiel sehen, das den bereits bekannten Dichternamen abbildet:

**Abbildung 7.6. Ein komplexer Typ in XML Schema**

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="dichtername" type="dichternameType"/>
 <xs:complexType name="dichternameType">
 <xs:sequence>
 <xs:element name="vorname" type="xs:string"/>
 <xs:element name="nachname" type="xs:string"/>
 </xs:sequence>
 </xs:complexType>
</xs:schema>
```

**Folgen von Elementen** Das einzig neue Element in Abbildung 7.6, „Ein komplexer Typ in XML Schema“ ist das Element `sequence`, welches verwendet wird, um eine Sequenz von Unterelementen zu spezifizieren, die Unterelementdeklarationen selbst sind uns bereits bekannt. Allerdings ist an dieser Stelle noch anzumerken, dass diese Unterelemente nun lokal deklariert sind, d. h. sie sind außerhalb von `dichtername` nicht bekannt. Sequenz bedeutet hierbei, dass alle aufgeführten Elemente in dieser Reihenfolge in einer gültigen Instanz auftreten müssen.

Darüber hinaus stellt XML Schema mit den Schlüsselwörtern `all` und `choice` zwei weitere Mechanismen für die Definition von Folgen von Elementen für das Inhaltsmodell von komplexen Typen bereit, wobei diese Inhaltsmodelle wieder verschachtelbar sind. Wird `all` verwendet, so müssen in einem gültigen XML-Dokument alle innerhalb von `all` definierten Elemente an dieser Stelle auftreten, die Reihenfolge ist jedoch unerheblich. Benutzt man hingegen `choice`, so kann spezifiziert werden, welche der Unterelemente in einer Instanz auftreten können (siehe auch nächster Abschnitt), welche Elemente dies sind, ist dann nicht von Belang.

**Häufigkeitsaussagen** Die Möglichkeit festlegen zu können, wie viele Elemente in einer Instanz auftreten müssen, ist ein allgemein verwendbares Konzept in XML Schema. Dabei bedient man sich der Schlüsselattribute `minOccurs` und `maxOccurs`, um die untere bzw. die obere Grenze zu setzen. Wird eine Grenze nicht explizit spezifiziert, so wird sie stets als 1 angenommen. XML Schema bietet außerdem die Möglichkeit, für die obere Schranke den Wert `unbounded` zu verwenden, was gleichbedeutend damit ist, dass keine obere Schranke für das Auftreten des betreffenden Elements an dieser Stelle existiert.

Betrachten wir zunächst unser Dichternamen-Beispiel. Wollten wir einen optionalen Zwischennamen in unser Modell einfügen, so lässt sich dies durch das Setzen der unteren Grenze auf 0 und der oberen Grenze auf 1 (oder Weglassen des Attributs `maxOccurs`) bewerkstelligen:



### Abbildung 7.7. Eine Häufigkeitsaussage auf Elementebene in XML Schema

```
<xs:complexType name="dichternameType">
 <xs:sequence>
 <xs:element name="vorname" type="xs:string"/>
 <xs:element name="zweitername" type="xs:string" minOccurs="0"/>
 <xs:element name="nachname" type="xs:string"/>
 </xs:sequence>
</xs:complexType>
```

Ein weiteres Beispiel behandelt den im vorherigen Abschnitt eingeführten choice-Mechanismus im Zusammenhang mit der Begrenzung der Anzahl der auszuwählenden Elemente.

Wir legen die Vorgabe einer Mutter an ihr Kind, sich höchstens 3 Kugeln Eis aussuchen zu dürfen, in ein XML Schema um, dessen Instanzen dann in diesem Sinne zulässige Zusammenstellungen darstellen. Dabei sind drei Kugeln Vanille genauso zulässig wie eine Fürst-Pückler-Variante<sup>9</sup>:

### Abbildung 7.8. Ein Strukturtyp mit Auswahl in XML Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="iceCream" type="iceCreamType"/>
 <xs:complexType name="iceCreamType">
 <xs:choice maxOccurs="3">
 <xs:element name="chocolate"/>
 <xs:element name="strawberry"/>
 <xs:element name="vanilla"/>
 <xs:element name="pistachio"/>
 </xs:choice>
 </xs:complexType>
</xs:schema>
```

Grundsätzlich lassen sich `minOccurs` und `maxOccurs` immer beim Deklarieren von Elementen verwenden, die in einer Folge vorkommen, also unterhalb von `sequence`, `all`, `any`, `local groups`, `local elements` oder `choice` (vgl. Abbildung 7.7, „Eine Häufigkeitsaussage auf Elementebene in XML Schema“), jedoch auch die Häufigkeit der Folgen selbst kann mit diesen Attributen spezifiziert werden (vgl. Abbildung 7.8, „Ein Strukturtyp mit Auswahl in XML Schema“). In `global elements` bzw. `global groups` dürfen `minOccurs` und `maxOccurs` nicht vorkommen. Zu beachten ist außerdem der Sonderfall `all`, bei dem `minOccurs` und `maxOccurs` nur den Wert "1" annehmen dürfen.

## 7.2.5 Attribute

Elemente in XML-Dokumenten können bekanntermaßen (nahezu) beliebige Attribute haben. Dieses Sprachelement lässt sich natürlich ebenfalls mit XML Schema abbilden, es gibt also die Möglichkeit, sowohl den Namen als auch den Datentyp eines Attributs festzulegen. Dabei können Attribute analog zu Elementen wieder global oder lokal deklariert werden. Die globale Deklaration von Attributen hat natürlich nur den Zweck, die entsprechenden Attribute für Typdefinitionen referenzierbar zu machen.

Betrachten wir als Beispiel die Spezifizierung eines Dichters, welcher einen `vorname`, einen `nachname`, einen optionalen `zweitername` sowie ein `geburtsdatum` hat:

<sup>9</sup>Erdbeere, Vanille und Schokolade

### Abbildung 7.9. Attributdeklarationen in XML Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="dichter">
 <xs:complexType>
 <xs:attribute name="vorname" type="xs:string"/>
 <xs:attribute name="zweitername" type="xs:string"
 use="optional"/>
 <xs:attribute name="nachname" type="xs:string"/>
 <xs:attribute ref="geburtsdatum"/>
 </xs:complexType>
 </xs:element>
 <xs:attribute name="geburtsdatum" type="xs:date"/>
</xs:schema>
```

Zunächst halten wir fest, dass es sich beim Element `dichter` um ein Element mit einem lokal definierten, komplexen Typen handelt. Wie wir bereits wissen, ist ein komplexer Typ in XML Schema die Voraussetzung an ein Element, um Attribute aufnehmen zu können. Die Deklaration eines Attributs verläuft dann analog zu einer Elementdeklaration über das Element `attribute`, welches wieder den Namen des deklarierten Attributs übernimmt. Durch die Verwendung von `use="optional"` markieren wir das Attribut `zweitername` als nicht notwendigerweise anzugeben.

Das letzte Attribut von `dichter` haben wir im Gegensatz zu den anderen global deklariert und referenzieren es nun am gewünschten Ort über seinen Namen und das Attribut `ref`. Auf genau diese Weise funktioniert im Übrigen auch die Referenzierung von global deklarierten Elementen. Hätten wir für dieses Schema einen Namensraum verwendet, so wäre das Attribut `dichter` nur über dessen Präfix erreichbar. Wenn dies nicht erwünscht ist, das Attribut aber dennoch global sein soll, so empfiehlt sich das Verwenden einer Attributgruppe (siehe z. B. [VV02]).

## 7.2.6 Vererbung

Bei den einfachen Typen („7.2.3 Einfache Typen (Datentypen)“) haben wir mit der Einschränkung vordefinierter einfacher Typen (`restriction`) eine Funktionalität von XML Schema verwendet, welche dem Konzept der Vererbung ähnelt, das wir aus der objektorientierten Programmierung kennen. Bei komplexen Typen besteht die Möglichkeit, benutzerdefinierte Typen einzuschränken oder zu erweitern. Dieses Ableiten neuer komplexer Typen von anderen bezeichnet man in XML Schema als Vererbung.

**Vererbung durch Erweiterung** Unter einer Erweiterung eines Typen versteht man das Hinzufügen weiterer Merkmale wie beispielsweise zusätzlicher Attribute oder Kindelemente. Dabei ergibt sich als abgeleiteter Typ derselbe wie er durch das direkte Anfügen dieser Merkmale an den Basistypen entstehen würde. Um einen Typen zu erweitern, verwendet man das Element `extension`, wie wir im folgenden Beispiel sehen, das das bereits bekannte Beispiel `dichtername`, der die beiden Elemente `vorname` und `nachname` enthält, um einen `titel` erweitert:

### Abbildung 7.10. Vererbung durch Erweiterung in XML Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="dichtername" type="dichternameType"/>
 <xs:complexType name="dichternameType">
 <xs:sequence>
 <xs:element name="vorname" type="xs:string"/>
 <xs:element name="zweitername" type="xs:string" minOccurs="0"/>
 <xs:element name="nachname" type="xs:string"/>
 </xs:sequence>
 </xs:complexType>
 <xs:complexType name="extendedNameType">
 <xs:complexContent>
 <xs:extension base="dichternameType">
 <xs:sequence>
 <xs:element name="titel"/>
 </xs:sequence>
 </xs:extension>
 </xs:complexContent>
 </xs:complexType>
</xs:schema>
```

Die Definition des abgeleiteten Typen `extendedNameType` hängt das Element `titel` an den existierenden Typen `dichternameType` an. Ein sehr nützlicher Aspekt bei der Vererbung durch Erweiterung zeigt sich hinsichtlich der Toleranz gegenüber Veränderungen: Wird der Basistyp modifiziert, so erbt ein durch Erweiterung entstandener Untertyp automatisch alle Veränderungen. Dies ist insbesondere bei umfangreicheren Software-Projekten wichtig, bei denen Basis- und Untertyp von verschiedenen `dichter` entwickelt werden.

**Vererbung durch Restriktion** Will man einen Typen erzeugen, der eine Teilmenge der Merkmale eines anderen Typen besitzt, so kann man dies in XML Schema mit Hilfe des Schlüsselwortes `restriction` ausdrücken. Bei den einfachen Typen haben wir diesen Mechanismus bereits verwendet, bei komplexen Typen jedoch gestaltet sich die Restriktion etwas schwieriger.

Um von einem Strukturtypen durch Einschränkung zu erben, muss man den vollständigen Basistypen reproduzieren und dabei die gewünschten Merkmale restringieren:

### Abbildung 7.11. Vererbung durch Restriktion in XML Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="dichtername" type="basicdichternameType"/>
 <xs:complexType name="basicdichternameType">
 <xs:sequence>
 <xs:element name="vorname" minOccurs="0"
 maxOccurs="unbounded"/>
 <xs:element name="nachname"/>
 <xs:element name="titel" minOccurs="0"/>
 </xs:sequence>
 </xs:complexType>
 <xs:complexType name="simplifiedichternameType">
 <xs:complexContent>
 <xs:restriction base="basicdichternameType">
 <xs:sequence>
 <xs:element name="vorname"/>
 <xs:element name="nachname"/>
 </xs:sequence>
 </xs:restriction>
 </xs:complexContent>
 </xs:complexType>
</xs:schema>
```

Die in Abbildung 7.11, „Vererbung durch Restriktion in XML Schema“ vorgenommenen Einschränkungen sind das Setzen von `minOccurs` und `maxOccurs` auf "1" beim Element `vorname` sowie das Setzen von `maxOccurs` auf "0" beim Element `titel`. Man beachte, dass das

bloße Weglassen von Elementen nur erlaubt ist, wenn der Basistyp dessen minimale Häufigkeit mit "0" beziffert.

Wie wir gesehen haben, muss bei der Vererbung durch Restriktion der Basistyp teilweise reproduziert werden. Dies führt jedoch zu Redundanzen im Schema, die nicht automatisch aufgelöst werden können. So bleibt ein Typ, dessen Basistyp verändert wurde, von diesen Veränderungen unberührt, und das Schema wird dadurch inkonsistent. Deshalb ist die Vererbung durch Restriktion in diesem Zusammenhang nicht besonders empfehlenswert.

**Instanziierung** Wie in der objektorientierten Programmierung können abgeleitete Typen in Instanzen anstelle des Basistypen verwendet werden. Um den verwendeten Typen in der Instanz zu identifizieren, verwendet man am betreffenden Element das `xsi:type`-Attribut, welches als Wert den Namen des Typen erhält.

Betrachten wir z. B. eine Instanz zum XML Schema in Abbildung 7.11, „Vererbung durch Restriktion in XML Schema“, in der wir den Typen `simplifiedichtername` für das Wurzelement verwenden wollen:

### Abbildung 7.12. Typenkennzeichnung bei der Instanziierung in XML Schema

```
<? xml version="1.0" encoding="UTF-8"?>
<dichtername xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation="name1.xsd"
 xsi:type="simplifiedichternameType">
 <vorname>Rainer Maria</vorname>
 <nachname>Rilke</nachname>
</dichtername>
```

Da im Schema das Wurzelement `name` den Typen `basicdichtername` hat, geben wir den gewünschten abgeleiteten Typen `simplifiedichtername` im Attribut `xsi:type` an.

## 7.2.7 Substitutionsgruppen

In einer Substitutionsgruppe kann man Elemente angeben, die alle statt eines bestimmten Elements, des sogenannten Head-Elements in Instanzen verwendet werden können. Damit ist dieses Konzept dem Vererbungsmechanismus relativ ähnlich, allerdings adressiert es den Elementnamen, nicht den Typen. Trotzdem gibt es die Vorgabe, dass Elemente in der Substitutionsgruppe eines Head-Elements entweder denselben Typ wie das Head-Element oder einen davon abgeleiteten Typen haben müssen, wodurch eine im Sinne des Konzepts der Vererbung sinnvolle Verwendung der Substitutionsgruppen erzwungen wird.

Um eine Substitutionsgruppe zu erstellen, muss bei einer Elementdeklaration über das Attribut `substitutionGroup` das Head-Element für die Gruppe angegeben werden, zu der das deklarierte Element gehören soll. Alle Elemente einer Substitutionsgruppe inklusive des Head-Elements müssen global deklariert sein.

Betrachten wir ein Beispiel, in dem wir ein Schema für Haustiere erstellen. Wir definieren sowohl einen allgemeinen Haustier-Typen, als auch eine spezielle Variante, den Papageien-Typ. Für jeden Typ deklarieren wir ein globales Element und setzen das Papageien-Element in die Substitutionsgruppe des Haustier-Elements:



### Abbildung 7.13. Substitutionsgruppen in XML Schema

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="myPets">
 <xs:complexType>
 <xs:sequence maxOccurs="unbounded">
 <xs:element ref="pet" />
 </xs:sequence>
 </xs:complexType>
 </xs:element>
 <xs:element name="pet" type="basicPetType" />
 <xs:complexType name="basicPetType">
 <xs:sequence>
 <xs:element name="name" />
 <xs:element name="favouriteFood" />
 </xs:sequence>
 </xs:complexType>
 <xs:element name="parrot" type="parrotType" substitutionGroup="pet" />
 <xs:complexType name="parrotType">
 <xs:complexContent>
 <xs:extension base="basicPetType">
 <xs:sequence>
 <xs:element name="coatColor" />
 </xs:sequence>
 </xs:extension>
 </xs:complexContent>
 </xs:complexType>
</xs:schema>
```

Als Wurzelement erwarten wir das Element `myPets`, welches eine Auflistung aller unserer Haustiere enthalten soll. Wie wir sehen, sind damit sowohl der Typ `parrotType`, als auch das Element `parrot` selbst abgeleitet von einem Element und dessen Typ. Diese Form der Erweiterung wird nach [C02] *double extension* genannt.

Ein Vorteil der Substitutionsgruppen ist es, dass der in der Instanz verwendete Typ nicht mehr explizit mittels `xsi:type` spezifiziert werden muss, sondern vom XML Schema- Prozessor über den Elementnamen inferiert wird. So können in der Instanz die Elemente `pet` und `parrot` wahlweise beide verwendet werden:

### Abbildung 7.14. Elemente einer Substitutionsgruppe in einer Instanz

```
<?xml version="1.0" encoding="iso-8859-1"?>
<myPets xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation="pets.xsd">
 <pet>
 <name>Nemo</name>
 <favouriteFood>Water Fleas</favouriteFood>
 </pet>
 <parrot>
 <name>Polly</name>
 <favouriteFood>Cracker</favouriteFood>
 <coatColor>Red</coatColor>
 </parrot>
</myPets>
```

## 7.2.8 Abstraktion

Von der objektorientierten Programmierung her ist das Konzept der abstrakten Typen bereits bekannt. Dabei handelt es sich um Typen, die vor allem strukturelle Eigenschaften besitzen, weniger konkrete Implementierungen, welche aber vor allem nicht direkt instanziiert werden können.

In XML Schema gibt es sogar zwei Möglichkeiten, Abstraktion zu verwenden, nämlich zum einen auf Elementebene und zum anderen auf Typebene. In beiden Fällen verwendet man das boolesche Attribut `abstract`, um das betreffende Sprachelement als abstrakt zu deklarieren. Ein abstraktes

Element darf in keinem Instanzdokument verwendet werden; stattdessen kann jedoch ein Element aus einer Substitutionsgruppe benutzt werden, dessen Head-Element das abstrakte Element ist. Ebenso darf ein Element, welches einen als abstrakt markierten Typ besitzt, nicht in einer Instanz auftreten.

**Tabelle 7.2. Verschiedene Verwendungsweisen von XML Schemata und zugehörige Rollen**

Rolle	Verwendung von XML Schemata
Schema-Implementierer	Implementierung eines Schemas
Schema-Benutzer	Verwenden modellierter Konzepte
Schema-Erweiterer	Erweiterung eines existierenden Schemas

## 7.2.9 Verwendung und Rollen

Im Software-Engineering gibt es naturgemäß verschiedene Sichtweisen auf Software-Entwickler, die von deren Aufgaben in Bezug auf die Herangehensweise an die Implementierung von bestimmten Komponenten abhängen. Nach [M02] sind diese verschiedenen Typen von Entwicklern im Einzelnen Klassenimplementierer, Klassenbenutzer und Klassenerweiterer.

Da das Erstellen von XML Schemata im Grunde ein Spezialfall der Implementierung von Modellen im Sinne des Software-Engineering ist, lassen sich analog zu diesen Typen je nachdem, wie ein XML Schema bei der Software-Entwicklung zum Einsatz kommt, verschiedene Rollen identifizieren, die die Art der Verwendung des Schemas charakterisieren:

- Schema-Implementierer realisieren XML Schemata, indem sie die einzelnen Typen und Elemente nach bestimmten Vorgaben direkt in XML Schema umsetzen.
- Schema-Benutzer verwenden ein existierendes XML Schema, um darin modellierte Konzepte anwenden zu können.
- Schema-Erweiterer entwickeln XML Schemata, die bestimmte Typen oder Elemente aus anderen XML Schemata erweitern oder verfeinern.

Betrachtet man die einzelnen Rollen, so wird klar, dass es für jede Rolle verschiedene Anforderungen an das entsprechende Schema gibt, die zur Vereinfachung der Entwicklung oder des Gebrauchs beitragen.

Die Implementierung eines Schemas, also die konkrete Umsetzung eines Modells, ist der Gegenstand der Untersuchungen in diesem Teil der Vorlesung. Dabei sind neben der korrekten Abbildung des Modells gleichbleibende Herangehensweisen beim Übersetzen und die Verwendung von bewährten Methoden und Mustern besonders wichtig, um das resultierende Schema möglichst leicht erweiterbar zu halten. Wie bei der Implementierung von Klassen objektorientierter Programmiersprachen ist die automatische Umsetzung dieser Konzepte besonders anstrengenswert, um möglichst wenig Fehlerquellen zu schaffen. Jedoch ist im Gegensatz zur Umsetzung in Klassen der Anspruch insofern geringer, als dass ein XML Schema keine Operationen der modellierten Entitäten darstellen kann, und deshalb eine automatische Übersetzung, möglicherweise sogar ein round-trip-engineering-Ansatz realistischer ist. Andere Anforderungen an das zu erstellende Schema lassen sich grundsätzlich als Parameter für den Übersetzer interpretieren.

Beim Verwenden modellierter Konzepte ist es wichtig, dass diese immer auf dieselbe Art und Weise umgesetzt wurden, denn nur so lässt sich eine intuitive Arbeitsweise mit diesem Schema erwirken. Solange das verwendete Schema keine Drittanbieter-Komponente ist, kann dies durch geeignete Vorgaben für die Umsetzung erreicht werden.

Schema-Erweiterer müssen sich – ebenso wie Schema-Benutzer – auf die Korrektheit des Schemas verlassen. Für ihre Tätigkeit ist jedoch ein möglichst "gutes" Design des Schemas wichtig.

Viele Aspekte dieser Güte resultieren bereits aus vernünftigen Entwurfsentscheidungen während der vorhergehenden Phasen der Modellierung. Jedoch gibt es auch XML Schema-spezifische Entscheidungen, die in diesem Fall zu einer Erleichterung etwaiger Erweiterungen führen. Diese sind in etwa vergleichbar mit der Verwendung von Design-Patterns [BKST07] in der objektorientierten Software-Entwicklung, um die zu entwickelnde Software robust gegen Veränderungen zu machen, und lassen sich als Patterns für XML Schema zusammenfassen.

## 7.3 Patterns für XML Schema

Im objektorientierten Software-Engineering bedient man sich bei der Entwicklung neuer Software oft sogenannter Design Patterns [BKST07] oder Entwurfsmuster, also „Lösungsvorlagen, die Entwickler mit der Zeit verfeinert haben, um eine Reihe wiederkehrender Probleme zu lösen“ ([M02], S. 348) und spezielle Klassen vor zukünftigen Änderungen zu schützen.

Auch bei der Entwicklung von XML Schemata kann man auf Design Patterns zurückgreifen, die sich darin unterscheiden, ob Elemente und Datentypen global oder lokal deklariert und definiert werden sollen. Diese unterschiedlichen Strategien begünstigen verschiedene weitere Arbeitsschritte mit den zu erstellenden Schemata.

Um die verschiedenen Design Patterns für XML Schema<sup>10</sup> zu erläutern, bedienen wir uns des folgenden Ausschnitts eines XML-Dokuments, der ein *gedicht* beinhaltet:

```
<gedicht>
 <autor>
 <vorname>Rainer Maria</vorname>
 <nachname>Rilke</nachname>
 </autor>
 <titel>Als ich die Universität bezog</titel>
</gedicht>
```

### 7.3.1 Russian Doll

Beim Russian Doll Design Pattern<sup>11</sup> werden lokale Elementdeklarationen in anonymen Typdefinitionen immer tiefer verschachtelt. Man erhält somit ausschließlich lokale Datentypen und mit Ausnahme des Wurzelements auch nur lokale Elemente:

**Abbildung 7.15. Das Russian Doll XML Schema Design Pattern**

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="gedicht"/>
 <xs:complexType>
 <xs:sequence>
 <xs:element name="autor">
 <xs:complexType>
 <xs:sequence>
 <xs:element name="vorname" type="xs:string"/>
 <xs:element name="nachname" type="xs:string"/>
 </xs:sequence>
 </xs:complexType>
 </xs:element>
 <xs:element name="titel" type="xs:string"/>
 </xs:sequence>
 </xs:complexType>
</xs:schema>
```

Das Russian Doll Design Pattern hat unter anderem folgende Auswirkungen auf die Struktur, Erweiterbarkeit und Wiederverwendbarkeit (nach [GHJV95]):

<sup>10</sup>Siehe auch [BD04] und [GHJV95].

<sup>11</sup>Der Name dieses Design Patterns rührt von der Matryoshka her, einem aus Holz gefertigten russischen Spielzeug, welches aus mehreren ineinander schachtelbaren Puppen besteht.

- **Verborgener Inhalt.** Der Inhalt des `gedicht`-Elements ist weder für andere Schemata, noch für andere Bereiche desselben Schemas sichtbar. Dies bedeutet, dass keiner der darin definierten Datentypen wiederverwendbar ist.
- **Lokaler Geltungsbereich.** Teilelemente einer Komponente sind nur lokal sichtbar. Dadurch wird die Komplexität verringert.
- **Kompaktheit.** Alle zugehörigen Elemente und Datentypen sind in einer einzigen Einheit gekapselt.
- **Entkopplung.** Jede Komponente ist vollständig gekapselt, es existiert also keine Beziehung zu anderen Komponenten. Dies bedeutet, dass Veränderungen an einer Komponente nur begrenzten Einfluss auf andere Komponenten haben.
- **Kohäsion.** Zusammengehörende Konzepte und Daten sind in sich geschlossen und gekapselt.

Das Russian Doll Design Pattern zielt somit vor allem auf die Verminderung von Komplexität ab. Wiederverwendbarkeit ist damit nur begrenzt erreichbar.

Eine drastische Einschränkung, die das Russian Doll Design Pattern mit sich bringt, ist die Beschränkung der Schachtelungstiefe von Instanzen. Da jedes Schema endlich ist und nur lokale Elementdeklarationen und Typdefinitionen verwendet werden, ist es damit unmöglich, rekursive Strukturen abzubilden. Rekursive Strukturen kommen jedoch in der Praxis relativ häufig vor, man denke beispielsweise an das Composite-Pattern [BKST07] und seine praktischen Anwendungen.

## 7.3.2 Salami Slice

Beim Salami Slice Design Pattern werden alle Elemente global deklariert, jedoch definiert man alle Datentypen wieder lokal und somit anonym:

**Abbildung 7.16. Das Salami Slice XML Schema Design Pattern**

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="gedicht">
 <xs:complexType>
 <xs:sequence>
 <xs:element name="autor"/>
 <xs:element name="titel"/>
 </xs:sequence>
 </xs:complexType>
 </xs:element>
 <xs:element name="autor">
 <xs:complexType>
 <xs:sequence>
 <xs:element ref="vorname"/>
 <xs:element ref="nachname"/>
 </xs:sequence>
 </xs:complexType>
 </xs:element>
 <xs:element name="vorname" type="xs:string"/>
 <xs:element name="nachname" type="xs:string"/>
 <xs:element name="titel" type="xs:string"/>
</xs:schema>
```

Das Salami Slice Design Pattern verfolgt sozusagen den umgekehrten Ansatz wie das Russian Doll Design Patterns, da hierbei das Modell in seine Einzelteile zerlegt wird, statt die Komponenten komplett zusammen zu halten (nach [GHJV95]):

- **Transparenter Inhalt.** Die Einzelteile eines Gedichts sind für andere Schemata und andere Teile desselben Schemas sichtbar. Dadurch werden diese Elemente wiederverwendbar.
- **Globaler Geltungsbereich.** Teilelemente einer Komponente sind global sichtbar. Dadurch wird die Komplexität erhöht.



- **Klarheit.** Alle Elemente und deren Typen sind klar ersichtlich und übersichtlich angeordnet.
- **Kopplung.** Wenn sich z. B. das Element `autor` ändern würde, würde dies Auswirkungen auf das Element `gedicht` haben; die einzelnen Komponenten sind gekoppelt.
- **Kohäsion.** Zusammengehörende Konzepte und Daten sind in sich geschlossen und gekapselt.

Das Salami Slice Design Pattern schützt nicht vor der Komplexität einzelner Komponenten, ermöglicht jedoch deren Wiederverwendbarkeit.

### 7.3.3 Venetian Blind

Soll sowohl ein Maximum an Wiederverwendbarkeit, als auch ein Schutz vor der Komplexität einzelner Komponenten erreicht werden, so empfiehlt es sich, das Venetian Blind Design Pattern verwenden. Hierbei teilt man die Komponenten wie beim Salami Slice Design Pattern in Einzelteile auf, jedoch beschreibt man nicht Elemente, sondern erstellt globale Typen für diese Elemente:

**Abbildung 7.17. Das Venetian Blind XML Schema Design Pattern**

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="gedicht" type="gedichtType"/>
 <xs:complexType name="gedichtType">
 <xs:sequence>
 <xs:element name="autor" type="autorType"/>
 <xs:element name="titel" type="xs:string"/>
 </xs:sequence>
 </xs:complexType>
 <xs:complexType name="autorType">
 <xs:sequence>
 <xs:element name="vorname" type="xs:string"/>
 <xs:element name="nachname" type="xs:string"/>
 </xs:sequence>
 </xs:complexType>
</xs:schema>
```

Die Charakteristika des Venetian Blind Design Patterns sind demnach (nach [GHJV95]):

- **Transparenter Inhalt.** Die Einzelteile eines Gedichts sind für andere Schemata und andere Teile desselben Schemas sichtbar. Dadurch werden diese Elemente wiederverwendbar.
- **Lokaler Geltungsbereich.** Teilelemente einer Komponente sind nur lokal sichtbar. Dadurch wird die Komplexität verringert.
- **Klarheit.** Alle Typen sind klar ersichtlich und übersichtlich angeordnet.
- **Kopplung.** Die einzelnen Komponenten sind gekoppelt.
- **Kohäsion.** Zusammengehörende Konzepte und Daten sind in sich geschlossen und gekapselt.

### 7.3.4 Garden of Eden

Das Garden of Eden Design Pattern (nach [BD04]) ist die logische Erweiterung der anderen drei. Beim Garden of Eden Design Pattern werden sowohl Elemente als auch deren Datentypen global beschrieben:

### Abbildung 7.18. Das Garden of Eden XML Schema Design Pattern

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="gedicht" type="gedichtType"/>
 <xs:complexType name="gedichtType">
 <xs:sequence>
 <xs:element ref="autor"/>
 <xs:element ref="titel"/>
 </xs:sequence>
 </xs:complexType>
 <xs:complexType name="autorType">
 <xs:sequence>
 <xs:element ref="vorname"/>
 <xs:element ref="nachname"/>
 </xs:sequence>
 </xs:complexType>
 <xs:element name="vorname" type="xs:string"/>
 <xs:element name="nachname" type="xs:string"/>
 <xs:element name="titel" type="xs:string"/>
</xs:schema>
```

Die Charakteristika des Garden of Eden Design Patterns sind somit:

- **Transparenter Inhalt.** Die Einzelteile eines Gedichts sind für andere Schemata und andere Teile desselben Schemas sichtbar und somit wiederverwendbar.
- **Globaler Geltungsbereich.** Teilelemente einer Komponente sind global sichtbar. Dadurch wird die Komplexität erhöht.
- **Klarheit.** Alle Typen sind klar ersichtlich und übersichtlich angeordnet.
- **Kopplung.** Die einzelnen Komponenten sind gekoppelt.
- **Kohäsion.** Zusammengehörende Konzepte und Daten sind in sich geschlossen und gekapselt.

### Tabelle 7.3. Zusammenfassung der XML Schema Design Patterns

Design Pattern	Elementdeklarationen	Typdefinitionen
Russian Doll	Lokal	Lokal
Salami Slice	Global	Lokal
Venetian Blind	Lokal	Global
Garden of Eden	Global	Global

Tabelle 7.3, „Zusammenfassung der XML Schema Design Patterns“ fasst die in diesem Abschnitt vorgestellten Design Patterns noch ein Mal anhand der Geltungsbereiche von Elementen und Typen zusammen.

---

## **Teil III. Datenmodell und Abfragesprachen in XML**

---

---

# Inhaltsverzeichnis

8. Baummodell vs. XQuery 1.0 und XPath 2.0 Datenmodell (XDM) .....	67
1.1 Orientierung .....	67
1.2 XML-Dokumentarten .....	67
1.3 Ein Baummodell für XML-Dokumente .....	67
1.4 XQuery 1.0 und XPath 2.0 Datenmodell (XDM) .....	70
1.4.1 Knoten .....	73
1.4.2 Atomic Values .....	75
1.4.3 Sequenzen .....	76
9. XPath .....	78
2.1 Orientierung .....	78
2.2 XPath-Ausdrücke .....	79
2.2.1 Schritte .....	79
2.2.2 Prädikate .....	81
10. XQuery als Abfragesprache für XML .....	86
3.1 Orientierung .....	86
3.2 Input Funktionen .....	86
3.2 Erzeugen von XML Knoten und Attributen .....	86
3.2.1 Miteinbezogene Elemente und Attribute aus der Input-Dokument .....	87
3.2.2 Direkte Element-Konstruktoren .....	87
3.2.3 Computed Constructors .....	90
3.3 Verbinden und Restrukturieren von Knoten (FLWOR) .....	92
3.3.1 For und Let Klausel .....	93
3.3.2 Where Klausel .....	96
3.3.3 Order by Klausel .....	97
3.3.4 Return Klausel .....	97
3.3.5 Joins .....	98
3.4 Operatoren und Bedingte Ausdrücke .....	100
3.4.1 Arithmetische Operatoren .....	101
3.4.2 Vergleichsoperatoren .....	101
3.4.3 Sequenz Operatoren .....	103
3.4.4 Bedingte Ausdrücke .....	103
3.5 Funktionen .....	104
3.5.1 Built-In Funktionen .....	104
3.5.2 Benutzerdefinierte Funktionen .....	105
3.6 Implementierungen .....	108
3.6.1 Saxon .....	108
3.6.2 eXist .....	109

---

# Kapitel 8. Baummodell vs. XQuery 1.0 und XPath 2.0 Datenmodell (XDM)

## 1.1 Orientierung

In den vorhergehenden Kapiteln standen Schema-Ansätze am Beispiel von DTD's und XML-Schema im Mittelpunkt. Wir haben die wichtigsten Syntax-Regeln für den Aufbau einer DTD bzw. eines XML-Schemas kennengelernt. Des weiteren haben wir einige Regeln, die für die DTD-Erstellung relevant sind, näher betrachtet.

In diesem und in den nächsten zwei Kapiteln werden wir uns vorwiegend mit den Sprachen XML Query Language (XQuery) und die XML Path Language (XPath), die zur Abfrage bzw. für die Adressierung von Teilen eines XML-Dokuments konzipiert wurden befassen, wobei zunächst die Prinzipien des zugrunde liegenden XQuery 1.0 und XPath 2.0 Datenmodell (XDM) erfolgt.

## 1.2 XML-Dokumentarten

Allgemein lassen sich Daten und somit auch Dokumente, anhand ihres beabsichtigen Gebrauchs und ihres Strukturierungsgrades in strukturierte, unstrukturierte und semistrukturierte Daten bzw. Dokumente klassifizieren.

- **Strukturierte (oder "datenzentrierte") Dokumente:** Dokumente, die hauptsächlich für die maschinelle Verarbeitung bestimmt sind. Die Dokumente haben ein spezielles Schema (Struktur) aufzuweisen, das heißt, dass die Daten in einer bestimmten Art und Weise angeordnet und verknüpft werden, um den Zugriff auf sie und ihre Verwaltung bequem zu ermöglichen.
- **Unstrukturierte (oder "dokumentzentrierte") Dokumente:** Dokumente, die von Menschen auch ohne zusätzliche Metainformationen verständlich sind. Die Elemente eines Dokuments werden hauptsächlich zur semantischen Markierung von Passagen des Dokuments genutzt (z.B. Kapitel oder Paragraphen eines Buches). Der Begriff „unstrukturiert“ ist ein wenig irreführend, da alle Dokumente eine gewisse Struktur haben, auch wenn diese Struktur nur implizit gegeben ist (z.B. Satzzeichen). Aufgrund der schwachen Strukturierung ist eine maschinelle Verarbeitung schwierig. XML könnte verwendet werden, um unstrukturierte Daten auszuzeichnen und zu repräsentieren. Diese Möglichkeit sollte aber vermieden werden. Im Allgemeinen ist XML für die semantische Auszeichnung gedacht. Die Präsentation der Daten sollte z.B. an der Extensible Stylesheet Language Transformation (XSLT) überlassen werden.
- **Semistrukturierte Dokumente:** Eine Mischform für Dokumente, die stärker strukturiert als dokumentzentrierte Dokumente und wiederum schwächer strukturiert als datenzentrierte Dokumente sind.

In der Praxis ist meistens nicht der Typ der reinen strukturierten Daten zu finden, da die meisten Daten nicht in sorgfältig strukturierte Form gesammelt werden. Solche Daten können wir in einem Baum- oder Graphen-Datenstrukturen sogenanntes Baummodell erfassen.

## 1.3 Ein Baummodell für XML-Dokumente

Generell werden zum Aufbau eines XML-Dokumentes zwei wichtige Strukturierungskonstrukte angewendet: Elemente und Attribute, wobei ein Attribut in XML die gleiche Bedeutung hat wie bei den Dokumentenbeschreibungssprachen HTML oder SGML, konkreter ausgedrückt: Attribute bieten zahlreiche zusätzliche Informationen, die Elemente beschreiben. Des weiteren werden anhand Textstrukturen weitere Informationen innerhalb von XML-Dokumente eingebunden.

Schauen wir den Aufbau des folgenden XML-Dokumentes an:

### Beispiel 8.1. gedicht.xml

```
<gedicht>
 <kopf>
 <titel>Als ich die Universität bezog</titel>
 <autor>Rainer Maria Rilke</autor>
 <jahr>1895</jahr>
 </kopf>
 <strophe xml:id="erste-strophe">
 <zeile>Ich seh zurück, wie Jahr um Jahr</zeile>
 <zeile>so müheschwer vorüberrollte;</zeile>
 <zeile>nun endlich bin ich, was ich wollte</zeile>
 <zeile>und was ich strebte: ein Skolar.</zeile>
 </strophe>
 <strophe>
 <zeile>Erst 'Recht' studieren war mein Plan;</zeile>
 <zeile>doch meine leichte Laune schreckten</zeile>
 <zeile>die strengen, staubigen Pandekten,</zeile>
 <zeile>und also ward der Plan zum Wahn.</zeile>
 </strophe>
 <strophe>
 <zeile>Theologie verbot mein Lieb,</zeile>
 <zeile>konnt mich auf Medizin nicht werfen,</zeile>
 <zeile>so daß für meine schwachen Nerven</zeile>
 <zeile>nichts als - Philosophieren blieb.</zeile>
 </strophe>
 <strophe>
 <zeile>Die Alma mater reicht mir dar</zeile>
 <zeile>der freien Künste Prachtregister,-</zeile>
 <zeile>und bring ichs nie auch zum Magister,</zeile>
 <zeile>bin was ich strebte: ein Skolar.</zeile>
 </strophe>
</gedicht>
```

Das obige Beispiel 8.1, „gedicht.xml“ zeigt ein XML-Element namens `gedicht`: Wie bereits erwähnt werden Elemente in einem XML-Dokument wie in HTML durch ihr Start- und End-Tag identifiziert. Die Tag-Namen stehen zwischen eckigen Klammern, `<...>`, und die End-Tags bekommen zusätzlich einen Schrägstrich, `</...>`. Elemente können wiederum in einfachen und komplexen Elementen klassifiziert werden. Komplexe Elemente werden aus anderen Elementen hierarchisch gebildet, während einfache Elemente nur aus reinem Text bestehen. Somit können wir in unserem Beispiel zwischen komplexen (`gedicht`, `kopf`, und `strophe`) und einfachen Elementen (`zeile`, `titel`, `autor` und `jahr`) unterscheiden.

Das Beispiel 8.1, „gedicht.xml“ könnte als Baummodell, das aus Knoten und Kanten besteht, dargestellt werden: Jedes Element (z. B. `strophe`) bildet einen Knoten und kann über Attribute (z. B. `id`) verfügen, neben den Elementknoten gibt es sogenannte Textknoten (z. B. *Als ich die Universität bezog*), die für den Inhaltstext zuständig sind. Alle Knoten mit Ausnahme des obersten Knotens ("Wurzelknoten") besitzen genau einen Elternknoten. Die Elementknoten haben einen Namen, der über einen Präfix eindeutig benannt werden kann. Alle Textknoten sind Blätter des Baums, haben daher auch keine Kindknoten.

Ein Baummodell für das Beispiel 8.1, „gedicht.xml“ könnte den nachfolgend gezeigten Aufbau haben:



## 1.4 XQuery 1.0 und XPath 2.0 Datenmodell (XDM)

Zunächst lernen wir das Beispiel, das uns in diesem und in den nächsten zwei Kapiteln begleiten wird, kennen. Im Anschluss werden wir das XQuery 1.0 und XPath 2.0 Datenmodell (XDM) vorstellen.

Unser Beispiel, was uns demnächst begleiten wird, ist eine aus iTunes exportierte Mediathek. Das Beispiel orientiert sich an klassische Musikstücke und Playlisten. Die wichtigsten Eigenschaften eines Musikstücks sind in der XML Dokument eingetragen (z.B. Name, Composer, Artist, Album, Genre, etc.). Beispiel 8.2, „Auszug aus generierte Mediathek.xml“ enthält Informationen über zwei Musikstücke. Die gesamte Mediathek enthält 714 Musik-stücke. Beispiel 8.3, „Auszug aus generierte PlaylistMediathek.xml“ enthält Informationen über eine Wiedergabeliste (engl. playlist). Die gesamte PlaylistMediathek enthält 45 Wiedergabelisten.



## Beispiel 8.2. Auszug aus generierte Mediathek.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<library>
 <MajorVersion>1</MajorVersion>
 <MinorVersion>1</MinorVersion>
 <ApplicationVersion>7.4.3</ApplicationVersion>
 <Features>1</Features>
 <ShowContentRatings/>
 <MusicFolder>file://localhost/C:/Users/brueggem/Music/iTunes/
 iTunes%20Music/</MusicFolder>
 <LibraryPersistentID>7ECC146FD9B46861</LibraryPersistentID>
 <Tracks>
 <track>
 <TrackID>464</TrackID>
 <Name>Titel 01</Name>
 <Kind type="aac">AAC-Audiodatei</Kind>
 <Size>8244635</Size>
 <TotalTime>508733</TotalTime>
 <TrackNumber>1</TrackNumber>
 <TrackCount>10</TrackCount>
 <DateModified>2007-08-17T18:33:58Z</DateModified>
 <DateAdded>2007-08-17T18:33:04Z</DateAdded>
 <BitRate>128</BitRate>
 <SampleRate>44100</SampleRate>
 <PlayCount>7</PlayCount>
 <PlayDate>3270873449</PlayDate>
 <PlayDateUTC>2007-08-25T06:57:29Z</PlayDateUTC>
 <Normalization>3084</Normalization>
 <PersistentID>7ECC146FD9B469EF</PersistentID> 14
 <Disabled/>
 <TrackType>File</TrackType>
 <Location>file://localhost/C:/Users/brueggem/
 Music/iTunes/iTunes%20Music/
 Unbekannter%20Interpret/
 Unbekanntes%20Album/01%20Titel
 %2001.m4a</Location>
 <FileFolderCount>4</FileFolderCount>
 <LibraryFolderCount>1</LibraryFolderCount>
 </track>
 <track>
 <TrackID>1031</TrackID>
 <Name>Violin Concerto In E Minor, Op. 64: I. Allegro
 Molto Appassionato</Name>
 <Artist>Fritz Kreisler</Artist>
 <Composer>Felix Mendelssohn</Composer>
 <Album>Mendelssohn And Brahms Concerti</Album>
 <Genre>Classical</Genre>
 <Kind type="aac">AAC-Audiodatei</Kind>
 <Size>11795355</Size>
 <TotalTime>727920</TotalTime>
```

```
<TrackNumber>1</TrackNumber>
<TrackCount>6</TrackCount>
<Year>1935</Year>
```

```
<DateModified>2007-10-12T23:25:51Z</DateModified>
<DateAdded>2007-10-12T23:24:41Z</DateAdded>
<BitRate>128</BitRate>
<SampleRate>44100</SampleRate>
<Normalization>8848</Normalization>
<PersistentID>1E5B6CC3960FDC4D</PersistentID>
<Disabled/>
<TrackType>File</TrackType>
<Location>file:///localhost/C:/Users/brueggem/
 Music/iTunes/iTunes%20Music/Fritz
 %20Kreisler/Mendelssohn%20And
 %20Brahms%20Concerti/01%20Violin
 %20Concerto%20In%20E%20Minor,
 %20Op.%206.m4a</Location>
<FileFolderCount>4</FileFolderCount>
<LibraryFolderCount>1</LibraryFolderCount>
</track>
</Tracks>
</library>
```

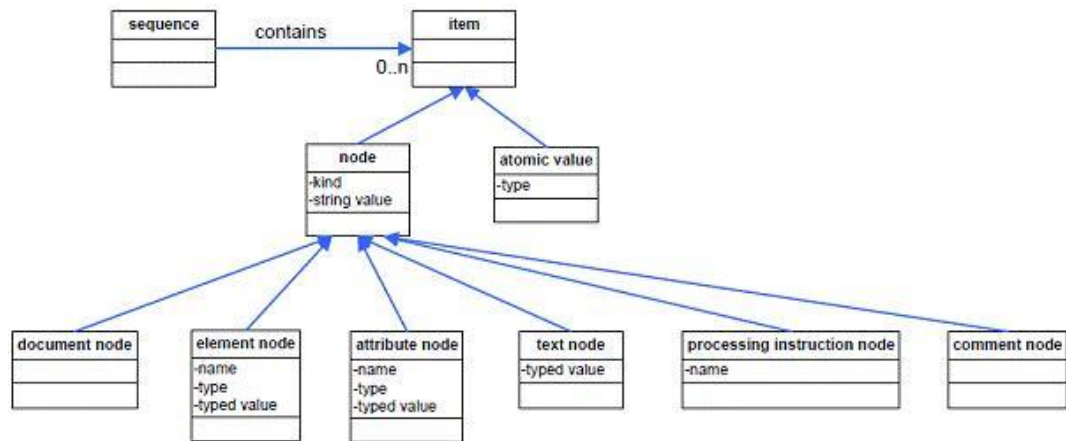
### Beispiel 8.3. Auszug aus generierte PlaylistMediathek.xml

```
<library>>
 <Playlists>
 <Playlist>
 <Name>Mediathek</Name>
 <Master/>
 <PlaylistID>1178</PlaylistID>
 <PlaylistPersistentID>7ECC146FD9B46862
 </PlaylistPersistentID>
 <Visible/>
 <AllItems/>
 <PlaylistItems>
 <TrackID>464</TrackID>
 <TrackID>1031</TrackID>
 </PlaylistItems>
 </Playlist>
 </Playlists>
</library>
```

Das XDM ist als ein formales Datenmodell definiert und nicht als XML-Text. Das XDM ist offiziell als XQuery 1.0 und XPath 2.0 Datenmodell bekannt. Das Datenmodell ist nicht mit dem Infoset (das W3C Modell für XML, [CT04]) identisch, weil das XDM auch solche Werte (Ergebnisse) unterstützen soll, die kein vollständiges, d.h. kein wohlgeformtes XML-Dokument sind. Ein Beispiel für ein solches Ergebnis ist eine Sequenz, die kein Wurzelement hat. Das Datenmodell beschreibt sowohl die Struktur der Eingabedokumente, als auch die Struktur der Ausgabe. Die Komponenten des XQuery 1.0 und XPath 2.0 Datenmodell sind:

- Knoten (engl. node): Ein XML- Konstrukt wie z.B. ein Element oder ein Attribut
- Atomic value: Ein einfacher Wert, bei dem kein Markup assoziiert ist.
- Item: Ein generischer Ausdruck, dass auf einem Knoten oder auf einem Datenwert verweist
- Sequenz: Eine geordnete Liste bestehend aus keinem, einem oder mehreren Items

**Abbildung 8.3. Datenmodellkomponenten**



XQuery 1.0 und XPath 2.0 Datenmodellinstanzen können auf verschiedene Art und Weise konstruiert werden. Die XDM-Spezifikation beschreibt, wie eine XDM Instanz aus dem Infoset und dem PSVI (kürz. Post-Schema-Validation-Infoset) erzeugt werden kann. Instanzen können auch direkt erzeugt werden, entweder als Ausgabe einer XQuery, oder durch direkte Konstruktion mittels einer Applikation. Die XDM Spezifikation definiert die XDM Instanz als eine Sequenz von Items, wobei jedes Item ein Knoten oder ein Wert ist.

## 1.4.1 Knoten

Die Knoten werden verwendet, um Konstrukte wie Elemente und Attribute zu repräsentieren. Knoten kommen als Ergebnis einer Anfrage oft vor. Z.B der Pfadausdruck `doc("Mediathek.xml")/library/Tracks/track2` gibt 714 Elementknoten zurück.

Im Allgemeinen werden 7 Knotenarten unterschieden:

- Document Node: Stellt das gesamte XML-Dokument (und nicht das äußerste Element) dar.
- Element Node: ein XML-Element
- Attribute Node: ein XML- Attribut
- Text Nodes: repräsentieren Zeichenketten, die in einem Element enthalten sind
- Processing Instruction Node: XML Befehlsverarbeitung
- Comment Node: ein XML- Kommentar
- Namespace: Jeder Namespace Knoten repräsentiert die Bindung von einer Namespace URI zu einem Namespace-Präfix oder zum Default-Namespace

Wie bereits erwähnt war das Hauptaugenmerk unser bereits vorgestellten Baummodells auf Element-, und Attributknoten gesetzt, da die beiden Knotenarten am meisten in die Abfragen verwendet werden. Im Allgemeinen werden die Begriffe „Element“ und „Attribut“, statt Elementknoten und Attributknoten, verwendet.

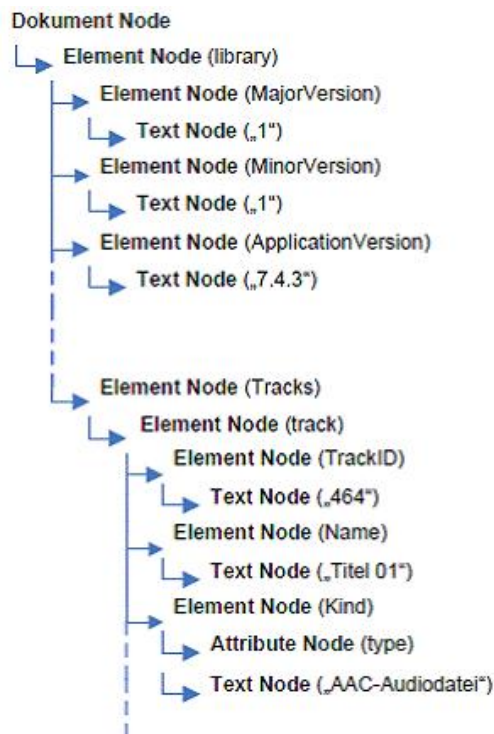
### 1.4.1.1 Knotenhierarchie

Ein XML-Dokument oder ein XML-Fragment ist mit Hilfe einer Knotenhierarchie aufgebaut. An dieser Stelle wird noch einmal auf Beispiel 8.2, „Auszug aus generierte Mediathek.xml“ verwiesen.

<sup>2</sup>`doc( )` ist eine Input-Funktion (Weitere Erläuterungen siehe „3.2 Input Funktionen“)

Wird dieses XML-Dokument (bzw. Fragment) im XDM übersetzt, wird die Knotenhierarchie ersichtlich (siehe Abbildung 8.4, „Knotenhierarchie“).

### Abbildung 8.4. Knotenhierarchie



#### 1.4.1.2 Knotenfamilie

Um die Verbindung zwischen den verschiedenen Knoten besser darzustellen, werden die Knoten in einer Art „Familie“ aufgeführt. Auf diese Weise hat jeder Knoten eine verschiedene Art und Anzahl von Verwandtschaften. In diesem Skript werden fünf Verwandtschaftsarten kategorisiert und unter Zuhilfenahme von Beispiel 8.4, „Knotenarten und Knotenverwandtschaft“ erklärt:

#### Beispiel 8.4. Knotenarten und Knotenverwandtschaft

```
<library>
 <track>
 <TrackID>1031</TrackID>
 <Name>Violin Concerto In E Minor,
Op. 64: I. Allegro Molto Appassionato</Name>
 <Artist>Fritz Kreisler</Artist>
 <Composer>Felix Mendelssohn</Composer>
 <Album>Mendelssohn And Brahms Concerti</Album>
 <Genre>Classical</Genre>
 <Kind type="aac">AAC-Audiodatei</Kind>
 <Size>11795355</Size>
 <TotalTime>727920</TotalTime>
 <TrackNumber>1</TrackNumber>
 <TrackCount>6</TrackCount>
 <Year>1935</Year>
 </track>
</library>
```

- **Children:** Ein Element kann keine, eine oder mehrere Elemente als Kinder haben. Es kann zusätzlich Text, Kommentar oder Instruktion als Kind aufweisen. Hier ist zu bemerken, dass Attribute nicht als Kind eines Elements aufgeführt werden. Ein Dokumentknoten kann ein Element als Kind haben. In diesem Fall ist dieses Element auch das äußerste (das root) Element. Kinder eines Dokumentknotens können auch Kommentare oder Instruktionen sein. Im Beispiel 8.4, „Knotenarten und Knotenverwandtschaft“: `TrackID`, `Name`, `Artist`, `Composer`, `Album`, `Genre`, `Kind`, `Size`, `TotalTime`, `TrackNumber`, `TrackCount` und `Year` sind Kinder von `track`.
- **Parent:** Das Parent (engl. für Eltern) von einem Element ist entweder ein übergeordnetes Element, oder der Dokumentknoten. Das Elternteil eines Attributs ist das Element, in dem das Attribut eingebunden ist. Im Beispiel 8.4, „Knotenarten und Knotenverwandtschaft“: Das Element `track` ist das Parent von `TrackID`, `Name`, `Artist`, `Composer`, `Album`, `Genre`, `Kind`, `Size`, `TotalTime`, `TrackNumber`, `TrackCount` und `Year`. Das Element `Kind` ist das Parent vom Attribut `aac`.
- **Ancestors:** Die Ancestors eines Elements sind alle "Vorfahren" dieses Elements. Anhand Beispiel 8.4, „Knotenarten und Knotenverwandtschaft“ kann die Beziehung besser veranschaulicht werden: Die Ancestors vom Element `TrackID` sind die Elemente `track` und `library`.
- **Descendants:** Die Descendants eines Elements sind alle "Nachkommen" dieses Elements. Anhand Beispiel 8.4, „Knotenarten und Knotenverwandtschaft“ kann die Beziehung besser veranschaulicht werden: Die Descendants des Elements `library` sind die Elemente `track`, `TrackID`, `Name`, `Artist`, `Composer`, `Album`, `Genre`, `Kind`, `Size`, `TotalTime`, `TrackNumber`, `TrackCount` und `Year`.
- **Siblings:** Elemente, die dieselben Eltern haben. Attribute innerhalb eines Elements sind keine Siblings. In Beispiel 8.4, „Knotenarten und Knotenverwandtschaft“ sind die Elemente `TrackID`, `Name`, `Artist`, `Composer`, `Album`, `Genre`, `Kind`, `Size`, `TotalTime`, `TrackNumber`, `Track-Count` und `Year` Siblings zueinander.

### 1.4.1.3 Knotenidentität

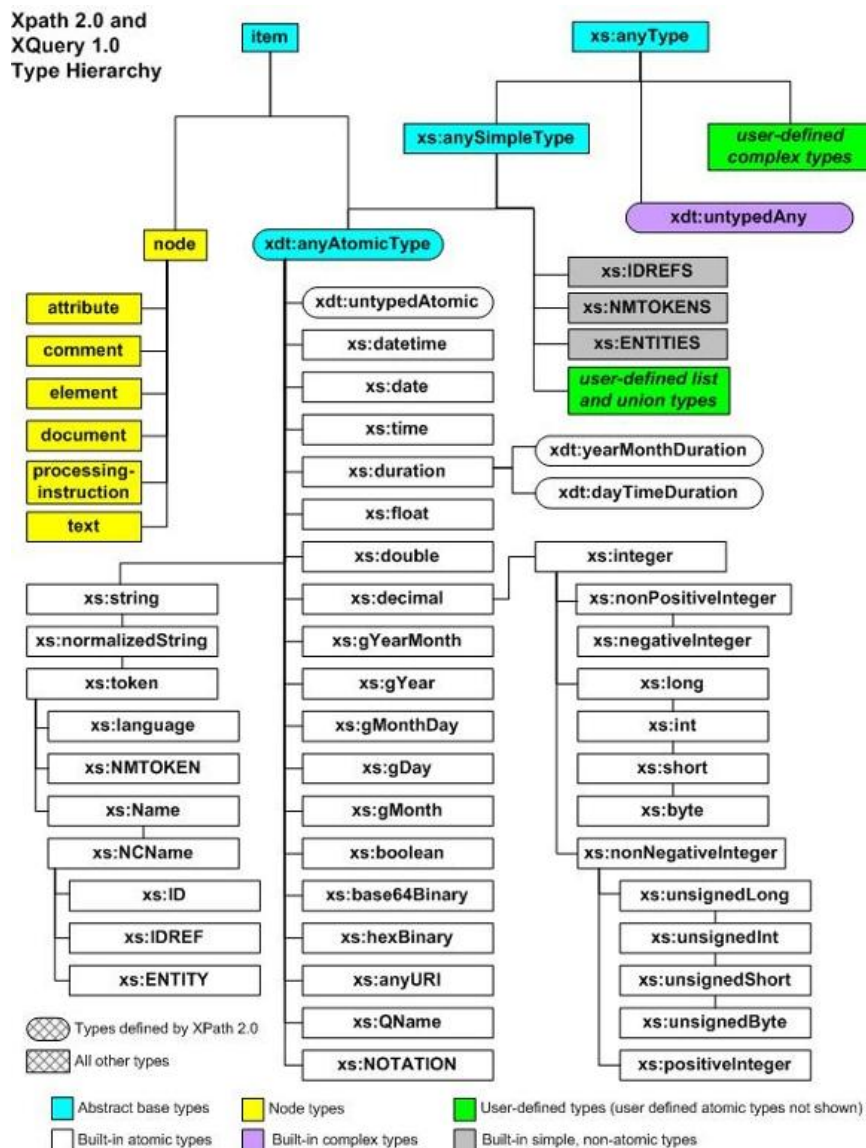
Jeder Knoten hat eine eindeutige Identität. Es kann vorkommen, dass ins Input-Dokument zwei oder mehrere Knoten existieren, die exakt den gleichen Inhalt haben. Das heißt jedoch nicht, dass die Knoten auch die gleiche Identität besitzen. Hier ist zu bemerken, dass die Identität jeden einzelnen Knoten eindeutig ist. Sie wird vom Prozessor zugeordnet und kann nicht abgerufen werden. Identitäten können jedoch mit dem `is` Operator (siehe „3.4.3 Sequenz Operatoren“) verglichen werden [W07-1].

Elemente und Attribute besitzen zusätzlich noch Namen. Die Namen können mittels der eingebauten Funktionen `name`, `node-name` und `local-name` abgerufen werden.

## 1.4.2 Atomic Values

Das Atomic Value ist ein Wert aus dem Werteraum eines atomic type. Atomic type ist seinerseits ein primitiver einfacher Datentyp oder ein Datentyp, der mit bestimmten Restriktionen von einen primitiven Datentyp abgeleitet ist. Es sind 23 primitive einfache Datentypen (engl. primitive simple datatypes) definiert [BMP04]. Die primitive simple datatypes können im Anhang (Anhang B, *Appendix: Primitive Datentypen*) nachgeschlagen werden (Siehe hierzu auch Abbildung 8.5, „Typhierarchie“). Der Wert `aac` ist ein Beispiel für atomic values.

Abbildung 8.5. Typhierarchie



## 1.4.3 Sequenzen

Sequenzen sind eine geordnete Reihe von Items. Eine Sequenz kann keine, genau eine oder mehrere Items beinhalten. Anschließend ist jedes Item entweder ein Knoten, oder ein Atomic Value.

Meistens werden Sequenzen erzeugt, nachdem ein Ausdruck oder eine Funktion ausgeführt wurde. Die Ergebnisse sind dann in der Form einer Sequenz. Z.B. der Ausdruck `doc („Mediathek.xml“)/library/Tracks/track` liefert eine Sequenz von 714 Items (in diesem Fall Musikstücke) zurück.

Eine Sequenz kann zusätzlich explizit erzeugt werden. Dazu wird ein Sequenz-konstruktor benötigt. Der Sequenzkonstruktor ist folgendermaßen aufgebaut: Eine Reihe von Werten, die mit Kommas voneinander getrennt sind. Der Ausdruck `( "a", "b", 3 )` erzeugt eine Sequenz, die die drei Werte beinhaltet. Der Sequenz-konstruktor kann zusätzlich ein oder mehrere Ausdrücke enthalten. Z.B. der Ausdruck `( doc("Mediathek.xml")/library/Tracks/track, "a", "b", 3 )` liefert eine Sequenz mit 717 Items (die 714 Musikstücke und die 3 Werte). An der Stelle ist zu bemerken, dass der Sequenzkonstruktor in Klammern umschlossen ist.

Sequenzen haben keine Namen, können aber an einer Variablen gebunden werden. Z.B. die `let` Klausel bindet die Sequenz von 714 Musikstücke an einer Variable (in diesem Fall `$track`):

```
let $track := doc("Mediathek.xml")/library/Tracks/track
```

Bei einer Sequenz, bestehend nur aus einem Item, gibt es keinen Unterschied zwischen die Sequenz selber und das beinhaltete Item. Deswegen können Funktionen, die auf Items operieren, verwendet werden.

Eine leere Sequenz (Sequenz ohne Items) ist unterschiedlich zum leeren String ("") oder zum Wert 0. Viele vordefinierte Funktionen akzeptieren die leere Sequenz als Argument. Der Pfadausdruck `doc("Mediathek.xml")/library/Tracks/track_foo` würde eine leere Sequenz liefern, wenn kein Element `track_foo` in dem Dokument "Mediathek.xml" existiert.

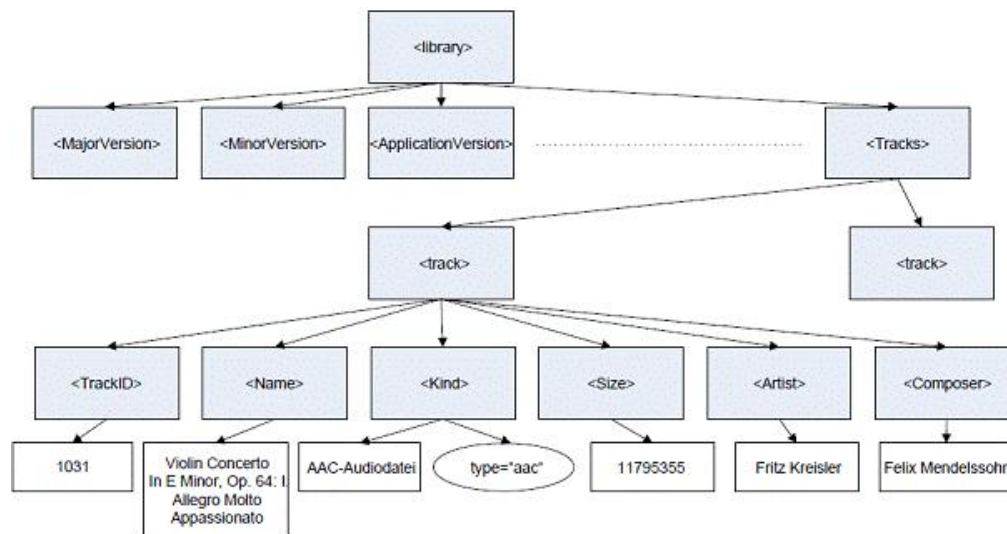
Ähnlich wie bei Atomic Values haben Sequenzen keine Identität. Es macht kein Sinn zu fragen ob `("a", "b", 3)` und `("a", "b", 3)` dieselbe Sequenz ist. Es kann abgefragt werden, ob der Inhalt gleich ist oder nicht.

# Kapitel 9. XPath

## 2.1 Orientierung

XML-Dokumente sind in Prinzip hierarchisch aufgebaut. Deshalb ist es einfach ein XML-Dokument als Baum zu repräsentieren (Siehe Abschnitt „1.3 Ein Baummodell für XML-Dokumente“ und Abschnitt „1.4 XQuery 1.0 und XPath 2.0 Datenmodell (XDM)“). Das "Mediathek" Dokument, das in Teil III, „Datenmodell und Abfragesprachen in XML“ vorgestellt wurde, ist als Baum in Abbildung 9.1, „Das "Mediathek" Dokument“ dargestellt. Die Abbildung ist nicht vollständig, sie soll vor allem dazu dienen, den Zusammenhang von Baummodell und XQuery bzw. XPath zu erleuchten.

**Abbildung 9.1. Das "Mediathek" Dokument**



Was für Fragen könnten anhand Abbildung 9.1, „Das "Mediathek" Dokument“ gestellt werden? Es könnte z.B. abgefragt werden, wie der Name eines Musikstücks lautet. Wenn die Daten in eine relationale Datenbank gespeichert sind, und als Abfragesprache SQL verwendet wird, so sollte bekannt gemacht werden in welcher Zelle (Tabelle, Spalte, Zeile) diese Information steckt [BM06]. Bei einem XML-Dokument könnte der Name des Musikstücks auf zwei verschiedene Art und Weisen gefunden werden.

Die erste Möglichkeit ist nach dem Wert des Namens zu fragen. D.h. "Liefere den Wert des Name Elements". Dieses Vorgehen ist in Ordnung, falls das XML-Dokument einfacher ist. Was passiert, wenn es mehrere Name Elemente innerhalb des Dokumentes gibt – z.B. ein Name Element für den Namen des Musikstücks, und ein Name Element für den Namen des Komponisten oder des Musikers?

Die zweite Möglichkeit nach dem Namen des Musikstücks zu fragen ist den Baum durchzulaufen. Bei diesem Verfahren müssen explizite Instruktionen gegeben werden, wie von dem Wurzelement (wenn das ein XML Dokument ist, hat es nur ein Wurzelement) zum gewünschten Element navigiert werden soll. Eine Anfrage könnte folgendermaßen formuliert werden: "Starte mit dem Wurzelement, gehe nach unten bis zum ersten Kindelement, und gib den Wert zurück". Es kann einfach die Struktur durchgelaufen werden, wenn bekannt ist, wo sich die Daten im Baum befinden. Jedoch ist es vernünftiger Bedingungen (Prädikaten) zu setzen, um z.B. die Knotennamen zu untersuchen.

XPath ist eine Sprache zum Adressieren (Lokalisieren) von Teilen eines XML-Dokuments (XML-Baumes) mittels Pfaden. Im Zusammenhang mit der Standardisierung von XQuery wurde am 23. Januar 2007 mit XPath Version 2.0 die seit 1999 gültige Version XPath 1.0 verabschiedet. XPath 2.0 hat einige Konzepte von XPath 1.0 neu definiert und verfügt zudem über einen wesentlich erweiterten Funktionsumfang. Die XPath-Funktionalität ist eng mit dem darunterliegenden Datenmodell verbunden. Mit einigen Beispielen wird erklärt, was hinter die Pfadausdrücke steckt. Dann werden sie in Bezug auf das Datenmodell definiert.



## 2.2 XPath-Ausdrücke

Das primäre syntaktische Konstrukt in XPath ist der Ausdruck. Ein XPath-Ausdruck setzt sich zusammen aus:

- einem oder mehreren Lokalisierungsschritten (Location Steps).
- optional gefolgt von einem oder mehreren Prädikaten (predicates).

### 2.2.1 Schritte

Die am meisten verwendeten Operatoren in Pfadausdrücke lokalisieren Knoten, indem sie die Stelle des Knotens in der Baumhierarchie identifizieren. Ein Pfadausdruck beinhaltet eine Reihe von ein oder mehreren Schritte (engl. steps), die voneinander durch / (Schrägstrich), oder // (Doppelschrägstrich) getrennt sind. Jeder Schritt wird zu einer Knotensequenz ausgewertet. Das folgende Beispiel soll beachtet werden:

#### Beispiel 9.1. Einfache Navigation durch den XML-Baum

```
doc("Mediathek.xml")/library/Tracks/track/Name
```

Anfrage:	Finde die Namen aller Musikstücke	
XPath Ausdruck:	doc("Mediathek.xml")/library/Tracks/track/Name	
Erläuterung:	doc("Mediathek.xml")	öffnet das Dokument "Mediathek.xml"
	doc("Mediathek.xml")/	Selektiert den imaginären Wurzelknoten des Baumes
	doc("Mediathek.xml")/library	Selektiert die Kinderknoten namens "library"
	doc("Mediathek.xml")/library/	Selektiert die Kinderknoten von /library
	doc("Mediathek.xml")/library/Tracks	Selektiert die Kinderknoten namens Tracks
	doc("Mediathek.xml")/library/Tracks/	Selektiert die Kinderknoten von /library/Tracks
	doc("Mediathek.xml")/library/Tracks/track	Selektiert die Kinderknoten namens track
	doc("Mediathek.xml")/library/Tracks/track/	Selektiert die Kinderknoten von /library/Tracks/track
	doc("Mediathek.xml")/library/Tracks/track/Name	Selektiert die Kinderknoten namens Name

Der Pfadausdruck aus Beispiel 9.1, „Einfache Navigation durch den XML-Baum“ öffnet das Dokument "Mediathek.xml" mit Hilfe der doc()-Funktion. Die doc()-Funktion gibt den Dokumentknoten zurück. Der Ausdruck verwendet /library, um das library Element zu selektieren. Der Ausdruck verwendet weiterhin /Tracks um das Tracks Element innerhalb (bzw. unterhalb) library zu selektieren. Der Ausdruck verwendet track, um alle track Elemente innerhalb Tracks zu selektieren. /Name wird verwendet, um die Name Elemente innerhalb track auszuwählen. Der angegebene Pfadausdruck enthält 5 Schritte. Dieselben Namen der Musikstücke könnten auch mit Hilfe einer Anfrage, die // verwendet, gefunden werden. Die Anfrage doc("Mediathek.xml")//Name würde alle Name Elemente ausgeben, egal an welchen Level sich die Elemente befinden haben.

Im Beispiel Beispiel 9.1, „Einfache Navigation durch den XML-Baum“ wurde gezeigt, dass Schritte direkte Ausdrücke sein können, wie z.B. ein Funktionsaufruf (`doc("Mediathek.xml")`) oder Referenz auf einer Variable (z.B. `$track`). Jeder (Pfad-)Ausdruck, der Knoten zurückliefert, kann auf der linken Seite des Schrägstrichoperators (/) gesetzt werden.

Eine andere Art von Schritte sind die so genannten Achsenschnitte (engl. axis step), die es möglich machen, durch die XML-Knotenhierarchie zu navigieren. Es sind zwei Arten von Achsenschnitten definiert:

- **Vorwärtsschritt** – Dieser Schritt selektiert alle Nachkommen oder Knoten, die nach dem Kontextknoten auftreten
- **Rückwärtsschritt** – Dieser Schritt selektiert alle Vorfahren oder Knoten, die vor dem Kontextknoten auftreten.

Im letzten Beispiel sind `library`, `Tracks`, `track`, `Name` Achsenschnitte (in diesem Fall sind es Vorwärtsschnitte). Die Syntax jeder einzelne Schritt wird folgendermaßen definiert.

```
achse::knotentest[prädikat1][prädikat2]...
```

Ein Lokalisierungsschritt besteht daher aus:

- Achse (axis).
- Knotentest (node-test).

## 2.2.1.1 Achsen

Jeder Vorwärts- / oder Rückwärtsschritt hat eine Achse, die die Richtung und die Beziehung des selektierten Knotens definiert. Zum Beispiel die `child::` Achse (Vorwärtsschritt) kann verwendet werden, um nur Kinderknoten zu selektieren. Andererseits kann die `parent::` Achse (Rückwärtsschritt) verwendet werden, um nur den Elternknoten eines Knotens zu selektieren. Die 12 Achsen sind in Tabelle 9.1, „Achsen“ aufgelistet und erläutert.

**Tabelle 9.1. Achsen**

Achsen	Bedeutung
<code>self::</code>	Der Kontext-Knoten
<code>child::</code>	Kinder des Kontext-Knoten. Attribute sind keine Kinder eines Elements. Wenn keine Achse angegeben ist, ist das die default Achse.
<code>descendant::</code>	Alle Nachkommen des Kontext-Knoten. Attribute sind keine Nachkommen.
<code>descendant-or-self::</code>	Der Kontext-Knoten und seine Nachkommen
<code>attribute::</code>	Attribut/e des Kontext-Knoten
<code>following::</code>	Liefert alle Knoten, die nachfolgende Geschwisterknoten oder deren Nachkommen sind.
<code>following-sibling::</code>	Liefert alle nachfolgenden Geschwisterknoten des aktuellen Knotens.
<code>parent::</code>	Die Eltern des Kontext-Knoten. Das ist entweder das Element oder der Dokument-Knoten. Der Eltern eines Attributs ist das Element selber.
<code>ancestor::</code>	Alle Vorfahren des Kontext-Knoten
<code>ancestor-or-self::</code>	Der Kontext-Knoten und alle Vorfahren

Achsen	Bedeutung
<code>preceding::</code>	Liefert alle Knoten die innerhalb der Dokumentstruktur vor dem aktuellen Knoten liegen aber keine Vorfahren des aktuellen Knotens sind.
<code>preceding-sibling::</code>	Liefert alle vorhergehenden Geschwisterknoten des aktuellen Knotens.

### Beachte:

Die verschiedene Implementierungen sind nicht aufgefordert die folgenden Achsen zu unterstützen: `following`, `following-sibling`, `ancestor`, `ancestor-or-self`, `preceding` und `preceding-sibling`.

## 2.2.1.2 Knotentests

Jeder Achsensschritt hat zusätzlich ein Knotentest. Mit Hilfe von Knotentests lässt sich die Suche nach dem gewünschten Knoten oder der gewünschten Knotenmenge noch näher beschreiben. Ein Knotentest ist dabei nichts weiter als eine zusätzliche Definition zu den oben genannten Achsendefinitionen. Als Knotentest können folgende Werte verwendet werden:

- **Knotenname** - Wird ein Name angegeben, so wird innerhalb der Knotenmenge aus der Achsendefinition nach den Knoten gesucht, die über einen solchen erweiterten Namen verfügen. Als Ergebnis erhält man damit die Knotenmenge der Knoten mit dem entsprechenden erweiterten Namen.
- **\*** - Der Stern bedeutet, dass alle Knoten innerhalb der Knotenmenge der Achsendefinition gefunden werden sollen - egal welchen Namen sie tragen.
- **Präfix:Name** oder **Präfix:\*** - Bei Angabe eines Präfixes wird nach allen Knoten aus dem entsprechenden Namensraum gesucht die den angegebenen Namen erfüllen. Alternativ kann auch der Stern statt dem Namen verwendet werden - dann werden alle Knoten des Namensraums gefunden - egal welchen Namen sie tragen.
- **text()** - Wählt alle Textknoten aus.
- **comment()** - Wählt alle Kommentarknoten aus.
- **processing-instruction()** - Wählt alle Processing Instruction-Knoten aus. Optional kann als Parameter innerhalb der Klammern der Name der zu suchenden Processing Instruktion angegeben werden. `processing-instruction("xml-stylesheet")` findet alle PI's mit dem Namen `xml-stylesheet`.
- **node()** - Wählt alle Knoten beliebigen Typs aus.

## 2.2.2 Prädikate

Prädikate dienen noch einmal der Verfeinerung einer Suche. Sie sind optional, werden dann aber immer mit eckigen Klammern ( `[]` ) notiert. Innerhalb dieser Klammern kann ein nahezu beliebiger Ausdruck stehen der eine gültige Aussage liefert.

### 2.2.2.1 Wertepredikat

Mit der Eingabe eines Wertepredikats wird das XML-Dokument durchgelaufen. Werte werden extrahiert anhand ihrer Position im Baum. Wenn XML-Daten abgefragt werden sollen, soll es möglich sein den Baum entsprechend vorgegebenen Bedingungen (Prädikaten) durchlaufen zu können. Beispiel 9.2, „Anfrage mit Wertepredikat“ zeigt, wie XPath solche Prädikate auswertet. Baumzweige, die die Bedingung nicht erfüllen werden nicht berücksichtigt.

### Beispiel 9.2. Anfrage mit Werteprädikat

Anfrage:	Finde den Namen aller Musikstücke, die von Johannes Brahms komponiert sind	
XPath Ausdruck:	doc("Mediathek.xml")/library/Tracks/ track[Composer = "Johannes Brahms"]/Name	
Erläuterung:	doc("Mediathek.xml")	öffnet das Dokument "Mediathek.xml"
	doc("Mediathek.xml")/	Selektiert den imaginären Wurzelknoten des Baumes
	doc("Mediathek.xml")/ library	Selektiert die Kinderknoten namens "library"
	doc("Mediathek.xml")/ library/	Selektiert die Kinderknoten von /library
	doc("Mediathek.xml")/ library/Tracks	Selektiert die Kinderknoten namens Tracks
	doc("Mediathek.xml")/ library/Tracks/	Selektiert die Kinderknoten von /library/Tracks
	doc("Mediathek.xml")/ library/Tracks/track	Selektiert die Kinderknoten namens track
	doc("Mediathek.xml")/ library/Tracks/ track[Composer = "Johannes Brahms"]	Aus der Sequenz von track Knoten, selektiere nur die Knoten, für die der Wert des Elements <i>Composer</i> gleich " <i>Johannes Brahms</i> " ist
	doc("Mediathek.xml")/ library/Tracks/ track[Composer = "Johannes Brahms"]/Name	Aus der Sequenz von track Knoten, wo <i>Composer</i> gleich <i>Johannes Brahms</i> ist, selektiere nur die Kinderknoten namens <i>Name</i>

### 2.2.2.2 Prädikate für Position

Beim Hinzufügen eines Prädikats für Position (Beispiel 9.3, „Anfrage mit Prädikat für Position“) ist die Möglichkeit gegeben den n-ten Knoten aus einer Knotensequenz auszuwählen. Das gilt nur wenn das XML-Dokument eine persistente Anordnung hat. Das XDM definiert die Dokument Anordnung (engl. document order) folgendermaßen: "Informally, document order is the order in which nodes appear in the XML serialization of a document" [CR99-2]. Die Dokument-Anordnung ist einer der Tatsachen, die den Unterschied zwischen XML-Daten und z.B. SQL-Daten macht. In relationale Datenbanken ist die Anordnung der Zeilen undefiniert. Eine Query muss explizit die Anordnung setzen, da sonst die Ergebnisse ungeordnet zurückgegeben werden.

### Beispiel 9.3. Anfrage mit Prädikat für Position

Anfrage:	Finde den Namen des 200-sten Musikstücks	
XPath Ausdruck:	doc("Mediathek.xml")/library/Tracks/track[200]/Name	
Erläuterung:	doc("Mediathek.xml")/ library/Tracks/track	Selektiert die Kinderknoten namens track
	doc("Mediathek.xml")/ library/Tracks/ track[200]	Aus der Sequenz von track Knoten, selektiere den Knoten am Position 200
	doc("Mediathek.xml")/ library/Tracks/ track[200]/Name	Aus dem 200-sten track-Knoten, selektiere nur die Kinderknoten namens <i>Name</i>

### 2.2.2.3 Das Kontext-Item

In Beispiel 9.4, „Das Kontext-Item“ wird die Funktion `contains` verwendet. Die `contains()`-Funktion ist eine in XPath/XQuery eingebundene Funktion [MMW07-2], die zwei String Parameter einnimmt. Die Funktion liefert *true* zurück, wenn der erste Parameter den zweiten beinhaltet. Dieses Beispiel zeigt, wie das Kontext-Item (`.`) verwendet wird. Das Kontext-Item stellt den aktuell betrachteten Knoten dar. Das Prädikat wird auf jeden Name Element angewendet.

#### Beispiel 9.4. Das Kontext-Item

Anfrage:	Finde alle Musikstücknamen, die den String <i>Sonata</i> beinhalten	
XPath Ausdruck:	<code>doc("Mediathek.xml")/library/Tracks/ track/Name[contains(., "Sonata")]</code>	
Erläuterung:	<code>doc("Mediathek.xml")/ library/Tracks/ track/Name</code>	Selektiert alle Musikstücknamen
	<code>doc("Mediathek.xml")/ library/Tracks/ track/Name[contains (., "Sonata")]</code>	Filtert die Sequenz von Name-Knoten mit der Bedingung <code>contains(., "Sonata")</code> . Die Funktion <code>contains</code> ist eine eingebaute Funktion. Der erste Parameter ( <code>.</code> ) ist das Kontext-Item.
	<b>Beachte:</b>  <code>doc("Mediathek.xml")/library/Tracks/ track[contains(Name, "Sonata")]/Name</code> ist äquivalent zum oberen XPath Ausdruck	

### 2.2.2.4 Der Baum in beiden Richtungen durchlaufen

Der XPath-Ausdruck aus Beispiel 9.5, „XML Baum nach oben und nach unten durchlaufen“ zeigt das Durchlaufen des Baumes bis zum Blatt-Knoten `Composer`. Anschließend wird der Baum "zurück" bis zum Elternknoten (`.`) durchgelaufen. Demnächst wird in einem anderen Baumzweig verzweigt (nach `Name`) um die Bedingung anzuwenden. Der äquivalente XPath Ausdruck durchläuft den Baum bis zum `track`-Knoten. Dann wird weiter nach unten (bis zum `Name`-Knoten) gegangen um die Bedingung anzuwenden. Anschließend wird von `track` nach `Composer` heruntergegangen um das Ergebnis auszuwählen.

### Beispiel 9.5. XML Baum nach oben und nach unten durchlaufen

Anfrage:	Finde die Komponisten in deren Musikstückname, den String <i>Sonata</i> beinhaltet ist	
XPath Ausdruck:	doc("Mediathek.xml")/library/Tracks/track/Composer [contains(..Name, "Sonata")]	
Erläuterung:	doc("Mediathek.xml")/library/Tracks/track/Composer	Selektiert alle Komponisten
	doc("Mediathek.xml")/library/Tracks/track/Composer [contains(..Name, "Sonata")]	Filtert die Sequenz von Composer-Knoten indem getestet wird, ob die Eltern (..) von Composer-Knoten ein Kind namens Name haben. Weiterhin wird getestet, ob die Name-Knoten den String <i>Sonata</i> beinhalten
	<b>Beachte:</b>  doc("Mediathek.xml")/library/Tracks/track[contains(Name, "Sonata")]/Composer ist äquivalent zum oberen XPath Ausdruck	

### 2.2.2.5 Vergleich zwischen verschiedene Baumzweige

Beispiel 9.6, „Vergleich zwischen verschiedene Baumzweige“ umfasst das Durchlaufen des Baumes in zwei verschiedene Baumzweige und das Vergleichen der Ergebnisse.

#### Beispiel 9.6. Vergleich zwischen verschiedene Baumzweige

Anfrage:	Finde alle Musikstücknamen, die die letzte Komposition eines CDs sind.	
XPath Ausdruck:	doc("Mediathek.xml")/library/Tracks/track/Name[../TrackNumber eq ../TrackCount]	
Erläuterung:	doc("Mediathek.xml")/library/Tracks/track/Name	Selektiert alle Musikstücknamen
	doc("Mediathek.xml")/library/Tracks/track/Name [../TrackNumber eq ../TrackCount]	Für jede Musikstückname gehe den Baum hinauf, und anschließend hinunter um den TrackNumber zu finden. Mache nochmal das gleiche um TrackCount zu finden. Gib diese Musikstücknamen, bei denen TrackNumber gleich zum TrackCount ist.
	<b>Beachte:</b>  doc("Mediathek.xml")/library/Tracks/track[TrackNumber eq TrackCount]/Name ist äquivalent zum oberen XPath Ausdruck	

## 2.2.2.6 Finden eines Elements anhand seinen Namen

### Beispiel 9.7. Element anhand seines Namens finden

Anfrage:	Finde alle Knoten, die Name heißen.	
XPath Ausdruck:	<code>doc("Mediathek.xml")//Name</code>	
Erläuterung:	//	Selektiert alle Knoten im Dokument
	//Name	Aus allen Knoten wähle die Name Knoten

Am Anfang dieses Kapitels wurde erwähnt, dass es möglich sein soll, Elemente anhand ihres Namens zu suchen. Beispiel 9.7, „Element anhand seines Namens finden“ zeigt, wie alle Musikstücknamen als Ergebnis zurückgeliefert werden können. Im Prinzip werden hier alle Elemente, die Name heißen, abgefragt. Diese Methode ist jedoch als gefährlich angesehen. Werden wirklich alle Name-Elemente gebraucht? Was passiert, wenn es noch ein Name-Element unter z.B. `Composer` gibt? Hier ist aber der Kontext wichtig und mit der Ausdruck `//Name` wird kein Kontext gesetzt.

---

# Kapitel 10. XQuery als Abfragesprache für XML

## 3.1 Orientierung

Mit Ihrer Version 2.0 bildet XPath eine Grundlage für viele aktuelle XML-Technologien (XQuery, XSLT, etc.). Nachdem wir wissen, wie Elemente im Dokumentenbaum durch Navigation lokalisiert werden, lernen wir in diesem Kapitel die Abfragesprache XQuery kennen.

Die XQuery 1.0 Abfragesprache ist entwickelt worden, um einen strikten Superset von XPath 2.0 zu sein. D.h. jeder gültige XPath 2.0 Ausdruck ist ein gültiger XQuery 1.0 Ausdruck (Query). Mit Hilfe der XPath Ausdrücken, können verschiedene Informationen aus XML-Dokumente extrahiert werden. In Prinzip ist XQuery viel mächtiger als XPath, da es möglich macht Informationen über mehrere Dokumente zu verbinden (join) und neue XML-Fragmente zu generieren. Zusätzlich ermöglicht XQuery die Definition von Funktionen (built-in, user-defined functions).

In diesem Kapitel werden die Input-Funktionen erklärt. Anschließend wird gezeigt, wie neue XML-Fragmente (Elemente und Attribute) erzeugt werden können. Die FLWOR Ausdrücke werden ins Detail behandelt. Mit deren Hilfe ist es möglich Knoten miteinander zu verbinden oder zu rekonstruieren. Anschließend werden die eingebauten Funktionen, sowie die benutzerdefinierte Funktionen erklärt. Schließlich werden zwei Implementierungen vorgestellt, die XQuery verwenden

## 3.2 Input Funktionen

XQuery verwendet Eingabefunktionen (engl. input functions) um die Daten zu identifizieren, die abgefragt werden. Es existieren zwei Input-Funktionen:

- `doc()` – liefert das gesamte Dokument zurück. Das Dokument wird anhand eines Universal Resource Identifier (URI) identifiziert. Um genauer zu sein, gibt die `doc()`-Funktion den Dokumentknoten (document node) zurück.
- `collection()` – liefert eine Kollektion zurück. Die `collection()`-Funktion mit einen Argumenten nimmt ein String, der die URI beinhaltet. Wenn die URI mit einer Kollektion assoziiert ist, wird die Datenmodellrepräsentation dieser Kollektion ausgegeben. Die Kollektion ist eine Sequenz von Knoten, die mit einem URI assoziiert sind. Die `collection()`-Funktion ohne Argument gibt die default collection (eine während der Implementierung definierte Knotensequenz) zurück.

Die `collection()`-Funktion wird meistens verwendet, um eine Datenbank zu identifizieren. Die Datenbankinhalte werden anschließend in der Query verwendet bzw. abgefragt.

Wenn die Daten in einer Datei namens "Mediathek.xml" gespeichert sind, dann gibt die folgende Query das gesamte Dokument zurück:

```
doc("Mediathek.xml")
```

Wenn die `doc()`-Funktion die angegebene Datei nicht lokalisieren kann, wird eine dynamische Ausnahme (engl. dynamic error) ausgeworfen. Dasselbe gilt wenn die `collection()`-Funktion die angegebene Kollektion nicht finden kann.

## 3.2 Erzeugen von XML Knoten und Attributen

In den meisten Fällen beinhalten die Abfragen einige XML-Elemente sowie Attribute, die den Ergebnissen eine Struktur geben. Im Kapitel 9, *XPath* wurde gezeigt, wie mit Hilfe von XPath-



Ausdrücke Elemente und Attribute von einem Input-Dokument aufgefunden werden können. In diesem Abschnitt wird erläutert, wie komplett neue Elemente und Attribute generiert werden können. Es wird zusätzlich erklärt, wie die erzeugten Elemente bzw. Attribute in den Ergebnissen eingebunden werden können.

Es gibt zwei Möglichkeiten Elemente und Attribute zu erzeugen:

- direkte Konstruktoren (engl. direct constructs): Ein direct constructor ist ein Konstruktor, bei dem der Name des konstruierten Elements eine Konstante ist. Ein Konstruktor für ein Element entspricht genau der XML-Syntax für ein Element.
- computed constructors: Ein computed constructor ist ein Konstruktor, bei dem der Name des konstruierten Elements dynamisch in der Abfrage generiert wird

## 3.2.1 Miteinbezogene Elemente und Attribute aus der Input-Dokument

Einige Abfragen ziehen Elemente und Attribute in die Ergebnisse mit sich. Beispiel 10.1, „Elemente aus dem Input-Dokument“ beinhaltet die ausgewählten Elemente in die Ergebnisse.

### Beispiel 10.1. Elemente aus dem Input-Dokument

```
(:Query:)
for $track in doc("Mediathek.xml")/library/Tracks/track
where $track/Artist = "Fritz Kreisler, Michael Raucheisen"
return $track/Name
(:Ergebnis:)
<Kreisler: Pollchinelle>
<Kreisler: La Précieuse (In The Style Of Couperin)>
<Kreisler: Chanson Louis Xiii & Pavane (In The Style Of Couperin)>
<Schubert/Kreisler: Rosamunde, D 797 - Ballet #2>
<Brandl/Kreisler: The Old Refrain>
<Glazunov/Kreisler: Serenade Espagnole>
<Weber/Kreisler: Larghetto>
<Heuberger/Kreisler: Midnight Bells>
```

Beachte: Da das gesamte Name-Element zurückgegeben wird, beinhalten die Ergebnisse die Name-Elemente (öffnende und schließende Tags) und nicht nur ihre Atomic Values. Wenn die Abfrage Elemente zurückgibt, die Attribute und Nachfolger haben, dann sind alle Elemente bzw. Attribute ein Teil des Ergebnisses. Wenn die return-Klausel aus Beispiel 10.1, „Elemente aus dem Input-Dokument“ `return $track` wäre, dann würden alle Informationen (mit Elementen und Attributen) des jeweiligen Musikstücks ausgegeben.

Wenn Pfadausdrücke verwendet werden, gibt es keine Möglichkeit Kinder-Elemente, Attribute oder Namensräume hinzuzufügen oder zu löschen. Die Vorgehensweise wie solche Modifikationen vorgenommen werden können, wird in den folgenden Abschnitten erläutert.

## 3.2.2 Direkte Element-Konstruktor

XML-Elemente und Attribute können in der Queryergebnisse eingefügt werden. Der direkte Element-Konstruktor definiert das XML-Element (optional mit Attribut) indem XML-Syntax verwendet wird. Beispiel 10.2, „Konstruieren von XML-Elemente mit XML-Syntax“ zeigt die Ergebnisse als ein XHTML (kürz. Extensible HyperText Markup Language) Fragment. Das Fragment beinhaltet die selektierten Daten.

## Beispiel 10.2. Konstruieren von XML-Elemente mit XML-Syntax

```
(:Query:)
<html>
 <h1>Musikstücke gespielt von Fritz Kreisler, Michael
 Raucheisen</h1>
 {
 for $track in doc("Mediathek.xml")/library/Tracks/track
 where $track/Artist = "Fritz Kreisler, Michael Raucheisen"
 return Lied: {data($track/Name)}
 }
</html>
(:Ergebnis:)
<html>
 <h1>Musikstücke gespielt von Fritz Kreisler, Michael
 Raucheisen</h1>

 Lied: Kreisler: Pollchinelle
 Lied: Kreisler: La Précieuse
 (In The Style Of Couperin)
 Lied: Kreisler: Chanson Louis Xiii
 & Pavane (In The Style Of Couperin)
 Lied: Schubert/Kreisler: Rosamunde,
 D 797 - Ballet #2
 Lied: Brandl/Kreisler:
 The Old Refrain
 Lied: Glazunov/Kreisler:
 Serenade Espagnole
 Lied: Weber/Kreisler: Larghetto
 Lied: Heuberger/Kreisler:
 Midnight Bells

</html>
```

Die Elemente `h1`, `ul` und `li` treten in die Ergebnisse als XML-Elemente auf. Das `h1` Element enthält den String *Musikstücke gespielt von Fritz Kreisler, Michael Raucheisen*, der als Inhalt von `h1` erscheint.

Andererseits enthält der `ul` Element-Konstruktor ein weiterer XQuery-Ausdruck. Der XQuery-Ausdruck ist in geschweiften Klammern umschlossen (die sogenannte *enclosed expression* – engl. eingebetteter Ausdruck). Der Wert dieses Ausdrucks wird zum Inhalt des `ul`-Elements. Im Fall von Beispiel 10.2, „Konstruieren von XML-Elemente mit XML-Syntax“ wird der „umschlossene Ausdruck“ zur einer Sequenz von `li`-Elemente ausgewertet. Die `li`-Elemente tauchen anschließend in die Ergebnisse als Kinder von `ul` auf.

Der `li`-Element Konstruktor enthält eine Kombination von Stringwerte *Lied:* und eingebettete Ausdrücke `{data($track/Name)}`, die allesamt zur Atomic Values ausgewertet werden. Jeder Inhalt der außerhalb der geschweiften Klammern steht wird zu einem String ausgewertet.

Direkte Konstruktoren verwenden eine sehr an XML angelehnte Syntax. Die Tags verwenden dieselbe spitze Klammer Syntax, die Namen müssen valide XML-Namen sein, jedes Start-Tag muss ein entsprechendes End-Tag haben. Wie in XML müssen die Attribut-Namen eindeutig sein. Es gibt aber auch kleine Unterschiede im Vergleich zum XML. Es ist beispielsweise möglich `<` innerhalb der geschweiften Klammern zu nutzen.

Zusammenfassung: Der direkte Element-Konstruktor kann Strings, andere Element-Konstrukturen und eingebettete Ausdrücke enthalten.

Anwendungsfall: Element aus dem Input-Dokument modifizieren

Annahme: Es wird gewünscht, dass Elemente aus dem Input-Dokument berücksichtigt werden. Dazu sollen kleinere Modifikationen vorgenommen werden, wie z.B. Hinzufügen oder Löschen eines Kinderelement oder Attribut. Um das zu bewältigen, muss ein neues Element mit Hilfe des Konstruktors erzeugt werden. Zum Beispiel es wird gewünscht die `track`-Elemente aus dem Input-Dokument beizubehalten. Zusätzlich wird gewünscht, dass ein neues Attribut `id` erzeugt wird. Der Inhalt dieses Attributs soll gleich dem Inhalt des `TrackID`-Elements sein.

### Beispiel 10.3. Hinzufügen eines Attributs zu einem Elementen (Ergebnis unvollständig aus Platzgründen)

```
(:Query:)
for $track in doc("Mediathek.xml")/library/Tracks/track
where $track/TrackID >1030 and $track/TrackID < 1034
return <track id="{ $track/TrackID }">
 { $track/(@*,*) }
</track>
(:Ergebnis:)
<track id="1031">
 <TrackID>1031</TrackID>
 <Name>Violin Concerto In E Minor, Op. 64:
 I. Allegro Molto Appassionato</Name>
</track>
<track id="1032">
 <TrackID>1032</TrackID>
 <Name>Violin Concerto In E Minor, Op. 64: II. Andante</Name>
 <Artist>Fritz Kreisler</Artist>
</track>
<track id="1033">
 <TrackID>1033</TrackID>
 <Name>Violin Concerto In E Minor, Op. 64: III.
 Allegretto Non Troppo - Allegro Molto Vivace</Name>
</track>
```

Lösung: Die Query erstellt eine neue Kopie des `track`-Elements. Das neue Element enthält der eingebetete Ausdruck `{ $track/(@*,*) }`, um alle Attribute und Kinder-Elemente aus dem originalen `track`-Element zu kopieren. Es kann auch der erweiterte Ausdruck `{ $track/(@*,node()) }` verwendet werden, um alle Kinderknoten des Elements zu kopieren. Dazu zählen auch Textknoten, Kommentarknoten und Processing Instruktionen.

Ein weiterer Anwendungsfall wäre, wenn nur einige Elemente aus dem Input-Dokument relevant sind (siehe Beispiel 10.4, „Entfernen von Kinder-Elemente“). Andere Elemente können nicht berücksichtigt werden. Um die Vorgehensweise besser zu erläutern ist auf der eingebetete Ausdruck `{ $track/(@*, Name, Artist, Composer) }` zu achten. Der Ausdruck selektiert alle Attribute und die Elemente `Name`, `Artist` und `Composer`, die Kinder des `track`-Elementen sind. Eine weitere Möglichkeit bestimmte Elemente bzw. Attribute auszulassen ist, indem der Ausdruck `{ $track/(@*, except Name, Artist, Composer) }` verwendet wird. In diesem Fall werden alle Attribute und alle Elemente außer `Name`, `Artist` und `Composer` selektiert.

### Beispiel 10.4. Entfernen von Kinder-Elemente

```
(:Query:)
for $track in doc("Mediathek.xml")/library/Tracks/track
where $track/TrackID >1030 and $track/TrackID < 1034
return <track id="{ $track/TrackID}">
 { $track/(@*, Name, Artist, Composer) }
</track>
(:Ergebnis:)
<track id="1031">
 <Name>Violin Concerto In E Minor, Op. 64:
 I. Allegro Molto Appassionato</Name>
 <Artist>Fritz Kreisler</Artist>
 <Composer>Felix Mendelssohn</Composer>
</track>
<track id="1032">
 <Name>Violin Concerto In E Minor, Op. 64:
 II. Andante</Name>
 <Artist>Fritz Kreisler</Artist>
 <Composer>Felix Mendelssohn</Composer>
</track>
<track id="1033">
 <Name>Violin Concerto In E Minor, Op. 64:
 III. Allegretto Non Troppo - Allegro Molto Vivace</Name>
 <Artist>Fritz Kreisler</Artist>
 <Composer>Felix Mendelssohn</Composer>
</track>
```

## 3.2.3 Computed Constructors

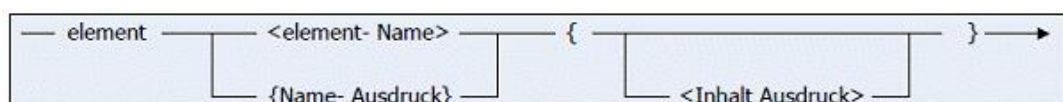
Im Allgemeinen wenn die Element-, und Attributnamen, die in das Ergebnis verwendet werden sollen, bekannt sind, kann die XML-ähnliche Syntax verwendet werden. Dieses Vorgehen wurde im „3.2.2 Direkte Element-Konstruktoren“ im Detail erklärt. Allerdings soll es möglich ein Element-, und Attributnamen dynamisch zu bestimmen. In diesem Fall kann die ehe deklarative Konstruktion, die so genannten "computed constructors" verwendet werden. Sie können nützlich sein wenn:

- Elemente aus dem Input-Dokument kopiert werden sollen, wobei kleine Änderungen dessen Inhalt vorgenommen werden. Zum Beispiel wenn ein id Attribut zu jedem Elementen hinzugefügt werden soll, oder wenn die Elemente von einem zu einem anderen Namensraum überführt werden sollen.
- der Inhalt eines Elements aus dem Input-Dokument als Name für ein neues Element oder Attribut dienen soll. Zum Beispiel ein Element soll erzeugt werden, dessen Name der Inhalt des Attributs type<sup>1</sup> ist.

Computed Konstruktoren können für Elemente, Attribute und andere Knotenarten verwendet werden.

Ein computed Konstruktor verwendet das Schlüsselwort `element`, das von den Namen des Elements und den Inhalt gefolgt ist. Die Syntax des Konstruktors ist Abbildung 10.1, „Computed Konstruktor Syntax“ dargestellt.

### Abbildung 10.1. Computed Konstruktor Syntax



Beispiel 10.5, „Einfacher computed Konstruktor“ zeigt eine Query, die mit der Query aus Beispiel 10.2, „Konstruieren von XML-Elemente mit XML-Syntax“ gleiche Ergebnisse liefert. Der Unterschied liegt in der Tatsache, dass in diesem Fall die Elemente mit Hilfe des computed Konstruktors erzeugt sind.

### Beispiel 10.5. Einfacher computed Konstruktor

```
element h1 {"Musikstücke gespielt von Fritz Kreisler,
Michael Raucheisen"},
element ul {
 for $track in doc("Mediathek.xml")/library/Tracks/track
 where $track/Artist = "Fritz Kreisler, Michael Raucheisen"
 return element li { "Lied:", data($track/Name)}
}
```

Der Elementname in einem computed Konstruktor wird, entweder durch einem Qualified Name, oder durch einen Ausdruck (in geschweiften Klammer) repräsentiert. Der Ausdruck soll ebenso zu einem Qualified Name auswerten. Der Name des Elements ist von dem Inhalt des Elements gefolgt. Der Inhalt ist durch geschweifte Klammer umschlossen. Zum Beispiel der Konstruktor

```
element h1 {"Musikstücke gespielt von Fritz Kreisler, Michael
Raucheisen"}
```

verwendet Strings um den Element `<h1>Musikstücke gespielt von Fritz Kreisler, Michael Raucheisen</h1>` zu bilden. Andererseits kann der Elementname dynamisch generiert werden:

```
let $number := 1
return
element {concat("h", $number)} {"Musikstücke gespielt von Fritz
Kreisler, Michael Raucheisen"}
```

Das nächste Teil des computed Konstruktor ist ein eingebeteter Ausdruck, der den Inhalt des Elements enthält (siehe Abbildung 10.1, „Computed Konstruktor Syntax“). Wie mit den direkten XML-Konstrukturen, werden alle Elemente, die von der eingebeteten Ausdruck zurückgeliefert sind, Kinder des neuen Elements. Des weiteren werden Attribute im eingebeteten Ausdruck zu Attributen ausgewertet. Es existiert eine Regel, dass Attribute vor Zeichenketten auftreten sollen [W07-2].

Die Syntax der computed Konstrukturen ist unterschiedlich im Vergleich zur Syntax der direkten Konstrukturen. In einem computed Konstruktor darf ausschließlich ein Paar geschweifte Klammer auftauchen. Z.B mit einem direkten Konstruktor kann das `li` Element folgendermaßen konstruiert werden:

```
for $a in doc("Mediathek.xml")//track
return
ID: {data($a/TrackID)}, Name: {data($a/Name)}
```

Hier wird der Text mit der Ausdruck innerhalb der geschweiften Klammern zusammen geschrieben. Für die Konstruktion desselben Elements mit Hilfe einer computed Konstruktor ist folgender Quellcodeauszug nötig:

```
for $a in doc("Mediathek.xml")//track
return
element li {"ID: ", data($a/TrackID), ", Name: ", data($a/Name)}
```

Hier ist zu beachten, dass der Text in Anführungszeichen steht. Der folgende Ausdruck (innerhalb der geschweiften Klammern) ist mit dem Text durch Komma getrennt. Der Ausdruck `data($a/TrackID)` ist selber nicht in geschweifte Klammer, wie es beim direkten Konstruktor der Fall ist.

## 3.3 Verbinden und Restrukturieren von Knoten (FLWOR)

In diesem Unterkapitel werden die Möglichkeiten erläutert, wie die Daten in XQuery selektiert, gefiltert und verbunden werden können. Hier wird die Syntax der FLWOR-Ausdrücke (`for`, `let`, `where`, `order by`, `return`) im Detail erklärt.

Im Kapitel 9, *XPath* wurde beschrieben, wie mit Hilfe von Pfadausdrücken XML-Elemente aus dem Input-Dokument selektiert werden können. Zum Beispiel der Ausdruck

```
doc("Mediathek.xml")/library/Tracks/track[TrackID = "1031"]/Name
```

kann verwendet werden, um den Namen des Musikstücks mit *TrackID 1031* zu bestimmen. Es können weitere Prädikate (die Ausdrücke in den eckigen Klammern) verwendet werden, um die Ergebnisse anhand mehrere Kriterien zu filtern. Logische Ausdrücke, sowie andere Ausdrücke, sind als Prädikat erlaubt:

```
(1)doc("Mediathek.xml")/library/Tracks/track[TrackID = "1031" or
TrackID = "1032"]/Name
```

Es gibt keine Voraussetzung, dass in jeder Query ein FLWOR-Ausdruck existieren soll. Pfadausdrücke sind hilfreich wenn sie in Abfragen verwendet werden, wo keine neue Elemente erzeugt werden und die Ergebnisse nicht sortiert werden sollen. Pfadausdrücke sind in diesem Fall vorzuziehen, da sie kompakter sind. Zusätzlich werden die Pfadausdrücke bei manchen Implementierungen schneller ausgewertet [W07-3].

FLWOR-Ausdrücke werden in Abfragen verwendet, die komplexer sind. Mit Hilfe der Ausdrücke ist es möglich Daten aus mehreren Quellen zu verbinden, neue Elemente und Attribute zu erzeugen, Funktionen auf Zwischenwerte anzuwenden und auszuwerten. Nicht an letzter Stelle ist auch zu erwähnen, dass die Ergebnisse zusätzlich sortiert werden können.

FLWOR ist die Abkürzung von „for, let, where, order by, return“ – die Schlüsselwörter, die im Ausdruck verwendet werden. Die FLWOR-Ausdrücke sind unter den mächtigsten und häufigsten Ausdrücken in XQuery. Sie sind ähnlich mit der `SELECT-FROM-WHERE` Anweisung in SQL. Allerdings sind die FLWOR-Ausdrücke nicht bezüglich Tabellen, Zeilen oder Spalten definiert. In den FLWORs werden in den `for` und `let` Klauseln Variablen an Werte gebunden. Die Variablen werden im Nachhinein verwendet um Ergebnisse zu erzeugen. Die Kombination von Variablenbindungen, die in den `for` und `let` Klausel erzeugt wurden, wird Tupel genannt. Beispiel 10.6, „FLWOR“ zeigt ein FLWOR-Ausdruck, der mit (1) äquivalent ist.

### Beispiel 10.6. FLWOR

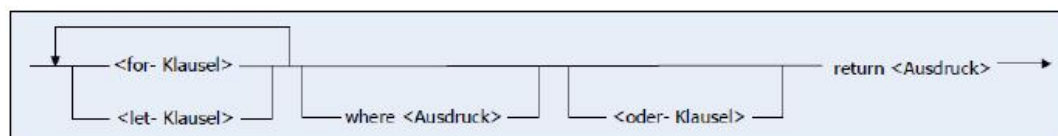
```
for $a in doc("Mediathek.xml")/library/Tracks/track
let $track := $a/Name
where $a/TrackID = "1031" or $a/TrackID = "1032"
return $track
```

Beim Vergleich der Pfadausdruck aus (1) und die Abfrage aus Kapitel 9, *XPath* fällt auf, dass für einfachere Abfragen der Pfadausdruck vorzuziehen ist. Der FLWOR-Ausdruck soll mehr als eine Illustration dienen, bevor mit komplexeren Abfragen angefangen wird. Wie aus Kapitel 9, *XPath* zu entnehmen, ist ein FLWOR-Ausdruck aus den folgenden Teilen aufgebaut:

- **for**: Mittels einer **for** Klausel werden eine oder mehrere Variablen an Ausdrücke gebunden und erzeugen so einen Strom von Tupeln.
- **let**: Durch eine **let** Klausel werden eine oder mehrere Variable jeweils an das gesamte Resultat eines Ausdrucks gebunden.
- **where**: Das **where** Konstrukt dient zur Eliminierung unerwünschter Tupel. Sinnvollerweise nimmt sein boolescher Ausdruck Bezug auf mindestens eine der in den **for** und **let** Konstrukten gebundenen Variablen.
- **order by**: Das **order by** Konstrukt dient zum Sortieren der Tupel
- **return**: Im **return** Konstrukt werden schließlich die Variablen von Interesse zurückgegeben, möglicherweise eingebettet in direkt oder indirekt konstruierte Elemente.

Die Syntax eines FLWOR-Ausdrucks ist in Beispiel 10.7, „Syntax der FLWOR- Ausdruck“ dargestellt. Die Syntax der **for** und **let** Klausel wird später im Detail erklärt.

### Beispiel 10.7. Syntax der FLWOR- Ausdruck



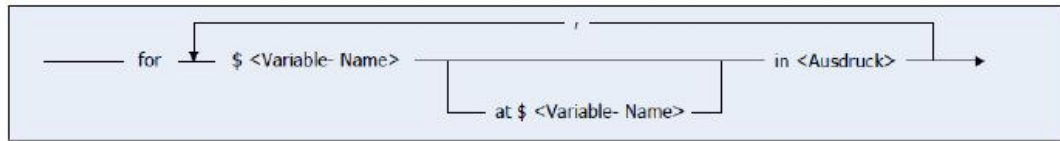
Es können mehrere **for** und **let** Klausel in beliebiger Reihenfolge existieren, die optional von einer **where** Klausel gefolgt sind, gefolgt von einer optionalen **order by** Klausel, gefolgt von der erforderlichen **return** Klausel. Ein FLWOR-Ausdruck enthält mindestens ein **for** oder ein **let** Klausel.

Die gesamte Query kann ein FLWOR sein. FLWOR können allerdings zusätzlich in andere Ausdrücke angewendet werden. Z.B. in der **return** Klausel oder innerhalb einer Funktion.

## 3.3.1 For und Let Klausel

Die Syntax der **for** Klausel ist in Beispiel 10.8, „Syntax der **for** Klausel“ dargestellt. Die **for** Klausel erzeugt eine Iteration, die ermöglicht, dass der Rest der FLWOR mehrmals ausgewertet wird. Der Rest der FLWOR wird einmal für jeden Item der Sequenz, die vom Ausdruck nach dem Schlüsselwort **in** zurückgegeben wird, ausgeführt. Die Sequenz (binding sequence) kann eine Sequenz von keinen, einen oder mehreren Items sein. Wenn die Sequenz eine leere Sequenz ist, so wird der Rest der FLWOR-Ausdruck nicht ausgewertet (iteriert Null mal).

### Beispiel 10.8. Syntax der for Klausel



Jede Klausel in einem FLWOR-Ausdruck ist definiert in Bezug auf Tupel. Die `for` und die `let` Klausel erzeugen diese Tupel. Deshalb ist es erforderlich, dass jeder FLWOR mindestens eine `for` oder eine `let` Klausel enthält. Es ist sehr wichtig zu verstehen, wie Tupel in FLWOR-Ausdrücke generiert werden. Aus diesem Grund werden ein paar Beispielabfragen gezeigt:

### Beispiel 10.9. Query mit for Klausel

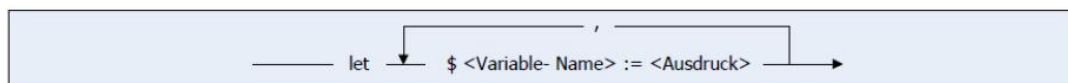
```
(:Query:)
for $i in (1,2,3)
return
<Tupel><i>{$i}</i></Tupel>
(:Ergebnis:)
<Tupel><i>1</i></Tupel>
<Tupel><i>2</i></Tupel>
<Tupel><i>3</i></Tupel>
```

Im Beispiel 10.9, „Query mit for Klausel“ wird die Variable `$i` zur Sequenz (1, 2, 3) gebunden. XQuery hat eine allgemeine Syntax, was dazu führt, dass die `for` und `let` Klausel zum jedem beliebigen XQuery-Ausdruck gebunden werden können.

Aus Beispiel 10.9, „Query mit for Klausel“ ist ebenfalls zu sehen, dass die Anordnung der ausgewerteten Items mit der Anordnung aus dem Ausdruck identisch ist. Die `for` Klausel behält die Reihenfolge, wenn es Tupel erzeugt.

Die `let` Klausel ist eine bequeme Möglichkeit eine Variable an einem Wert zu binden. Im Gegensatz zur `for` Klausel, resultiert die `let` Klausel keine Iteration. Es wird die ganze Sequenz an die Variable gebunden und nicht ein Item pro Iteration. Die Syntax der `let` Klausel ist in Beispiel 10.10, „Syntax der let Klausel“ dargestellt.

### Beispiel 10.10. Syntax der let Klausel



Beispiel 10.11, „Query mit let Klausel“ ist mit dem vorherigen Beispiel ähnlich. Hier wird die (1, 2, 3) Sequenz mit einem `let` statt `for` an die Variable `$i` gebunden. Das Ergebnis der Query enthält nur ein Tupel. In diesem Tupel ist die Variable `$i` an der ganzen Integer-Sequenz gebunden.

### Beispiel 10.11. Query mit let Klausel

```
(:Query:)
let $i := (1,2,3)
return
<Tupel><i>{$i}</i></Tupel>
(:Ergebnis:)
<Tupel><i>1 2 3</i></Tupel>
```



Beispiel 10.12, „Query mit for und let Klausel“ zeigt eine `let` Klausel in einem FLWOR-Ausdruck mit eine oder mehrere `for` Klausel. Die Variablenbindungen der `let` Klauseln werden zu den erzeugten Tupeln der `for` Klauseln hinzugefügt.

### Beispiel 10.12. Query mit for und let Klausel

```
(:Query:)
for $i in (1, 2, 3)
let $j := (1, 2, 3)
return
<Tupel><i>{ $i }</i><j>{ $j }</j></Tupel>
(:Ergebnis:)
<Tupel><i>1</i><j>1 2 3</j></Tupel>
<Tupel><i>2</i><j>1 2 3</j></Tupel>
<Tupel><i>3</i><j>1 2 3</j></Tupel>
```

Die Query aus Beispiel 10.13, „Query mit for und let Klausel1“ kombiniert die `for` und `let` Klausel auf derselbe Art und Weise, wie in Beispiel 10.12, „Query mit for und let Klausel“. Die Query liefert den Namen und die Anzahl Musikstücke aller Playlists.

### Beispiel 10.13. Query mit for und let Klausel1

```
(:Query:)
for $a in doc("PlaylistMediathek.xml")/library//Playlist
let $b := $a//TrackID
return <playlist>{$a/Name, <count>{count($b)}</count>}</playlist>
(:Ergebnis (Auszug) :)
<playlist>
 <Name>Mediathek</Name><count>714</count>
</playlist>
<playlist>
 <Name>Bach Partita E</Name><count>71</count>
</playlist>
<playlist>
 <Name>Bach Partita E Gavotte en rondeau</Name>
 <count>11</count>
</playlist>
```

Es ist möglich mehrere Variablen in einer `for` Klausel zu definieren. In diesem Fall enthalten die Ergebnistupel alle möglichen Item-Kombinationen (siehe Beispiel 10.14, „For Klausel: mehrere Variablen“). Die Kombination aller möglichen Kombinationen wird kartesischer Produkt genannt.

### Beispiel 10.14. For Klausel: mehrere Variablen

```
(:Query:)
for $i in (1, 2, 3),
 $j in (4, 5)
return
<Tupel><i>{ $i }</i><j>{ $j }</j></Tupel>
(:Ergebnis:)
<Tupel><i>1</i><j>4</j></Tupel>
<Tupel><i>1</i><j>5</j></Tupel>
<Tupel><i>2</i><j>4</j></Tupel>
<Tupel><i>2</i><j>5</j></Tupel>
<Tupel><i>3</i><j>4</j></Tupel>
<Tupel><i>3</i><j>5</j></Tupel>
```

## 3.3.2 where Klausel

Die where Klausel ist optional. Sie wird verwendet um Tupel wegzulassen, die eine bestimmte Bedingung nicht erfüllen. Die Query aus Beispiel 10.15, „Query mit where Klausel“ liefert ausschließlich die Playlisten zurück, die mehr als 40 Musikstücke enthalten.

### Beispiel 10.15. Query mit where Klausel

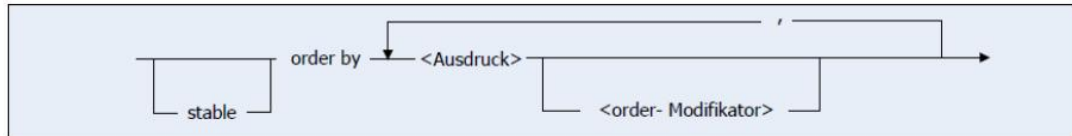
```
(:Query:)
for $a in doc("PlaylistMediathek.xml")/library//Playlist
let $b := $a//TrackID
where count($b) > 40
return <playlist>{$a/Name, <count>{count($b)}</count>}</playlist>
(:Ergebnis:)
<playlist>
 <Name>Mediathek</Name><count>714</count>
</playlist>
<playlist>
 <Name>Bach Partita E</Name><count>71</count>
</playlist>
<playlist>
 <Name>Kreisler</Name><count>75</count>
</playlist>
<playlist>
 <Name>Musik</Name><count>714</count>
</playlist>
<playlist>
 <Name>Musik der 90er</Name><count>57</count>
</playlist>
```

In der where Klausel kann jeder beliebiger Ausdruck stehen, der zu einem Boolean Wert ausgewertet wird. In ein FLWOR kann maximal eine where Klausel existieren. Allerdings kann die where Klausel von mehreren Ausdrücken gebildet werden. Die einzelnen Ausdrücke werden mit and oder or verknüpft.

### 3.3.3 Order by Klausel

Die `order by` Klausel ist optional und kann verwendet werden, um die generierten Tupeln zu sortieren. Die Syntax der `order by` Klausel ist in Beispiel 10.16, „Syntax der `order by` Klausel“ dargestellt.

#### Beispiel 10.16. Syntax der `order by` Klausel



Um die `order by` Klausel besser zu beschreiben ist auf Beispiel 10.17, „Order by Klausel“ verwiesen. Das Beispiel gibt alle Komponisten alphabetisch sortiert aus. Die `for` Klausel generiert eine Tupelsequenz mit einem `Composer` Knoten in jedes Tupel. `Order by` sortiert diese Tupel anhand des Wertes des `Composer`-Element. Die `return` Klausel liefert die sortierten Tupel zurück.

#### Beispiel 10.17. Order by Klausel

```
for $a in doc("Mediathek.xml")//track
order by $a/Composer
return $a/Composer
```

In XQuery ist mit Hilfe der `order by` Klausel möglich, auch nach Werte zu sortieren, die nicht in das Ergebnis einfließen. Das ist mit SQL und relationale Datenbanken nicht der Fall. Beispiel 10.18, „Order by Klausel 2“ zeigt, wie nach `Artist` sortiert wird, obwohl nur der Name des Musikstücks ausgegeben wird.

#### Beispiel 10.18. Order by Klausel 2

```
for $a in doc("Mediathek.xml")//track
order by $a/Artist
return $a/Name
```

Es gibt eine Reihe von Modifikatoren, die optional verwendet werden können:

- `ascending` und `descending`: Spezifizieren die Sortierrichtung. Voreingestellt ist `ascending`.
- `empty greatest` und `empty least`: Spezifizieren, wie leere Sequenzen sortiert werden.
- `collation`: Spezifiziert eine Kollation, die für die Sortierung von Strings verwendet wird. Weitere Details für Kollationen können bei Bedarf nachgeschlagen werden – [W07-4].

### 3.3.4 Return Klausel

Die `return` Klausel enthält den Ausdruck, der zurückgegeben wird. Der Ausdruck wird einmal pro Iteration ausgewertet. Hier ist angenommen, dass der Ausdruck in der optionalen `where` Klausel zu `True` ausgewertet wurde. Das Ergebnis des gesamten FLWOR ist eine Sequenz von Items, die bei jeder Auswertung der `return` Klausel zurückgegeben wurden. Z.B das Ergebnis des FLWORS:

```
for $i in (1, 2, 3)
return <Auswertung>{$i}</Auswertung>
```

ist eine Sequenz von 3 Auswertung-Elemente. Ein Element für jede Auswertung der return Klausel.

Wenn es mehrere Ausdrücke in der return Klausel gibt, können die Ausdrücke in einer Sequenz kombiniert werden z.B.:

```
for $i in (1, 2, 3)
return (<Auswertung>{$i}</Auswertung>, (<Auswertung_2>{$i}
</Auswertung_2>)
```

Das Ergebnis dieser Query ist eine Sequenz von 6 Elemente – 2 für jede Evaluation der return Klausel. Die Klammer und das Komma werden verwendet, um hinzudeuten, dass eine Sequenz von zwei Elementen zurückgegeben werden soll.

In den beiden Abfragen ist zu sehen, dass ein Element-Konstruktor verwendet wurde. Element innerhalb der return Klausel zu bilden, ist eine übliche Technik, um das Ergebnis zu formatieren, um die Hierarchie des Dokuments zu ändern oder eine neue Struktur zu definieren.

## 3.3.5 Joins

Einer der größten Vorteile der FLWOR-Ausdrücke ist die Möglichkeit Daten aus mehreren Datenquellen zu verbinden. Angenommen es sollen Informationen aus "Mediathek.xml" und "PlaylistMediathek.xml" verbunden werden. Beispielsweise sollen die Namen der Musikstücke, die von *Fritz Kreisler* gespielt wurden, und die Namen der Playlists, wo diese Stücke eingetragen sind, ausgegeben werden. Beispiel 10.19, „Join“ zeigt eine Query, die dieses Join ausführt.

### Beispiel 10.19. Join

```
(:Query:)
for $a in doc("Mediathek.xml")/library/Tracks/track,
$b in doc("PlaylistMediathek.xml")/library/Playlists/Playlist
let $name := $a/Name
let $playlist := $b/Name
where $a/TrackID = $b/PlaylistItems/TrackID
and $a/Artist = "Fritz Kreisler"
return (<LiedName>{data($name)}</LiedName>,
 <PlaylistName>{data($playlist)}</PlaylistName>)
(:Ergebnis:) (:Auszug:)
<LiedName>Cavatina</LiedName>
<PlaylistName>Mediathek</PlaylistName>
<LiedName>Cavatina</LiedName>
<PlaylistName>Kreisler</PlaylistName>
<LiedName>Cavatina</LiedName>
<PlaylistName>Musik</PlaylistName>
<LiedName>Melodie (Gluck)</LiedName>
<PlaylistName>Mediathek</PlaylistName>
<LiedName>Melodie (Gluck)</LiedName>
<PlaylistName>Musik</PlaylistName>
```

Das erste Teil der `for` Klausel selektiert alle Musikstücke. Das zweite Teil selektiert alle Playlisten. In der `where` Klausel wird überprüft, ob das jeweilige `TrackID` aus dem ersten Dokument in einer `Playlist` (den zweiten Dokument) vorkommt. Weiterhin wird nur nach Musikstücke gesucht, die von *Fritz Kreisler* gespielt wurden. Ist die Bedingung erfüllt, so werden der Name des Musikstücks (aus dem ersten Dokument) und der Name des Playlists (aus dem zweiten Dokument) ausgegeben. Aus dem Ergebnisauszug ist z.B. zu entnehmen, dass das Musikstück *Cavatina* in den Playlisten *Mediathek*, *Kreisler* und *Musik* eingetragen ist. Das Musikstück *Melodie (Gluck)* ist in den Playlisten *Mediathek* und *Musik* vorhanden.

Folgende Query (Beispiel 10.20, „Join mit Prädikat“) liefert exakt das gleiche Ergebnis. Hier wird ein Prädikat genutzt, um die übereinstimmende `TrackIDs` zu selektieren. In diesem Fall entfällt die Filterbedingung in der `where` Klausel.

### Beispiel 10.20. Join mit Prädikat

```
(:Query:)
for $a in doc("Mediathek.xml")/library/Tracks/track
[Artist = "Fritz Kreisler"],
$b in doc("PlaylistMediathek.xml")/library/Playlists/Playlist
[../TrackID = $a/TrackID]
let $name := $a/Name
let $playlist := $b/Name
return (<LiedName>{data($name)}</LiedName>,
 <PlaylistName>{data($playlist)}</PlaylistName>)
```

Das Verwenden eines Prädikats oder die Nutzung der `where` Klausel ist dem Abfrageentwickler überlassen. Wenn es mehrere Bedingungen gibt, ist eine Query mit `where` Klausel empfehlenswert, da die Query lesbarer wird. Wenn es wenige Bedingungen gibt ist die Nutzung von Prädikate zu bevorzugen, da manche Implementierungen bessere Performanz erzielen [W07-5]. Eine Kombination beider Techniken ist ebenfalls zulässig.

Das Verbinden von mehreren Quellen ist nicht auf zwei Dokumente begrenzt. Es können beliebig viele Datenquellen (Dokumente, collections) verbunden werden. Beide Schreibweisen sind identisch:

```
for $a in doc("erstesDokument.xml"), $b in doc("zweitesDokument.xml"),
$c in doc("drittesDokument.xml")
```

und

```
for $a in doc("erstesDokument.xml")
for $b in doc("zweitesDokument.xml")
for $c in doc("drittesDokument.xml")
```

Die Join Beispiele (Beispiel 10.19, „Join“ und Beispiel 10.20, „Join mit Prädikat“) sind bekannt als `inner joins`. Es werden nur die Musikstücke von *Fritz Kreisler* und die dazugehörigen Playlisten ausgegeben.

Angenommen es sollen alle Musikstücke ausgegeben werden und nur für die Musikstücke, die von *Fritz Kreisler* gespielt wurden, soll auch der Playlistname zurückgegeben werden. In den relationalen Datenbanken ist dieser Join als `outer join` bekannt. Die Query aus Beispiel 10.21, „Outer Join“ führt dieses `outer join` aus.

### Beispiel 10.21. Outer Join

```
(:Query:)
for $a in doc("Mediathek.xml")/library/Tracks/track
let $name := $a/Name
return (<LiedName>{data($name)}</LiedName>,
<PlaylistName>
 {
 for $b in doc("PlaylistMediathek.xml")/library/
 Playlists/Playlist
 where $a/TrackID = $b/PlaylistItems/TrackID
 and $a/Artist = "Fritz Kreisler"
 return $b/Name
 }
</PlaylistName>)
```

Es wurden zwei FLWORS verwendet. Der zweite FLWOR ist in der return Klausel des ersten FLWOR eingebettet. Das äußere FLWOR liefert alle Musikstücknamen. Das innere FLWOR selektiert den Playlistnamen nur für Musikstücke, die von *Fritz Kreisler* gespielt sind.

## 3.4 Operatoren und Bedingte Ausdrücke

Die Anfragen, die bis jetzt gezeigt wurden, enthalten Operatoren. Ähnlich wie bei anderen Programmiersprachen unterstützt XQuery Arithmetische- und Vergleichs-operatoren. Da Sequenzen von Knoten ein wesentlichen Datentyp in XQuery sind, ist es nötig, dass auch Sequenzoperatoren unterstützt werden. In diesem Kapitel werden die Operatoren detaillierter erläutert. Insbesondere wird erklärt, wie XQuery mit einigen Situationen umgeht. Z.B. wie soll der Ausdruck  $1 * \$a$  interpretiert werden. Wie sieht das Ergebnis aus, wenn  $\$a$  eine leere Sequenz ist, oder ein String, oder eine Sequenz mit 3 Knoten?

Zwei grundlegende Operationen spielen eine wichtige Rolle, wenn Operatoren und Funktionen verwendet werden. Die erste Operation ist das Extrahieren des typisierten Wertes (engl. typed value extraction). Diese Operation wurde bereits in den meisten Beispielqueries verwendet.

```
doc("Mediathek.xml")//track[Artist = "Fritz Kreisler"]
```

Im Prädikat stehenden *Artist* ist ein Element. *Fritz Kreisler* ist ein String. Wie können ein Element und ein String gleich sein? Die Antwort auf diese Frage ist simpel. Der `=` Operator extrahiert den typisierten Wert des Elements und vergleicht ihn mit dem angegebenen String. Wenn das XML-Dokument mit einer DTD verbunden ist, dann ist der Typ des Elements gleich mit dem in der DTD definierten Typ.

Das Extrahieren des typisierten Wertes ist definiert für ein einziges Item. Das allgemeine Vorgehen wird Atomization genannt. Atomization definiert, wie das Extrahieren des typisierten Werts für eine Sequenz von Items ausgeführt wird. Auf die folgende Query, wird Atomization angewandt:

```
let $a := (3, <e>5</e>, <s type="xs:integer">7</s>)
return
avg($a)
```

Atomization liefert den typisierten Wert jedes Items aus der Sequenz. Die Abfrage liefert als Ergebnis 5 zurück, da der Durchschnitt von 3, 5 und 7 berechnet wird. Atomization wird von den Operanden bei arithmetische und vergleich Operationen verwendet.

## 3.4.1 Arithmetische Operatoren

XQuery unterstützt die arithmetischen Operatoren `+`, `-`, `*`, `div`, `idiv` und `mod`. Der `div` Operator führt eine Division mit jedem beliebigen numerischen Typ aus. Der `idiv` Operator verlangt integer Argumente und liefert ein Integer als Ergebnis zurück. Alle anderen arithmetischen Operatoren haben ihre konventionelle Bedeutung.

Wenn ein Operand ein Knoten ist, dann wird Atomization (siehe „1.4.2 Atomic Values“) angewandt. Die folgende Query liefert als Ergebnis die Ganzzahl 3 zurück:

```
1 + <int>{2}</int>
```

Wenn ein Operand eine leere Sequenz ist, dann ist das Ergebnis ebenfalls eine leere Sequenz. Die leeren Sequenzen in XQuery funktionieren meistens wie `NULL` in SQL. Das Ergebnis dieser Query ist eine leere Sequenz:

```
1 + ()
```

Wenn ein Operand untypisiert ist, wird der Operand in `double` umgewandelt. Wenn die Umwandlung scheitert, wird eine Ausnahme ausgeworfen. Diese implizite Umwandlung ist wichtig, da es viele XML-Daten gibt, die keine einfachen oder komplexen Typen haben. Diese Daten werden als numerische Daten interpretiert. Die Größe der Musikstücke aus Beispiel 1 (Element `Size`) ist so ein Beispiel. Die folgende Query berechnet die gesamte Speichergröße der Musikstücke. Dabei werden nur die unterschiedlichen Größen summiert (siehe Beispiel 10.22, „Implizite Typumwandlung“). Das Ergebnis ist vom Typ `double`.

### Beispiel 10.22. Implizite Typumwandlung

```
let $size := doc("Mediathek.xml")//Size
return sum(distinct-values($size))
```

## 3.4.2 Vergleichsoperatoren

XQuery unterstützt Wertevergleich, generelles Vergleich, Knotenvergleich und Vergleich der Knotenreihenfolge. Operatoren für Wertevergleich und generelles Vergleich stehen in einer engen Beziehung zueinander. Für jeden Wertevergleich-Operator existiert ein korrespondierendes allgemeines Vergleichsoperator (Siehe Tabelle 10.1, „Operatoren für Wertevergleich bzw. generelles Vergleich“).

**Tabelle 10.1. Operatoren für Wertevergleich bzw. generelles Vergleich**

Operator für Wertevergleich	Operatoren für generelles Vergleich
<code>Eq</code>	<code>=</code>
<code>Ne</code>	<code>!=</code>
<code>Lt</code>	<code>&lt;</code>
<code>Le</code>	<code>&lt;=</code>
<code>Gt</code>	<code>&gt;</code>
<code>Ge</code>	<code>&gt;=</code>

Der Wertevergleich vergleicht zwei Atomic Values. Wenn ein Operand ein Knoten ist, wird Atomization angewandt. Wenn ein Operand untypisiert ist, wird es in String umgewandelt, und als

solches behandelt. Es folgt ein Beispiel (Beispiel 10.23, „Der eq Operator“), der den eq Operator verwendet.

### Beispiel 10.23. Der eq Operator

```
for $a in doc("Mediathek.xml")//track
where $a//Composer eq "Fritz Kreisler"
return $a//Artist
```

Mit dem Wertevergleich-Operator können ausschließlich Strings mit Strings verglichen werden. Wenn die Daten, die genutzt werden, in keiner DTD (kürz. Document Type Definition) deklariert sind, so sind die Daten untypisiert. Deswegen muss z.B. TrackID in einem decimal umgewandelt werden (cast) (siehe Beispiel 10.24, „Expliziter cast bei untyped Values“).

### Beispiel 10.24. Expliziter cast bei untyped Values

```
for $a in doc("Mediathek.xml")//track
where xs:decimal($a//TrackID) gt 1000
return $a//Name
```

Wenn die Datentypen in eine DTD definiert wurden, so wird die Umwandlung nicht gebraucht.

Wie arithmetische Operatoren, behandelt der Wertevergleich die leeren Sequenzen wie eine SQL NULL. Wenn ein Operand eine leere Sequenz ist, wird der Vergleich zu einer leeren Sequenz ausgewertet.

Wenn ein Operand mehrere Items beinhaltet, wird eine Ausnahme ausgeworfen.

Knotenvergleiche ermitteln, ob zwei Ausdrücke zum selben Knoten auswerten. Es gibt zwei Operatoren für Knotenvergleich in XQuery: `is` und `is not`. Jeder Operand soll ein einzelnen Knoten oder eine leere Sequenz sein. Wenn ein Operand eine leere Sequenz ist, ist das Ergebnis ebenfalls eine leere Sequenz.

Der `is` Operator vergleicht die Knoten auf Basis deren Identität (siehe „1.4.1.3 Knotenidentität“) und nicht auf Basis deren Inhalts. Um den Inhalt und Attributwerte von zwei Knoten zu vergleichen, soll die vordefinierte Funktion `deep-equal` verwendet werden [W07-6]. Tabelle 10.2, „Knotenvergleich“ zeigt Beispiele von Knotenvergleiche (Knoten `$a` und `$b` sind unterschiedlich).

### Tabelle 10.2. Knotenvergleich

Beispiel	Wert
<code>\$a is \$b</code>	False
<code>\$a is \$a</code>	True
<code>doc("Mediathek.xml")//track[1] is doc("Mediathek.xml")//track[./TrackID = 464]</code>	True
<code>doc("Mediathek.xml")//track[1]//@type is doc("Mediathek.xml")//track[2]//@type<sup>a</sup></code>	False

<sup>a</sup>Obwohl der type Attribut beim ersten und zweiten Musikstück gleich *aac* ist, liefert der Operator `is` als Ergebnis *false* zurück

XQuery stellt zwei Operatoren bereit, mit deren Hilfe ermittelt werden kann, ob ein Knoten vor oder nach einem anderen Knoten im Dokument auftritt. Die zwei Operatoren spielen eine große Rolle wenn, die Anordnung der Elemente wichtig ist. Das ist der Fall insbesondere bei Tabellen. Der Operator `$a << $b` liefert *true* zurück, wenn das Element `$a` vor dem Element `$b` im Dokument auftritt. Der Operator `$a >> $b` liefert *true* zurück, wenn das Element `$a` nach dem Element `$b` erscheint.



Die Query aus Beispiel 10.25, „Operator für Reihenfolgevergleich“ liefert alle Musikstücke zurück, deren `Artist` und `Composer` Elemente vorhanden sind und zusätzlich das Element `Artist` vor dem Element `Composer` auftritt.

### Beispiel 10.25. Operator für Reihenfolgevergleich

```
for $track in doc("Mediathek.xml")//track
let $artist := $track/Artist,
$composer := $track/Composer
where $artist << $composer
return $track
```

## 3.4.3 Sequenz Operatoren

XQuery unterstützt die Operatoren `union`, `intersect` und `except`, um Knotensequenzen zu kombinieren. Jeder dieser Operatoren kombiniert zwei Sequenzen, wobei das Ergebnis eine Sequenz ist. Wenn ein Operand ein Item enthält, der kein Knoten ist, wird eine Ausnahme ausgeworfen.

Der `union` Operator (`|` oder `union`) erwartet zwei Sequenzen als Operanden. Das Ergebnis ist eine Sequenz, die alle Knoten aus den beiden Eingabesequenzen enthält.

### Beispiel 10.26. Der union Operator

```
let $a := distinct-values(doc("Mediathek.xml")//(Artist|Composer))
return <union>{count($a)}</union>
```

Der `intersect` Operator erwartet zwei Sequenzen als Operanden. Das Ergebnis ist eine Sequenz, die alle Knoten beinhaltet, die in den beiden Eingabesequenzen auftritt.

Der `except` Operator liefert eine Sequenz mit Knoten, die in der ersten Sequenz auftreten und nicht in der zweiten Sequenz (siehe Beispiel 10.27, „Der except Operator“).

### Beispiel 10.27. Der except Operator

```
for $a in doc("Mediathek.xml")//track
where $a//Artist = "Fritz Kreisler"
return <track>{$a/* except $a/Size}</track>
```

Dieses Beispiel liefert alle Elemente zurück, außer das `Size` Element.

## 3.4.4 Bedingte Ausdrücke

Die bedingten Ausdrücke in XQuery werden ähnlich wie bedingte Ausdrücke in anderen Programmiersprachen verwendet. Die Schlüsselwörter für einen bedingten Ausdruck sind `if`, `then` und `else`. Die Syntax des bedingten Ausdrucks ist in Abbildung 10.2, „bedingter Ausdruck“ dargestellt.

### Abbildung 10.2. bedingter Ausdruck

————— if (<Ausdruck>) then <Ausdruck> else <Ausdruck> —————>

Der Ausdruck (sog. Test-Ausdruck) nach dem Schlüsselwort `if`, soll in Klammern geschrieben werden. Wenn der Ausdruck zu `true` ausgewertet wird, ist der Wert des bedingten Ausdruck, der

Wert aus der `then` Klausel. Ansonsten ist das Ergebnis des bedingten Ausdrucks, der Wert in der `else` Klausel. Die `else` Klausel soll immer angegeben werden. Es ist allerdings erlaubt als Ausdruck die leere Sequenz anzugeben (`else()`). Beispiel 10.28, „Bedingter Ausdruck innerhalb eines FLWORS“ zeigt ein bedingter Ausdruck, der in ein FLWOR eingebettet ist.

### Beispiel 10.28. Bedingter Ausdruck innerhalb eines FLWORS

```
for $a in doc("Mediathek.xml")//track
return if($a/Composer = "Fritz Kreisler")
then $a/Name
else ()
```

Wenn die `then` und `else` Klausel aus mehreren Ausdrücken bestehen, sollen die Ausdrücke in Klammern umschlossen sein. Bedingte Ausdrücke können auch geschachtelt sein. In diesem Fall wird das Schlüsselwort `else if` verwendet.

## 3.5 Funktionen

Funktionen sind eine sehr nützliche Eigenschaft von XQuery. Durch die Funktionen kann eine erweiterte Funktionalität erzielt werden. Mit Hilfe der Funktionen lassen sich XQuery-Module bilden, oder ein Teil der Query kann mehrmals verwendet werden. Es existieren zwei Arten von Funktionen: eingebaute Funktionen (engl. *build-in functions*) und benutzerdefinierte Funktionen. Die eingebauten XQuery Funktionen sind spezifiziert und werden von allen Implementierungen unterstützt. Die benutzerdefinierten Funktionen werden durch den Queryautor definiert. Sie können innerhalb einer Query sein, oder in einer externen Bibliothek.

### 3.5.1 Built-In Funktionen

XQuery besitzt eine große Anzahl von built-in Funktionen. Die gesamte Liste dieser Funktionen kann im Anhang Anhang C, *Appendix: Built-In Funktionen* nachgeschlagen werden. Eine Mehrzahl dieser Funktionen sind aus anderen Programmiersprachen bereits bekannt. In diesem Kapitel wird ausschließlich auf die meist bekannten Funktionen fokussiert, damit der rote Faden zum nächsten Kapitel nicht verloren geht.

SQL Programmierer kennen bestimmt die Funktionen `min()`, `max()`, `avg()`, `count()` und `sum()`. Die Query im Beispiel 10.29, „Built-in Funktionen“ liefert den Namen der Playlisten, die zwischen 10 und 20 Musikstücke enthalten.

### Beispiel 10.29. Built-in Funktionen

```
(:Query:)
for $b in doc("PlaylistMediathek.xml")/library/Playlists/Playlist
let $count := count($b/PlaylistItems/TrackID)
return $b[$count > 10 and $count < 20]/Name
(:Ergebnis:)
<Name>Bach Partita E Gavotte en rondeau</Name>
<Name>Bach Partita E Loure</Name>
<Name>Contzen: Saint-Saens, Debussy, Frank</Name>
<Name>Harlekin und Colombine</Name>
<Name>Party-Jukebox</Name>
<Name>Short Stories</Name>
<Name>Szerying: Kreisler</Name>
<Name>Zeit: Casals</Name>
```

Wenn der Typ des Funktionsarguments ein Atomic Typ ist, dann sollen folgende Regel beachtet werden:

1. wenn das Argument ein Knoten ist, dann wird sein typisierter Wert (typed value) extrahiert. Das ergibt eine Sequenz von Werten.
2. wenn ein Wert aus der Argumentsequenz nicht typisiert ist, so versucht XQuery diesen Wert im verlangten Typ zu konvertieren. Scheitert die Konvertierung, so wird eine Ausnahme ausgeworfen. Ein Wert wird akzeptiert, wenn es den erwarteten Typ hat.

Andere bekannte XQuery Funktionen sind z.B.:

- numerische Funktionen wie `round()`, `floor()` und `ceiling()`
- String Funktionen wie `concat()`, `string-length()`, `starts-with()`, `ends-with()`, `substring()`, `upper-case()`, `lower-case()`, etc.
- Cast Funktionen für die diversen generischen Typen

XQuery hat auch Funktionen, die nicht aus anderen Programmiersprachen bekannt sind. Die Eingabefunktionen `doc()` und `collection()`, und die Funktionen `not()` und `empty()` werden am meisten verwendet. Die `empty()` Funktion testet, ob eine Sequenz leer ist. Das Gegenteil der `empty()` Funktion ist die `exists()` Funktion. In diesem Fall wird getestet, ob eine Sequenz mindestens ein Item enthält.

In XQuery gibt es Funktionen, die auf mit einem Knoten assoziierten Informationen zugreifen. Die meist bekannten "Zugriffs"-Funktionen sind `string()` und `data()`. Die `string()` Funktion liefert den Stringwert des Knotens. Die `data()` Funktion liefert den typisierten Wert des Knotens. Beide Funktionen sollen weiter unter der Lupe genommen werden. Der Stringwert eines Knotens enthält die Stringrepräsentation des Textes, der in dem Knoten und in aller seiner Kinder steht. Der resultierende Text wird konkateniert ausgegeben. Die Reihenfolge wird beibehalten. Beispiel 10.30, „Die Funktion string“ illustriert die Nutzung der `string` Funktion.

### Beispiel 10.30. Die Funktion string

```
(:Query:)
string(doc("PlaylistMediathek.xml"))/Playlist[30]/PlaylistItems
(:Ergebnis:)
534535536537538539540541542543544545546547548549550551552553554
555556557558559560
```

## 3.5.2 Benutzerdefinierte Funktionen

XQuery bietet dem Benutzer die Möglichkeit Funktionen selber zu erstellen. Das macht möglich, Query-Fragmente wieder zu verwenden und / oder Bibliotheken zu entwickeln. Durch das Verwenden von benutzerdefinierte Funktionen kann erreicht werden, dass die Query lesbarer wird, indem Ausdrücke abgetrennt und benannt werden können. Einige gute Beispiele können auf [www.xquerfunctions.com](http://www.xquerfunctions.com) nachgechlagen werden.

Es gibt einige gute Gründe warum benutzerdefinierte Funktionen sinnvoll sind:

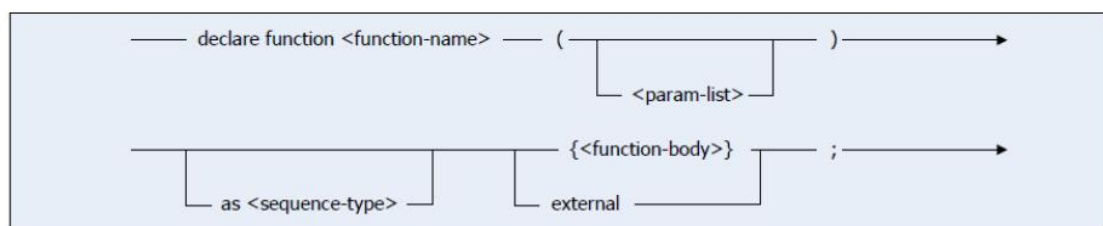
- Wiederverwendung: Wenn ein Ausdruck mehrmals ausgewertet wird, scheint es sinnvoll zu sein, den Ausdruck separat in einer Funktion zu definieren. Die Funktion kann von mehreren Stellen aufgerufen werden. Die Nutzung von Funktionen erhöht die Wartbarkeit der Query, da der Algorithmus nur an einer Stelle bei Bedarf geändert werden soll.

- Übersichtlichkeit: Funktionen machen die Query lesbarer. Eine benannte Funktion mit definierten Parametern kann als ein Template für die Dokumentation dienen. Auf diese Weise wird die Funktion physisch vom Rest der Query getrennt. Vor allem bei komplexere, geschachtelte Abfragen kann der Überblick behalten werden.

Weitere Gründe warum die Definition von benutzerdefinierte Funktionen sinnvoll ist, sind Rekursion, Typumwandlung, etc.

Um eine Funktion aufrufen zu können, soll sie erstmals deklariert werden. Die Deklaration kann im Prolog der Query oder in eine externe Bibliothek gemacht werden. Die Syntax der Funktionsdeklaration ist in Abbildung 10.3, „Syntax der Funktionsdeklaration“ dargestellt. Die Parameterliste ist optional. Nicht desto trotz müssen die Klammer immer angegeben werden. Der Ausgabetyt ist ebenfalls optional.

### Abbildung 10.3. Syntax der Funktionsdeklaration



### Beispiel 10.31. Funktionsdeklaration

```
(:Funktion:)
declare function local:albums (
 $album as xs:string?,
 $trackID as xs:decimal?
)
{
 for $a in doc("Mediathek.xml")/library/Tracks/track
 let $album := data($a/Album)
 let $trackID := data($a/TrackID)
 return
 (<Album>{$album}</Album>, <TrackID>{$trackID}</TrackID>)
};
```

Beispiel 10.31, „Funktionsdeklaration“ zeigt die Deklaration der Funktion `local:albums` innerhalb der Query-prolog. Die Funktion akzeptiert zwei Argumente – `album` und `trackID`. Die Funktionsdeklaration ist zusammengesetzt aus:

- Die Schlüsselwörter `declare function` gefolgt vom Funktionsnamen
- Eine Parameterliste umschlossen in Klammern. Die Parameter sind voneinander mit Komma getrennt.
- Der Returntyp der Funktion
- Das Funktionsbody – in geschweifte Klammer gefolgt von Semikolon.

Der Funktionsbody ist ein Ausdruck, der in geschweifte Klammer umschlossen ist. Es kann jeder gültigen XQuery Ausdruck enthalten – FLWOR, Pfadausdruck oder ein anderer XQuery Ausdruck. Die `return` Klausel ist nicht obligatorisch. Wenn die `return` Klausel nicht definiert ist, ist der

Rückgabewert der Funktion der Wert des Ausdrucks. Eine Funktion kann auch folgendermaßen deklariert sein:

```
declare function local:printTest(){ "Test" };
```

Innerhalb des Funktionsbody können weitere Funktionen aufgerufen werden. Die Funktionen können im selben Modul oder in einem importierten Modul sein. Die Reihenfolge der Deklaration ist irrelevant.

Jede Funktion ist eindeutig mit ihren Namen und Parameteranzahl definiert. Es können mehrere Funktionen mit demselben Namen existieren, jedoch muss die Parameteranzahl unterschiedlich sein. Der Funktionsname soll ein gültiger XML-Name sein.

Alle benutzerdefinierte Funktionen müssen in einem Namensraum deklariert werden. In dem Hauptquerymodul kann jedes Präfix verwendet werden, der im Prolog definiert ist. Weiterhin kann das vordefinierte Präfix `local`<sup>2</sup> verwendet werden. Innerhalb der Hauptquerymodul wird die Funktion mit dem Präfix `local` aufgerufen.

In Bezug auf den Funktionsnamen ist ebenfalls zu beachten, dass es Funktionsnamen gibt, die reserviert sind. Die reservierten Funktionsnamen sind in Tabelle 10.3, „Reservierte Funktionsnamen“ aufgeführt. Es ist nicht untersagt Funktionen mit den aufgeführten Namen zu deklarieren. Jedoch müssen diese Funktionen mit einem Präfix aufgerufen werden. Um mehr Übersichtlichkeit zu erzielen, sollen diese Funktionsnamen vermieden werden.

**Tabelle 10.3. Reservierte Funktionsnamen**

Attribute	if	schema-attribute
Comment	item	schema-element
document-node	node	processing-instruction
Element	text	typeswitch
empty-sequence		

Funktionen können sich selber rekursiv aufrufen. Beispielsweise sollen alle Nachfolger eines Elements gezählt werden. Es soll nicht nur die Anzahl der direkten Kinder, sondern die Anzahl aller Nachfolger ermittelt werden. Das kann mit Hilfe der Funktion aus Beispiel 10.32, „Rekursive Funktion“ erledigt werden.

### Beispiel 10.32. Rekursive Funktion

```
declare function local:num-descendant-elements
($el as element()) as xs:integer {
 sum(for $child in $el/*
 return local:num-descendant-elements($child)+1)
```

Die Funktion `local:num-descendant-elements` ruft sich rekursiv selber auf, um zu ermitteln wie viele Kinder das Element hat, wie viele Kinder die Kinder des Elementes haben, usw. Allerdings muss sichergestellt werden, dass die Funktion terminiert. In diesem Fall terminiert die Funktion, sobald ein Element erreicht wird, der keine Kinder hat. Dann wird die `return` Klausel nicht mehr ausgewertet. Die Funktion

---

<sup>2</sup>Namensraum siehe [BCFFRS07]

```
declare function local:infinite () {1+ local:infinite()};
```

würde nie terminieren.

## 3.6 Implementierungen

In diesem Kapitel werden zwei Implementierungen vorgestellt, die XQuery verwenden. Zunächst wird kurz der Saxon XSLT- und XQuery Prozessor erläutert. Anschließend wird auf die native XML-Datenbank eXist eingegangen. Es wird mit Hilfe von einige Beispiele erklärt, wie XML-Daten in/aus einer Datenbank gespeichert, abgefragt und modifiziert werden können.

### 3.6.1 Saxon

Der Saxon XSLT- und XQuery Prozessor [K09-1] ist von Michael Kay entwickelt worden. Die aktuelle Version ist 9.1.0.6. Es sind zwei Implementierungen vorhanden: für Java und .NET. Die Software gibt es sowohl als kommerzielle, als auch als quelloffene Variante. Saxon-SA (kommerziell) ist im Unterschied zu Saxon-B (kostenlos) in der Lage Dokumente schemasensitiv zu verarbeiten.

Der Laufzeitquellcode in Saxon, der XQuery und XSLT unterstützt, ist identisch. Das liegt an der Tatsache, dass beide Sprachen eine sehr ähnliche Semantik haben. Die XQuery Unterstützung von Saxon besteht hauptsächlich aus einem XQuery Parser. Der Parser generiert denselben interpretierbaren Kode wie den XSLT Prozessor.

Der XQuery Prozessor kann entweder über die Kommandozeile oder über eine benutzer-entwickelte Anwendung (durch den bereitgestellten API) aufgerufen werden. Es existiert keine graphische Benutzeroberfläche.

Saxon ist entwickelt worden, um Eingabedokumente zu verarbeiten, die im Arbeitsspeicher reinpassen. Saxon ist erfolgreich verwendet worden, um Eingabedokumente größer als 100MB zu verarbeiten. Hier soll beachtet werden, dass genügend Speicher für die Java VM zugeordnet werden soll. Es soll weiterhin beachtet werden, dass bei komplexeren FLWOR Ausdrücke die Ausführungszeit enorm ansteigen kann [K09-2].

Aus der Kommandozeile kann beispielsweise das folgende Kommando genutzt werden (.NET Implementierung):

```
D:\studium_master\MA_data\bin>Query.exe -o test.xml
"d:\studium_master\MA_data\example3.xq"
```

Die `-o` Option definiert das Ausgabedokument. Mit Eingabe von `Query -?` können die weiteren Optionen angezeigt werden. Die Aufrufsyntax für die .NET Variante ist folgendermaßen definiert:

```
Query [options] -q:queryfile [params...]
```

[params...] ist in der Form `variable=wert`. Wobei `variable` der Name der Variable ist und `wert` ist der Wert, der an der Variable gebunden wird. Auf die Parameter kann innerhalb der Query als externe Variablen zugegriffen werden:

```
declare $variable as xs:string external
```

Im Allgemeinen ist Saxon eine stabile und effiziente Software zur Transformation von XML-Dokumente.

### 3.6.2 eXist

eXist [eXDB01] ist eine native XML-Datenbank, die kostenlos als Open Source vertrieben wird. eXist läuft innerhalb einer Java Virtual Machine und ist damit komplett unabhängig vom verwendeten Betriebssystem. Native XML-Datenbanken speichern die Information, ähnlich wie bei der Speicherung von XML-Dokumenten im Dateisystem, direkt als XML-Dokumente ab. Sie werden daher auch als dokumentenorientiert bezeichnet. Im Gegensatz zu relationale Datenbank Management Systeme verwendet eXist XQuery, um die Daten abzufragen.

Um die Vorstellung der eXist Datenbank zu erleichtern, soll zunächst erläutert werden, was hinter dem Begriff "native XML-Datenbank" steckt.

In Prinzip die so genannten nativen XML-Datenbanken sind keine selbständige Datenbanken. Sie speichern das XML nicht wirklich in seinem echten nativen Form (z.B. Text). Eine native XML-Datenbank (laut [S01]):

- definiert ein logisches Modell für ein XML-Dokument, im Gegensatz zu den Daten in diesem Dokument. Die Datenbank speichert und ruft die Dokumente entsprechend dieses Modells ab. Das Modell soll mindestens Elemente, Attribute, PCDATA und Dokumentanordnung abbilden können. Beispiele für solche Modelle sind das XPath Datenmodell, das XML Infoset und die Modelle abgeleitet vom DOM (kürz. Document Object Model) und die Ereignisse in SAX (kürz. Simple API for XML).
- hat als zugrunde liegende (logische) Speichereinheit ein XML-Dokument, wie eine relationale Datenbank eine Zeile in einer Tabelle als zugrunde liegende (logische) Speichereinheit hat.
- soll nicht unbedingt ein bestimmtes zugrunde liegendes physisches Speichermodell haben. Z.B die Datenbank kann auf Basis einer relationalen, hierarchischen, oder objektorientierten Datenbank aufgebaut sein. Es kann ein proprietäres Speicherformat verwenden: indexierte oder komprimierte Dateien.

eXist speichert die Daten in Collections und Dokumente. Die Datenbank kann virtuell beliebig viele Collections und Dokumente beinhalten. Es existieren vier Funktionen, mit denen das Eingabe-Dokument (oder Collection) spezifiziert werden kann. `doc()` und `collection()` sind die Standard XQuery/Xpath-Funktionen, die verwendet werden können. Es existieren zusätzlich noch zwei eXist spezifische Funktionen `xmldb:document()` und `xmldb:xcollection()`. Unterschiede können unter [eXDB02] nachgeschlagen werden.

Ohne URI-Schema interpretiert eXist die Argumente von `doc()` und `collection()` als absolute oder relative Pfade, die zu einer Collection oder ein Dokument aus der Datenbank führen. Z.B. `doc("/db/coll1/coll2/datei.xml")` referenziert eine Ressource, die in `/db/coll1/coll2` gespeichert ist.

`doc("resource.xml")` referenziert eine Ressource relativ zum Basis-URI, die in der statischen XQuery Kontext definiert ist. Die Basis-URI ist für die verschiedenen Schnittstellen (z.B. XML:DB API, REST) unterschiedlich, deshalb ist es vorteilhaft auf das Dokument durch einen expliziten Pfad zuzugreifen.

Die XQuery/XPath Funktionsbibliothek enthält die gebräuchlichsten Funktionen für die Manipulation von Zeichenketten. Jedoch sind diese Funktionen nicht ausreichend, wenn eine Suche in einem großen Textabschnitt nach Schlüsselwörter oder Phrasen ausgeführt werden soll. Die Schwachstelle tritt auf, wenn mit dokumentorientierten Daten gearbeitet werden soll.

eXist bietet einige neue Operatoren und Funktionen, die die Volltextsuche (engl. fulltext search) ermöglichen.

eXist bietet Funktionen um den Datenbankinhalt zu manipulieren [eXDB03] sowie auch Utilityfunktionen wie z.B md5.

Die Basisfunktionen, die eine Datenbank erfüllen soll, sind:

- Anlegen der gewünschten Tabellen (in relationale DB), Collections
- Speichern von neue Daten
- Abfragen der abgelegten Daten
- Manipulieren der Daten

#### 1. Anlegen von Collections

Die gespeicherten XML-Daten sind in Collections strukturiert. Durch die Funktion `create-collection` ist die Möglichkeit gegeben, die gewünschte Collection-Struktur aufzubauen. Das Erzeugen einer neuen Collection ist in Beispiel 10.33, „Anlegen einer neuen Collection“ dargestellt.

### Beispiel 10.33. Anlegen einer neuen Collection

```
let $collection := 'xmldb:exist:///db'
let $collection-name := 'Collection-Name'
return
xmldb:create-collection($collection, $collection-name)
```

Die default Basiscollection in der eXist XML-Datenbank ist die `db Collection`. Sie dient als Root-Verzeichnis, auf dem weitergebaut werden kann. Beispiel 10.33, „Anlegen einer neuen Collection“ legt eine neue Collection `Collection-Name` unterhalb der `db Collection`.

#### 2. Speichern von Daten

Mit dem nächsten Beispiel wird vorgestellt, wie die bereits in der XML-Datenbank gespeicherte "Mediathek.xml" auf vier weitere XML-Dokumente zerlegt wird. Die vier neue Dokumente (Tracks.xml, Albums.xml, Interpreters.xml und Composers.xml) werden anschließend in der XML-Datenbank gespeichert (siehe Beispiel 10.34, „Speichern von XML-Dokumente in der Datenbank“).



### Beispiel 10.34. Speichern von XML-Dokumente in der Datenbank

```
xquery version "1.0";
declare namespace xmldb="http://exist-db.org/xquery/xmldb";
declare function local:importTracks ()
{
 <Tracks>
 {
 for $a in doc("xmldb:exist:///db/Library/Mediathek.xml")
 //track
 return
 <Track>
 <TrackID>{data($a/TrackID)}</TrackID>
 <Title>{data($a/Name)}</Title>
 <Year>{data($a/Year)}</Year>
 <TrackMetaInformation>
 <BitRate>{data($a/BitRate)}</BitRate>
 <Size>{data($a/Size)}</Size>
 <SampleRate>{data($a/SampleRate)}</SampleRate>
 <DateAdded>{data($a/DateAdded)}</DateAdded>
 <DateModified>{data($a/DateModified)}</DateModified>
 <Location>{data($a/Location)}</Location>
 </TrackMetaInformation>
 </Track>
 }
</Tracks>
};
declare function local:importInterpreters ()
{
 <Interpreters>
 {
 for $a in doc("xmldb:exist:///db/Library/Mediathek.xml")
 //track
 let $test := "("
 let $test2 := ")"
 return
 <Interpreter>
 <name>{substring-before(data(data($a/Artist)), $test)}
 </name>
 <dateOfBirth>{substring-before(substring-after
 (data($a/Artist), $test), $test2)}</dateOfBirth>
 <TrackID>{data($a/TrackID)}</TrackID>
 </Interpreter>
 }
</Interpreters>
};
declare function local:importAlbums ()
{
 <Albums>
 {
```

```
for $a in doc("xmldb:exist:///db/Library/Mediathek.xml")//track
return
```

```
<Album>
 <AlbumName>{data($a/Album)}</AlbumName>
 <TrackID>{data($a/TrackID)}</TrackID>
</Album>
}
</Albums>
};
declare function local:importComposers ()
{
 <Composers>
 {
 for $a in doc("xmldb:exist:///db/Library/Mediathek.xml")
 //track
 let $test := "("
 let $test2 := ")"
 return
 <Composer>
 <name>{substring-before(data($a/Composer), $test)}
 </name>
 <dateOfBirth>{substring-before(substring-after
 (data($a/Composer), $test), $test2)}</dateOfBirth>
 <TrackID>{data($a/TrackID)}</TrackID>
 </Composer>
 }
</Composers>
};
let $collection := 'xmldb:exist:///db/Library'
let $login := xmldb:login($collection, 'admin', '')
let $tracks := 'Tracks.xml'
let $interpreters := 'Interpreters.xml'
let $albums := 'Albums.xml'
let $composers := 'Composers.xml'
let $importTracks := local:importTracks()
let $importInterpreters := local:importInterpreters()
let $importAlbums := local:importAlbums()
let $importComposers := local:importComposers()
return
let $store-return-status :=
xmldb:store($collection, $tracks, $importTracks)
let $store-return-status2 :=
xmldb:store($collection, $albums, $importAlbums)
let $store-return-status3 :=
xmldb:store($collection, $composers, $importComposers)
let $store-return-status4 :=
xmldb:store($collection, $interpreters, $importInterpreters)
return <message>Tracks, Interpreters,
Albums and Composers have been successfully imported </message>
```

Zunächst werden vier benutzerdefinierte Funktionen gebraucht, die die Struktur der neuen Daten definieren. Z.B. die `importTracks` Funktion liest alle für ein Musikstück relevante Informationen aus der "Mediathek.xml", speichert die einzelnen Elementinhalte in entsprechende (neue) Elemente.

Mit der `xmldb:login` Funktion wird der Zugriff auf die angegebene Collection realisiert. Da sollen die Zugangsdaten der Datenbank angegeben werden.

Mit der `xmldb:store` Funktion werden neue Ressourcen in der Datenbank gespeichert. Die Funktion erwartet drei Argumente: das erste Argument ist die Collection, in der die neue Ressource gespeichert wird, das zweite Argument definiert der Name der neuen Ressource und das dritte Argument repräsentiert die Daten der neuen Ressource.

### 3. Abfragen von Daten in einer XML-Datenbank

Das Abfragen von Daten aus einer Datenbank ist identisch mit alle vorherigen Beispiele, wo die `doc / collection` Funktion verwendet wurde. Hier soll allerdings auf das Dokument / Collection folgendermaßen zugegriffen werden:

```
doc("xmldb:exist:///db/Library/Tracks.xml")/Tracks/Track
```

### 4. Manipulieren der Daten

`eXist` bietet Funktionen für das Manipulieren von Daten (`insert`, `replace`, `value`, `delete` und `rename`). Bis zum Zeitpunkt des Erstellens dieser Arbeit ist keine endgültige Spezifikation für die Update-Funktionalität von XQuery erschienen. Aus diesem Grund ist zu beachten, dass die jeweiligen Datenbank-Anbieter verschiedene Syntax verwenden.

#### a. Einfügen

```
update insert expr (into | following | preceding) exprSingle
```

Mit der oben gezeigten Syntax wird der Inhalt, der in `expr` spezifizierten Sequenz, in dem durch `exprSingle` angegebenen Elementknoten, eingefügt. Falls `exprSingle` mehrere Elementknoten beinhaltet, werden die Modifikationen auf jeden Knoten angewandt. Die Position des Einfügens ist durch die Schlüsselwörter `into`, `following` und `preceding` definiert.

- `into`: Der Inhalt wird nach dem letzten Kinderknoten des spezifizierten Elements hinzugefügt
- `following`: Der Inhalt wird gleich nach dem in `exprSingle` spezifizierte Knoten angehängt.
- `preceding`: Der Inhalt wird vor dem in `exprSingle` spezifizierte Knoten angehängt.

Beispiel 10.35, „Einfügen von neuen Knoten“ zeigt wie ein neues Musikstück in der Datenbank eingefügt wird. Zunächst wird mit der Funktion `last-node` das letzte vorhandene Musikstück ermittelt. Mit der Funktion `last-id` wird das `TrackID` des letzten Musikstücks ermittelt und mit 1 erhöht. Mit `let $trackDoc` wird definiert, wie der neue XML-Fragment strukturiert ist. Mit der Funktion `xmldb:login` wird versucht auf die Datenbank zuzugreifen. Am Ende dieses Beispiels wird das Manipulieren der Daten ausgeführt (`update insert $trackDoc following local:last-node($updateTracks/Tracks/Track)`).

### Beispiel 10.35. Einfügen von neuen Knoten<sup>3</sup>

```
xquery version "1.0";
declare namespace request="http://exist-db.org/xquery/request";
(:letzten Knoten ermitteln :)
declare function local:last-node
($nodes as node()*) as node()? {
($nodes/.)[last()]
} ;
(: letzte TrackID und inkrementieren :)
declare function local:last-id(){
for $a in doc("xmldb:exist:///db/Library/Tracks.xml")/Tracks
return local:last-node($a/Track/TrackID)+1
};
let $Title := "Title"
let $AlbumName := "AlbumName"
let $InterpreterName := "InterpreterName"
let $Composer := "Composer"
let $Year := "Year"
let $BitRate := "BitRate"
let $Size := "Size"
let $SampleRate := "SampleRate"
let $DateAdded := "DateAdded"
let $DateModified := "DateModified"
let $Location := "Location"
let $trackDoc :=
<Track>
 <TrackID>{local:last-id()}</TrackID>
 <Title>{$Title}</Title>
 <Year>{$Year}</Year>
 <TrackMetaInformation>
 <BitRate>{$BitRate}</BitRate>
 <Size>{$Size}</Size>
 <SampleRate>{$SampleRate}</SampleRate>
 <DateAdded>{$DateAdded}</DateAdded>
 <DateModified>{$DateModified}</DateModified>
 <Location>{$Location}</Location>
 </TrackMetaInformation>
</Track>
let $collection := 'xmldb:exist:///db/Library'
let $login := xmldb:login($collection, 'admin', '')
return
let $updateTracks := doc("xmldb:exist:///db/Library/Tracks.xml")
return
(:insert new track after the last track node:)
<message>
 {update insert $trackDoc following local:last-node
 ($updateTracks/Tracks/Track)}
 You have now {count($updateTracks/Tracks/Track)} tracks
</message>
```

b. Replace

update replace expr with exprSingle

Es werden die Knoten, die von `expr` zurückgegeben sind, mit den Knoten in `exprSingle` ersetzt. `expr` kann ein Element, Attribut oder ein Textknoten sein. Falls `expr` ein Element ist, soll `exprSingle` ebenfalls ein Elementknoten beinhalten. Falls `expr` ein Attribut oder ein Textknoten ist, wird sein Wert als den konkatenierten Stringwert von allen Items aus `exprSingle` gesetzt.

c. Value

```
update value expr with exprSingle
```

Ändert den Inhalt aller Knoten in `expr` mit den Items aus `exprSingle`. Falls `expr` ein Attribut oder ein Textknoten ist, wird sein Wert als den konkatenierten Stringwert von allen Items aus `exprSingle` gesetzt.

d. Delete

```
update delete expr
```

Löscht alle Knoten, die in `expr` angegeben sind.

Beispiel 10.36, „Löschen eines Musikstücks“ zeigt, wie das Musikstück mit `TrackID = 1178` gelöscht werden kann.

### Beispiel 10.36. Löschen eines Musikstücks

```
xquery version "1.0";
declare function local:delete_Track($del)
{
 for $a in doc('xmldb:exist:///db/Library/Tracks.xml')/Tracks
 /Track
 where $a/TrackID=$del
 return update delete $a
};
let $login := xmldb:login('xmldb:exist:///db
/Library', 'admin', '')
return
local:delete_Track(1178)
```

e. Rename

```
update rename expr as exprSingle
```

Benennt die Knoten spezifiziert in `expr` um, indem die neue Bezeichnung der Stringwert des ersten Items aus `exprSingle` ist.

---

## **Teil IV. Verarbeitung von XML-Daten**

---

---

# Inhaltsverzeichnis

11. XML-Dokumente mit XSLT transformieren .....	118
4.1 Orientierung .....	118
4.2 Transformation von XML-Dokumente .....	118
4.2.1 Cascading Style Sheets .....	120
4.2.2 XML Style Language Transformation .....	121
4.2.3 Paradigmen der Informatik .....	133
4.2.4 XSL Formatting Objects .....	134
12. Zusammenfassung .....	139

---

# Kapitel 11. XML-Dokumente mit XSLT transformieren

## 4.1 Orientierung

Der XML-Standard definiert eine Syntax mit der es möglich ist, Struktur und Inhalt eines Dokuments zu kodieren. Eine strikte Trennung dieser beiden Merkmale ist notwendig um das Modell der strukturierten Dokumente, welches diese "Separation of Concerns" erfordert, korrekt umsetzen zu können. Die Syntax ist primär darauf ausgelegt, möglichst einfache und effiziente automatische Verarbeitung (dazu zählen die bereits vorgestellte Anfragebearbeitung durch XQuery und die in diesem Kapitel vorgestellte Transformation mit Hilfe von XSLT) eines Dokuments zu ermöglichen. Dies wird durch die interne Baumstruktur erreicht, in der XML Dokumente abgelegt werden. Diese Struktur eignet sich zwar hervorragend für die maschinelle Weiterverarbeitung daten-orientierter Dokument ist aber genau aus diesem Grund für eine menschliche Verarbeitung, und somit für narrative Dokument, weniger geeignet (man denke nur an andere Programmiersprachen, die oft kryptisch und schwer lesbar für das menschliche Auge sind).

Als Beispiel kann man sich einmal eine XML-Datei in einem beliebigen Browser ansehen. Was man sieht, wird wahrscheinlich nicht den gestellten Erwartungen entsprechen. Der Browser zeigt, wenn überhaupt, nur den Quellcode bzw. die Baumstruktur der XML-Datei an aber von Formatierungen wie Überschriften, Absätzen o.ä. ist nichts zu sehen. Dies verdeutlicht, dass ein XML Dokument in seiner Rohform nicht für die Verarbeitung durch Menschen gedacht ist, gleichzeitig entsteht jedoch ein Bedarf nach einem Standard, der eben doch ermöglicht, ein XML Dokument auch für das menschliche Auge aufzubereiten. Solch ein Standard sollte Strukturelemente mit Formatvorgaben verknüpfen und durch Anwendung der Formatvorlage eine lesbare Version eines XML Dokuments produzieren.

Wer bereits mit HTML gearbeitet hat, ist sicher schon mit der Cascading Style Sheets-Sprache (CSS) in Berührung gekommen. CSS ist eine mögliche Lösung für die oben beschriebene Problematik. Ein sog. *Stylesheet* beschreibt Formatvorlagen für bestimmte HTML Elemente, beispielsweise, dass eine Überschrift immer rot dargestellt werden soll, und legt somit die Präsentation fest, die wiederum im Browser dargestellt werden kann. Der Browser wendet die Formatvorlage auf das HTML Dokument an und als Resultat sieht man eine entsprechend aufbereitete Webseite. Analog zu HTML können XML-Dokumente auch mit CSS kombiniert werden. Ein normaler Webbrowser kann also ein XML-Dokument mit einem CSS-Stylesheet interpretieren und entsprechend umsetzen.

Weiterhin verfügt XML aber noch über einen wesentlich mächtigeren Satellitenstandard als CSS, die sogenannte *Extensible Stylesheet Language Transformation* (XSLT). XSLT baut auf der logischen Baumstruktur eines XML-Dokumentes auf und dient zur Definition von Umwandlungsregeln. XSLT-Programme, sogenannte XSLT-Stylesheets, sind dabei selbst nach den Regeln des XML-Standards aufgebaut und ermöglichen die automatisierte Transformation von einer XML-formattierten Datei in z.B. einen XML-Dialekt wie SVG, HTML oder MathML aber auch in andere Formate wie PDF.

Dieses Kapitel soll lediglich einen detaillierten Überblick über XSLT vermitteln. Wir werden anhand einfacher, leicht nachvollziehbarer Beispiele die Syntax und den Leistungsumfang von XSLT einführen und dabei ein Verständnis für die unterschiedlichen XSLT-Programmierstile, die wir anschließend behandeln, schaffen.

## 4.2 Transformation von XML-Dokumente

Im Verlauf dieses Kapitels werden wir anhand verschiedener Beispiele die Konzepte der Transformation kennen lernen. Die Beispiele sind aus einem Konferenzverwaltungssystem entnommen.

Unser Beispiel 11.1, „Doe.xml“ enthält Registrierungsinformationen über verschiedene Konferenzteilnehmer. Zu einem Teilnehmer werden u.a. Name, Institution, Adressdaten, Anreise- und Abreisedatum sowie Zahlungsinformationen gespeichert.



## Beispiel 11.1. Doe.xml

```
<?xml version="1.0" encoding="utf-8"?>

<registration>
 <firstName>John</firstName>
 <lastName>Doe</lastName>
 <titles>Dr.</titles>
 <affiliation>Technische Universität München</affiliation>
 <address>
 <line>Department of Computer Science</line>
 <line>Boltzmannstraße 3</line>
 <line>85748 Garching b. München</line>
 <line>Germany</line>
 </address>
 <contact>
 <eMail>doe@tum.de</eMail>
 <telephone>+49 1234 5678 90</telephone>
 <fax>+49 1234 4568 91</fax>
 </contact>
 <payCash/>
 <arrive>Arrival: September 12, 2000</arrive>
 <leave>Departure: September 15, 2000</leave>
 <noCarParking/>
 <notes>
 Please confirm that payment by cheque (not credit card) at
 the conference is okay. Please tell me to whom the cheque
 should be payable.
 </notes>
</registration>
```

Nun sehen Sie sich dieses XML-Dokument in Ihrem Browserfenster an - je nach verwendetem Browser wird die Darstellung in etwa wie in Abbildung 11.1, „Doe.xml in Mozilla Firefox 3.5“ aussehen.

## Abbildung 11.1. Doe.xml in Mozilla Firefox 3.5

This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
-<registration>
 <firstName>John</firstName>
 <lastName>Doe</lastName>
 <titles>Dr.</titles>
 <affiliation>Technische Universität München</affiliation>
 -<address>
 <line>Department of Computer Science</line>
 <line>Boltzmannstraße 3</line>
 <line>85748 Garching b. München</line>
 <line>Germany</line>
 </address>
 -<contact>
 <eMail>doe@tum.de</eMail>
 <telephone>+49 1234 5678 90</telephone>
 <fax>+49 1234 4568 91</fax>
 </contact>
 <payCash/>
 <arrive>Arrival: September 12, 2000</arrive>
 <leave>Departure: September 15, 2000</leave>
 <noCarParking/>
 -<notes>
 Please confirm that payment by cheque (not credit card) at the conference is okay. Please tell me to whom the cheque should be payable.
 </notes>
</registration>
```

Wie Abbildung 11.1, „Doe.xml in Mozilla Firefox 3.5“ zu entnehmen ist, bietet die baumförmige Darstellung von unserem XML-Dokument doe.xml nicht das tatsächlich erwartete Layout an. Wünschenswert wäre z.B. eine Tabelle mit den einzelnen Daten, welche auch mit anderen Teilnehmerdaten zusammengefügt werden kann. Für diesen Zweck verfügt XML über entsprechende

Werkzeuge, sogenannte Stylesheets. Mittels Stylesheets lässt sich das Aussehen von XML-Dokumenten definieren. Es gibt zwei grundlegende Möglichkeiten um das Layout eines Dokuments festzulegen:

- Cascading Style Sheets (CSS), analog zu HTML können XML-Dokumente auch mit CSS kombiniert werden.
- Extensible Stylesheet Language (XSL), XSL deckt nicht nur die Formatierung von XML-Dokumenten ab, sondern ist mit dem Extensible Stylesheet Language Transformation (XSLT) auch für die Transformation von XML-Dokumenten in verschiedenen Ausgabeformate zuständig.

XSL schließt drei Bestandteile ein:

- XML Path Language (XPath), eine Sprache zum Adressieren (Lokalisieren) von Teilen eines XML-Dokuments (XMLBaumes) mittels Pfaden (Siehe Kapitel 9, *XPath*).
- Extensible Stylesheet Language Formatting Objects (XSL-FO) für die Beschreibung eines Dokuments als Baum mit Formatierungsanweisungen und Stilangaben.
- XML Style Language Transformation (XSLT) ist für die Transformation eines XML-Dokuments in einen anderen Baum zuständig und steht im Fokus dieses Kapitels.

Im folgenden, werde wir uns zuerst mit den Möglichkeiten von CSS beschäftigen und anschließend XSLT konkret vorstellen.

## 4.2.1 Cascading Style Sheets

Zunächst schauen wir uns einmal eine CSS-Beispiel namens "style.css" für die Formatierung von "Doe.xml" an:

### Beispiel 11.2. style.css

```
firstName, lastName, titles {background-color: #cccccc;}
affiliation {
 display: block;
 padding-top: 10px;
}
address {
 display: block;
 padding-bottom: 10px;
}
contact {
 display: block;
 padding-bottom: 10px;
}
eMail, telephone, fax {border: #0000FF 1px solid;}
arrive, leave {display: block;}
notes {
 display: block;
 padding-top: 10px;
}
```

Bevor wir uns "Doe.xml" im Browserfenster ansehen (Siehe hierzu Abbildung 11.2, „Doe.xml im Browser (mittels CSS formatiert)“), müssen wir die CSS-Datei "style.css" in unserem XML-Dokument einbinden. Dies geschieht mittels der folgenden Anweisung:

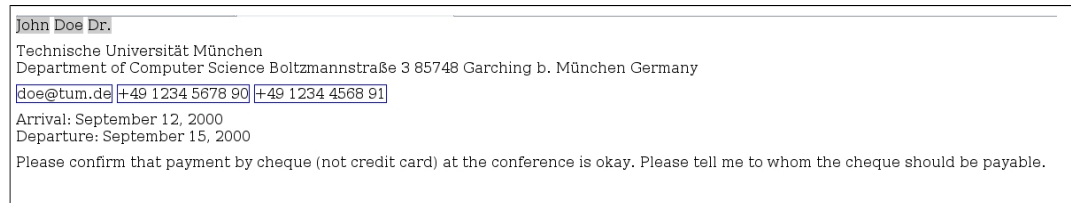
```
<?xml-stylesheet type="text/css" href="style.css"?>
```

In dem XML-Dokument folgt die Anweisung, mit deren Hilfe eine CSS-Datei eingebunden wird, direkt nach der XML-Dokument Deklaration `<?xml version="1.0" encoding="UTF-8"?`

> oder gleich im Anschluss an die Schema-Deklaration (beispielsweise Dokumenttyp-Deklaration), wenn eine vorhanden ist.

Mit Hilfe der Notation `xml:stylesheet` verweisen wir auf eine CSS-Datei, welche die Styleinformationen enthält. Die Referenz zu der CSS-Datei wird mit dem `href`-Attribut angegeben. Bei Verwendung eines CSS-Datei bekommt das `type`-Attribut den Wert `text/css`.

### Abbildung 11.2. Doe.xml im Browser (mittels CSS formatiert)



Wie der Abbildung 11.2, „Doe.xml im Browser (mittels CSS formatiert)“ zu entnehmen ist, legt die CSS-Datei die Eigenschaften der Darstellung eines XML- oder HTML-Dokuments fest, kann aber die Elemente selbst nicht beeinflussen und kann derzeit weder Daten vor der Ausgabe automatisch sortieren noch Daten bei der Anzeige filtern. In XSLT dagegen bestehen solche Möglichkeiten.

## 4.2.2 XML Style Language Transformation

Ähnlich wie bei CSS kann mithilfe von XSLT ein Stylesheet definiert werden. Eine XSLT-Datei stellt daher eine weitere Möglichkeit zur formatierten Darstellung von XML-Dokumenten in Web-Browsern und anderen Programmen dar. XSLT bietet jedoch weitaus mehr Formatierungsmöglichkeiten als CSS.

### 4.2.2.1 XSLT Steckbrief

Zusammen mit XQuery 1.0 bildet XSLT die Grundlage für die Operabilität von XML. Ohne die beiden genannten Mechanismen wäre XML stark eingeschränkt und wäre als Umsetzung für das Modell der strukturierten Dokumente nicht geeignet. Die beiden Sprachen bauen auf XDM auf und können als Analogon zu SQL gesehen werden, welches als Abfragesprache die Nutzbarkeit des relationalen Datenbankenmodells garantiert. Seit dem 23. Januar 2007 ist XSLT 2.0 als Standard durch das W3C verabschiedet worden. Das seit 1999 gültige XSLT 1.0 wurde dadurch abgelöst.

XSLT ist eine deklarative Sprache in der auf bestimmten Mustern (Templates) Anweisungen ausgeführt werden. Die Anwendung dieser Muster bzw. der Ablauf der Transformation wird im Wesentlichen durch den eingesetzten XSLT-Prozessor festgelegt.

Ein XSLT-Prozessor liest das XSLT-Stylesheet und das XML-Dokument, führt die Anweisungen des Stylesheets aus und erzeugt das Ergebnisdokument (Siehe Abbildung 11.3, „XSLT-Prozessor“). Zusätzlich wird ein XML-Parser-Aufruf ausgeführt, um die Wohlgeformtheit von Dokument und Stylesheet zu garantieren. XSLT-Prozessoren sind in XML-Entwicklungsumgebungen wie beispielsweise "oXygen" integriert und auch Browser (z.B. Internet Explorer Version 5 oder höher) können Transformationen eines XML-Dokuments mithilfe eines Stylesheets in verschiedene Ausgabeformate wie HTML oder XHTML durchführen.

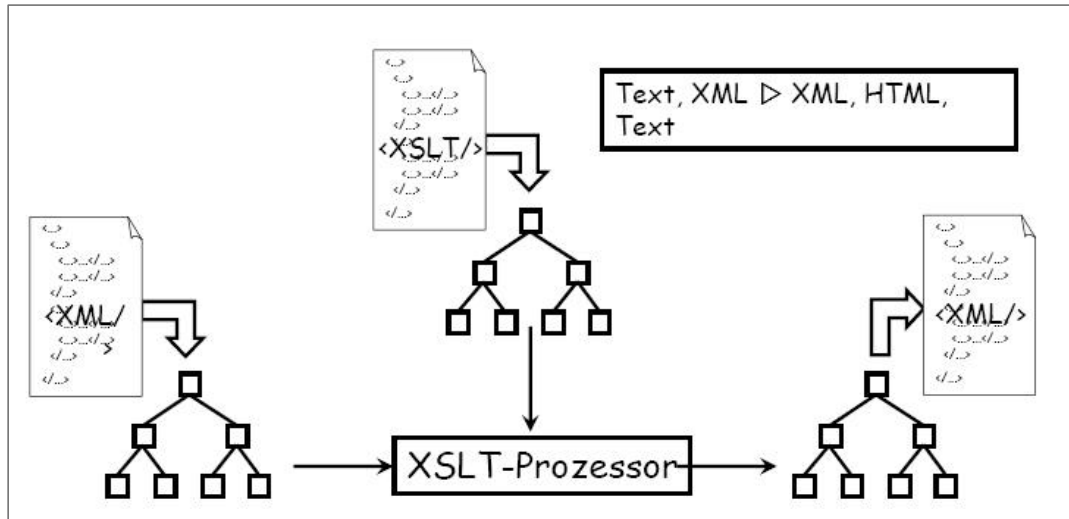
Es ist eine große Zahl verschiedener XSLT Prozessoren von diversen Herstellern verfügbar. Dazu zählen neben Open Source Projekten wie beispielsweise Xalan oder Saxon auch kommerzielle Software von Herstellern wie Microsoft.

Microsoft Internet Explorer (ab Version 5) sowie Mozilla transformieren XML-Dokumente ebenfalls, dazu muss in einer Processing Instruction auf ein entsprechendes Stylesheet verwiesen wird (Siehe „4.2.2.2 Aufbau eines XSLT-Stylesheets“).

XSLT garantiert die Cross-Media- und die Multi-Channel-Fähigkeit von XML. Das bedeutet, dass ein einziges XML Dokument unabhängig in verschiedenen Medienformen (Internet, Broschüre, Textbuch) publiziert werden kann, ohne das zugrunde liegende Dokument an sich zu ändern.

Der Drehscheibencharakter von XSLT ermöglicht es, beliebige (von XSLT-unterstützte) Ein- und Ausgabeformate miteinander zu kombinieren. So lassen sich beispielsweise Umwandlungen von XML nach MathML oder von SVG nach HTML oder von HTML in ein selbst-definiertes XML-Format vornehmen.

**Abbildung 11.3. XSLT-Prozessor**

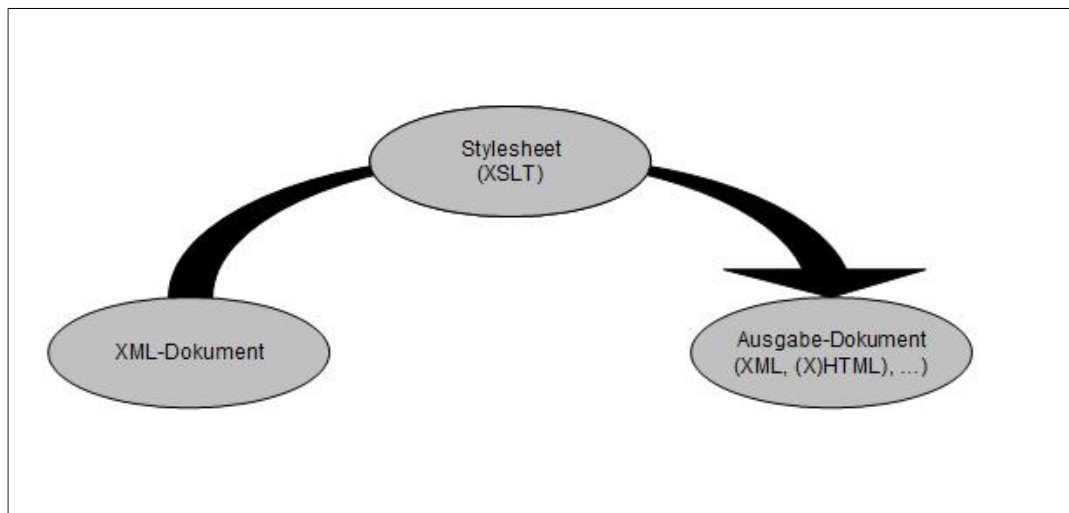


Zu den typischen Transformationen für Text- und Datendokumente zählen unter anderem:

- Umbenennungen von Textstrukturen
- Verschieben und Duplizieren von Textstrukturen
- Generieren und Unterdrücken von Textstrukturen
- Aggregation und Gruppierung von Textstrukturen
- Arithmetische Umrechnungen
- Sortieren
- Auswahl von Textstrukturen (XPath)

Abbildung 11.4, „XSLT: Eine Transformationssprache für XML-Dokumente“ zeigt das standard Prozessmodell für eine XSLT Transformation. Ein XML Input wird mittels eines XSLT Programms in ein Ausgabedokument beliebigen Formats transformiert.

**Abbildung 11.4. XSLT: Eine Transformationssprache für XML-Dokumente**



Das Paradigma der Transformation umfasst u.a. die Bereiche der Pipes und Filter, Definition von Publikationsketten, Abgeschlossenheit und Komponierbarkeit, welche für XSLT eine große Rolle spielen.

### 4.2.2.2 Aufbau eines XSLT-Stylesheets

Als einführendes Beispiel sehen wir uns einmal den Aufbau der Datei "style.xml" an, welche ein einfaches XSL stylesheet für unsere Registrierungen definiert.

#### Beispiel 11.3. style.xml

```
<?xml version="1.0" encoding="utf-8"?>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 version="2.0">
<xsl:template match="/">

<html>
 <head>
 <title>Registrations</title>
 </head>
 <body>
 <table border="1" width="80%">
 <tr>
 <th>Name</th>
 <th>Institution</th>
 <th>Address</th>
 <th>Arrival</th>
 <th>Departure</th>
 </tr>
 <xsl:for-each select="registration">
 <tr>
 <th>
 <xsl:value-of select="firstName" />
 <xsl:value-of select="lastName" />
 </th>
 <th><xsl:value-of select="affiliation" /></th>
 <th><xsl:for-each select="/registration/address/line">
 <xsl:value-of select="." /><xsl:text> </xsl:text>
 </xsl:for-each>
 </th>
 <th><xsl:value-of select="arrive" /></th>
 <th><xsl:value-of select="leave" /></th>
 </tr>
 </xsl:for-each>
 </table>
 </body>
</html>

</xsl:template>
</xsl:stylesheet>
```

Durch die erste Codezeile `<?xml version="1.0" encoding="utf-8" ?>` der XSLT-Datei "style.xml" wird deutlich, dass es sich bei einem XSLT-Stylesheet um ein XML-Dokument handelt. Ein Stylesheet wird durch ein `xsl:stylesheet`- oder `xsl:transform`-Element in einem XML-Dokument repräsentiert, dabei wird das Präfix `xsl:` verwendet, um Elemente aus dem XSLT-

Namensraum zu referenzieren. Der XSLT-Namensraum besitzt den URI: <http://www.w3.org/1999/XSL/Transform>. Weiterhin muss das `xsl:stylesheet`- bzw. das `xsl:transform`-Element ein `version`-Attribut enthalten. Das Beispiel "style.xml" besitzt die Version 2.0.

Die eigentliche Vorlage beginnt mit der dritten Zeile `<xsl:template match="/">`. Durch diese Anweisung werden alle Elemente der XML-Datei selektiert und stehen somit für die Weiterverarbeitung in der XSLT-Datei zur Verfügung. Die beiden Anweisungen `<xsl:for-each select="/.../elementname">` und `<xsl:value-of select=elementname"/>` sind auch von Bedeutung. Während `<xsl:for-each select="/.../elementname">` eine Schleifenkonstruktion zur Selektierung eines jeden einzelnen Datensatzes namens `elementname` einleitet, werden mittels `<xsl:value-of select="elementname"/>` der Inhalt des Datensatzes namens `elementname` in der Tabelle ausgegeben. Der Rest des Codes stellt nur eine Tabellenformatierung im HTML-Format dar.

Transformationsanweisungen für den Prozessor beginnen in der Regel mit `xsl`. Die weiteren Inhalte in einem XSLT-Stylesheet, wie z.B. HTML-Code, werden in der Regel unverändert in das Zieldokument übernommen.

Bevor wir uns das XML-Dokument "Doe.xml" im Browserfenster ansehen (Siehe hierzu Abbildung 11.5, „Doe.xml im Browser (mittels XSLT formatiert)“), müssen wir die XSLT-Datei "style.xml" in das XML-Dokument einbinden. Dies geschieht mittels der folgenden Anweisung:

```
<?xml-stylesheet type="text/xsl" href="style.xml"?>
```

Die formatierte Darstellung von "Doe.xml" sieht folgendermaßen aus:

**Abbildung 11.5. Doe.xml im Browser (mittels XSLT formatiert)**

Name	Institution	Address	Arrival	Departure
JohnDoe	Technische Universität München	Department of Computer Science Boltzmannstraße 3 85748 Garching b. München Germany	Arrival: September 12, 2000	Departure: September 15, 2000

Das obige Beispiel zeigt eine mögliche Transformation eines beliebigen XML Dokuments in ein durch den Browser anzeigbares HTML Dokument. Man kann hieran bereits die vielseitigen Möglichkeiten von XSLT im Vergleich zu CSS erkennen. In XSLT lassen sich z.B. die Inhalte beliebig umsortieren/anordnen und auch Text-/Strukturinformation kann hinzugefügt werden.

### 4.2.2.3 XSLT Processing Model

Wie bereits erwähnt, arbeitet XSLT auf dem Datenmodell XDM. In XDM sind Dokumente als Bäume dargestellt und alle XSLT Operationen werden auf der Baumstruktur ausgeführt. Die Baumstruktur ist auch Grundlage für XQuery und XPath. Dadurch wird erreicht, dass Daten einfach zwischen den einzelnen Standards ausgetauscht werden können. Das Baummodell ist in Kapitel 8, *Baummodell vs. XQuery 1.0 und XPath 2.0 Datenmodell (XDM)* beschrieben.

Eine XSLT Transformation benötigt ein Eingabedokument und ein Stylesheet um ein (oder auch mehrere) Ausgabedokument(e) zu erzeugen. Es gibt verschiedene Implementationen des Transformationsinterfaces, welches durch die XSLT Spezifikation beschrieben wird. Jedes dieser Implementationen muss die Bereitstellung der folgenden Übergabeparameter ermöglichen:

- Das Stylesheet

- Ein Quelldokument
- Ein Wurzeltemplate von dem alle Transformationen ausgehen
- Einen Modus
- Stylesheet Parameter
- Einen URI zur Bestimmung, wohin das Ausgabedokument geschrieben werden soll

Hierbei legt ein Modus fest, welche Art von Durchlauf die Transformation ausführen soll. Ein bekanntes Beispiel sind z.B. zwei verschiedene Durchläufe für die Erstellung eines Inhaltsverzeichnisses und für die Erstellung des eigentlichen Dokumentes. Beim Durchlauf für das Inhaltsverzeichnis sollen Überschriften anders behandelt werden als beim Erzeugen des Dokumentes. Die unterschiedliche Behandlung kann über Modi gesteuert werden.

In „4.2.2.2 Aufbau eines XSLT-Stylesheets“ wurde bereits ein XSLT Programm beispielhaft dargestellt und einige Bestandteile erklärt. Der Hauptbestandteil sind *Template Regeln*. Eine Template Regel wird mit `<xsl:template>` eingeleitet und enthält ein `match` Attribut, welches bestimmt welche Elemente im Quelldokument durch dieses Template verarbeitet werden sollen. In Beispiel 11.3, „style.xml“ wäre dies der Ausdruck

```
<xsl:template match="/">
```

welcher als Wurzeltemplate den Dokument Knoten matcht und somit den Einstieg in den Transformationsprozess bildet.

Der Template Inhalt ist eine Sequenz von Elementen und Textknoten und wird auch als *Sequenz Konstruktor* bezeichnet. Konstruktorelemente können hierbei sowohl Verarbeitungselemente als auch Datenelemente sein. In Beispiel 11.3, „style.xml“ ist z.B. der Ausdruck `<xsl:value-of select="firstName">` ein Verarbeitungselement wohingegen das `<head>` oder `<body>` Element Datenelement sind. Datenelemente werden auch *literal result elements* genannt.

## 4.2.2.4 XSLT-Programmierstile

XSLT-Programmierung unterstützt drei grundlegende Stile: den Push- und den Pull-Stil sowie Funktionales Programmieren. Im folgenden werden diese Ansätze anhand von Beispielen erläutert.

### 4.2.2.4.1 Pull-basierte Programmierung

Wir beginnen zuerst mit der Vorstellung der Pull-basierten Programmierung. "Pull" bedeutet ziehen und im Zusammenhang mit XSLT meint dies, dass das Transformationsprogramm sich die benötigten Informationen aus dem Dokument heraus zieht. Andere Bezeichnungen sind auch "Fill in the Blanks" oder "Outputsteuerung". Das Prinzip der Pull-Programmierung ist die einfachste Form der XSLT-Programmierung und eignet sich somit gut für einen Einstieg.

Beispiel 11.4, „simplePull.xml“ zeigt ein einfaches Beispiel für ein Pull-basiertes XSLT Programm

### Beispiel 11.4. simplePull.xsl

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

<xsl:transform
 version="2.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

 <xsl:output method="xml" indent="yes"
 encoding="iso-8859-1"
 xmlns:xalan="http://xml.apache.org/xslt"
 xalan:indent-amount="2"/>

 <xsl:template match="registration">
 <data>
 <badge>
 Name: <xsl:value-of select="firstName"/>
 <xsl:text> </xsl:text>
 <xsl:value-of select="lastName"/>
 <xsl:text>
 </xsl:text>
 <xsl:value-of select="affiliation"/>
 </badge>
 <address>
 <xsl:for-each select="address/line">
 <xsl:value-of select="."/>
 <xsl:text>
 </xsl:text>
 </xsl:for-each>
 </address>
 </data>
 </xsl:template>

</xsl:transform>
```

Wie in Beispiel 11.3, „style.xsl“ wird in der ersten Zeile das Programm als XML Dokument deklariert und anschließend das XSLT Programm mit Hilfe von `xsl:transform` definiert. `xsl:transform` und das in Beispiel 11.3, „style.xsl“ verwendete `xsl:stylesheet` sind hierbei vollkommen identisch. Das folgende `xsl:output` legt das Format der Ausgabedatei fest. Hierzu gibt es vier grundlegende Möglichkeiten: `xml`, `html`, `xhtml` und `text`. Durch die Angabe des Ausgabeformates wird erreicht, dass beim Erstellen des Dokumentes nach den Konventionen des Formates gearbeitet wird, d.h. wählt man `xhtml`, werden z.B. Zeilenumbrüche immer in der Form `<br />` dargestellt. Danach beginnt das eigentliche XSLT Programm.

Man kann erkennen, dass das Zieldokument (=Transformationsprogramm) ein Ergebniss-Skelett für das letztendlich ausgegebene Dokument darstellt. Die Struktur ist bereits vorgegeben, was noch fehlt, sind die Inhalte. Diese werden nun über XPath Ausdrücke aus dem Quelldokument importiert (gezogen) und in das Ausgabedokument eingefügt. Das zugehörige Ausgabedokument ist in Beispiel 11.5, „Ergebnisdokument nach Transformation von Doe.xml mittels simplePull.xsl“ dargestellt.



### Beispiel 11.5. Ergebnisdokument nach Transformation von Doe.xml mittels simplePull.xsl

```
<?xml version="1.0" encoding="iso-8859-1"?>
<data>
 <badge>
 Name: John Doe
 Technische Universität München</badge>
 <address>Department of Computer Science
 Boltzmannstraße 3
 85748 Garching b. München
 Germany
 </address>
</data>
```

Der Inhalteimport kann sowohl iteriert als auch bedingt erfolgen. Zeile 25 in Beispiel 11.4, „simplePull.xsl“ beispielsweise enthält einen iterierten Import. Mit `xsl:if` oder `xsl:choose` kann auf Bedingungen geprüft und ggf. importiert werden.

Aufgrund ihrer Einfachheit bringt die Pull-basierte Programmierung auch einige Nachteile mit sich. Diese sind:

- eingeschränkte Struktur der Ausgabedaten im Verhältnis zu Eingabedaten,
- keine gute Behandlung von mixed content und
- keine individuelle Behandlung von gemischten Elementen in einer Hierarchiestufe.

Das Ergebnisdokument ist genau ein XSLT-Template welches auf genau ein spezifisches Eingabeformat ausgerichtet ist. Dadurch entsteht eine erhebliche Einschränkung für die Wiederverwendbarkeit. Zusätzlich ist es äußerst Aufwendig, rekursive Strukturen zu erkennen und zu behandeln, da keine beliebig tiefe Schachtelung möglich ist. Die beschriebenen Nachteile können durch eine fortgeschrittene XSLT-Technik umgangen werden.

#### 4.2.2.4.2 Push-basierte Programmierung

Die Push-basierte XSLT Programmierung umgeht die oben gelisteten Probleme durch das Prinzip *mehrerer Templates*. Für jedes Element im Quelldokument wird ein Template und entsprechende Verarbeitungsvorschriften definiert. Dadurch lässt sich ein großes Template in mehrere Tempates modularisieren, welche wiederum in anderer Art und Weise neu kombiniert werden können und somit ein neues Template bilden. Das Push-Prinzip wird auch "Inputsteuerung" genannt.

In Beispiel 11.6, „simplePush.xsl“ ist das in Beispiel 11.4, „simplePull.xsl“ dargestellte Transformationsprogramm im Push Stil aufgeführt.

**Beispiel 11.6. simplePush.xsl**

```
<?xml version="1.0"?>
<xsl:transform
 version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="xml" indent="yes"
 encoding="iso-8859-1"
 xmlns:xalan="http://xml.apache.org/xslt"
 xalan:indent-amount="2"/>
<xsl:strip-space elements="*" />
<xsl:template match="/"><xsl:apply-templates/></xsl:template>

<xsl:template match="registration">
 <data>
 <badge>
 <xsl:apply-templates select="firstName"/>
 <xsl:apply-templates select="lastName"/>
 <xsl:apply-templates select="affiliation"/>
 </badge>
 <address><xsl:apply-templates select="address"/></address>
 </data>
</xsl:template>

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

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

<xsl:template match="lastName">
<xsl:text> </xsl:text><xsl:apply-templates/><xsl:text>
</xsl:text>
</xsl:template>

<xsl:template match="affiliation">
 <xsl:apply-templates/><xsl:text>
</xsl:text>
</xsl:template>

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

<xsl:template match="line">
<xsl:apply-templates/><xsl:text>
</xsl:text>
</xsl:template>

<xsl:template match="*" />

</xsl:transform>
```

Das Programm erzeugt den gleichen Output wie bereits das vorher vorgestellte Pull-Programm, man erkennt hier jedoch leicht die hierarchische Aufteilung in Untertemplates im Gegensatz zu dem einzelnen Template aus Beispiel 11.4, „simplePull.xsl“. Jedes Template definiert nun sein eigenes Ergebnis-Skelett und das importieren der Inhalte erfolgt über einen rekursiven Aufruf weiterer,

geschachtelter Templates, ausgehend vom gerade bearbeiteten Kontextknoten. Jeder Template-Aufruf gibt an, welche Elemente gerade bearbeitet werden und sucht dann nach passenden Templates für diese Elemente.

Das Push-Prinzip in Reinform ist folgendermaßen definiert:

- Ein separates Template für /
- Dann für jedes weitere Element ein Template
- Jedes Template erstellt sein Ergebniss-Skelett und ruft darin `<xsl:apply-templates>` auf, um die Inhalte zu importieren
- Textknoten werden unmodifiziert übernommen

Die Abarbeitung der Templates geht nach dem Prinzip der Tiefensuche vor. Das Push-Prinzip in Reinform erzeugt eine Ähnlichkeit zwischen Ein- und Ausgabedokument und ist somit besonders für narrative, text-orientierte Dokumente geeignet.

Im Allgemeinen Push-Prinzip kann auch eine gezielte Steuerung der Abarbeitungsreihenfolge der Templates erreicht werden, indem das `select`-Attribut des `xsl:apply-templates`-Elements verwendet wird. Hier kann angegeben werden, welches Template als nächstes angewandt werden soll.

Es ist leicht ersichtlich, dass durch den modularisierten Aufbau eines XSLT Push-Programms eine hohe Wiederverwendbarkeit gegeben ist, da die einzelnen Templates auch für Dokumente anderer Struktur (sofern der Name der Elemente gleich ist) verwendet werden können. Es ist sogar möglich, mehrere Verarbeitungsvorschriften für ein und dasselbe Element zu definieren. Dies kann über `xsl:import` und das zugehörige `xsl:apply-imports` erreicht werden. Dieses Konzept kann man als eine Art Vererbung für Templates betrachten. Beispiel 11.7, „import-example.xml“ veranschaulicht dies.

### Beispiel 11.7. import-example.xml

```
<xsl:import href="full-badge.xml"/>

<xsl:template match="registration">
 <badge>
 <xsl:apply-imports/>
 </badge>
</xsl:template>
```

Mit der zugehörigen Datei `full-badge.xml`

```
<xsl:template match="registration">
 <xsl:apply-templates select="firstName"/>
 <xsl:apply-templates select="lastName"/>
 <xsl:apply-templates select="affiliation"/>
 <xsl:apply-templates select="address"/>
</xsl:template>
```

Angewandt auf die Datei `Doe.xml` aus Abbildung 11.1, „Doe.xml in Mozilla Firefox 3.5“ wird der folgende Output produziert:

```
<badge>
 John
 Doe
 Technische Universität München
 Department of Computer Science
 Boltzmannstraße 3
```

```
85748 Garching b. München
Germany
</badge>
```

Die Namensplakette wird also mit Adressinformation ausgegeben. Möchte man dies nun ändern (z.B. eine Plakette ohne Adresse) bzw. die Möglichkeit haben, zwischen verschiedenen Formaten zu wechseln, reicht es aus, einfach weitere Untertemplates zu definieren und diese mit `xsl:import` einzubinden.

#### 4.2.2.4.3 Funktionales Programmieren

Das Paradigma der funktionalen Programmierung baut auf dem Lambdakalkül, welches von Alonso Church entwickelt wurde, auf. Grundsätzlich besagt es, dass ein Programm komplett aus (mathematischen) Funktionen besteht. Eine Funktion besteht aus Funktionen besteht aus Funktionen ... und auf der untersten Ebene schließlich stehen atomare Ausdrücke. Somit existieren keine imperative Konstrukte wie z.B. Schleifen und ein Programm besteht ausschließlich aus (rekursiven) Funktionsaufrufen, die Funktionen als Argumente erhalten. Die Hauptcharakteristiken einer funktionalen Sprache sind die folgenden:

- Es existieren keine Zuweisungsstatements, d.h. einmal initialisierte Variablen ändern ihren Wert nicht mehr.
- Funktionsaufrufe berechnen nur deren Ergebnis und sonst nichts (keine Änderungen von Variablenwerten, Zustandsänderungen etc.).
- Referentielle Transparenz: Variablen und Werte können zu jedem beliebigen Zeitpunkt substituiert werden.

Eine detaillierte Einführung über die theoretischen Grundlagen von Funktionaler Programmierung würde den Rahmen dieses Skriptes sprengen, deshalb begnügen wir uns mit dem Beispiel XSLT.

Dimitre Novatchev hat 2001 gezeigt, dass XSLT eine vollwertige Funktionale Programmiersprache wie z.B. Haskell oder Lisp ist. XSLT ist somit *berechenbarkeitsuniversell*. Ein Template entspricht in XSLT einer Funktion, zusätzlich gibt es Variablen und Parameter.

**Variablen in XSLT.** In XSLT können Variablen deklariert werden, die, nach dem Konzept der Funktionalen Programmierung, nach der Initialisierung nur noch lesbar, aber nicht mehr überschreibbar sind. Das heißt, die Variablen sind eigentlich Konstanten. Wie aus anderen Sprachen bekannt, gibt es globale Variablen, deren Gültigkeitsbereich das gesamte Transformationsprogramm ist, und lokale Variablen, die nur innerhalb einer Funktion (=Template) gelten. Variablen können über

```
<xsl:variable name="vName" select="value" />
```

oder

```
<xsl:variable name="vName" > ... </xsl:variable>
```

definiert werden. Anschließend kann über `$vName` darauf zugegriffen werden.

**Parameter in XSLT.** Analog gibt es Parameter, die bei einem Templateaufruf übergeben werden können. Es gibt wiederum globale und lokale Parameter die genau wie Variablen mit den Statements

```
<xsl:param name="pName" select="value" />
```

oder

```
<xsl:param name="pName" > ... </xsl:param>
```

definiert und über `$pName` verwendet werden können. Ein Template kann über

```
<xsl:call-template name="tName">
 <xsl:with-param name="pName"/>
</xsl:call-template>
```

mit dem entsprechenden Parameter aufgerufen werden.

Funktionale Programmierung ist in XSLT nun folgendermaßen umgesetzt. Templates können rekursiv mit Argumenten aufgerufen werden, die Ausführung ist jedoch nur Kontextabhängig und unabhängig von vorherigen Berechnungsschritten. Parameter sind read-only, allerdings wird durch einen (rekursiven) Template-Aufruf eine neue Parameterinstanz mit neuen Werten erzeugt. Beispiel 11.8, „rekursion.xml“ zeigt ein Beispiel für ein funktionales XSLT Programm.

### Beispiel 11.8. rekursion.xml

```
<xsl:template name="while">

 <xsl:param name="state"/>
 <xsl:if test="condition on $state">
 perform iteration
 <xsl:variable name="newState" select="compute new $state"/>
 <xsl:call-template name="while">
 <xsl:with-param name="state" select="$newState"/>
 </xsl:call-template>
 </xsl:if>

</xsl:template>
```

Der rekursive Template-Aufruf mit einer neuen Parameterinstanz findet hier in den Zeilen 6-8 statt.

## 4.2.2.5 Ausgewählte XSLT-Befehle

Im folgende wollen wir nun eine Referenz wichtiger XSLT Befehle anführen. Alle Elemente, die zur XSLT-Anwendungsdomäne gehören, werden dadurch identifiziert, dass Standardmäßig ihre Elementnamen mit `xsl:` anfangen und dass sie die folgende Namensraumdeklaration: `http://www.w3.org/1999/XSL/Transformation` verwenden.

**Tabelle 11.1. XSLT-Elemente**

Element	Beschreibung
<code>xsl:stylesheet</code>	repräsentiert das Wurzelement eines XSL-Stylesheets. Im Wurzelement sind alle anderen XSLT-Elemente enthalten.
<code>xsl:transform</code>	ist ein Synonym für <code>xsl:stylesheet</code>
<code>xsl:attribute-set</code>	ist ein TOP-Level-Element <sup>a</sup> definiert eine Gruppe von Attribute im Ergebnisbaum
<code>xsl:decimal-format</code>	Wenn Zahlen in das Ergebnisdokument geschrieben werden sollen, so können wir das <code>xsl:decimal-format</code> -Element um ein Zahlenformat, das verwendet werden soll, festzulegen (TOP-Level-Element)
<code>xsl:import</code>	Mit dem <code>xs:import</code> -Element kann XSLT-Stylesheet ein oder mehrere XSLT-Stylesheets importieren (TOP-Level-Element)
<code>xsl:include</code>	Das <code>xsl:include</code> -Element bindet die Definition eines weiteren XSLT-Stylesheet in das XSLT-Stylesheet, welches die folgende Anweisung enthält <code>&lt;xsl:include href="URI"/&gt;</code> . Im gegensatz zu den durch <code>xsl:import</code> zugänglichen Elementen aus anderen XSLT-Stylesheets wirken die mit <code>xsl:include</code>

Element	Beschreibung
	eingebundene Elemente als ob sie irgendwo im Eltern-Stylesheet definiert sind (TOP-Level-Element)
<code>xsl:key</code>	Mit dem <code>xsl:key</code> -Element werden Schlüssel für den Zugriff auf das Quelldokument deklariert (TOP-Level-Element)
<code>xsl:namespace-alias</code>	Mit dem Befehl <code>xsl:namespace-alias</code> ist es möglich, einen Alias für einen Namensraum zu definieren (TOP-Level-Element)
<code>xsl:output</code>	Das <code>xsl:output</code> -Element legt für den Ergebnisbaum den Dokumenttyp fest (TOP-Level-Element)
<code>xsl:param</code>	Dieser Befehl deklariert einen benannten Parameter (TOP-Level-Element)
<code>xsl:preserve-space</code>	besagt, dass der Leerzeichen-Textknoten erhalten werden sollen (TOP-Level-Element)
<code>xsl:strip-space</code>	besagt, dass der Leerzeichen-Textknoten entfernt werden sollen (TOP-Level-Element)
<code>xsl:variable</code>	Dieser Befehl deklariert eine benannte Variable (TOP-Level-Element)
<code>xsl:template</code>	Das <code>xsl:template</code> -Element definiert ein Templat für das Ergebnisdokument (Top-Level-Element)
<code>xsl:apply-imports</code>	Die mittels des <code>xsl:import</code> -Befehl aus anderen XSLT-Stylesheets importierte Template-Definitionen, können dann mit <code>xsl:apply-imports</code> an einer gewünschten Stelle angewendet werden können.
<code>xsl:apply-templates</code>	Das <code>xsl:apply-templates</code> -Element besagt, dass an der Stelle andere Templates eingebunden werden können
<code>xsl:attribute</code>	erzeugt ein Attributsknoten im Ergebnisbaum
<code>xsl:call-template</code>	Mittels dem <code>xsl:call-template</code> -Element kann eine Vorlage anhand des Namens aufgerufen werden
<code>xsl:choose</code>	Das <code>xsl:choose</code> erlaubt mehrere Bedingungen nebeneinander zu definieren
<code>xsl:comment</code>	generiert einen Text, der als Kommentar übernommen wird
<code>xsl:copy</code>	Das <code>xsl:copy</code> -Element erlaubt, flache Strukturen zu kopieren
<code>xsl:copy-of</code>	Im Gegensatz zum <code>xsl:copy</code> -Element wird das <code>xsl:copy-of</code> -Element benutzt, um tiefe Strukturen (ausgewählte Knoten, deren Kinder, Namensräume und Nachfahren) in den Ergebnisbaum hineinzukopieren
<code>xsl:element</code>	Elemente werden mit dem <code>xsl:element</code> -Konstrukt erzeugt
<code>xsl:fallback</code>	stellt eine alternative Ausführungsanweisung dar, falls der XSLT-Prozessor eine Anweisung nicht kennt und wird nur im Fehlerfall ausgeführt
<code>xsl:for-each</code>	ist eine Schleifenanweisung
<code>xsl:if</code>	ermöglicht eine bedingte Ausführung
<code>xsl:message</code>	gibt eine Nachricht aus
<code>xsl:number</code>	erzeugt einen Stringwert im Ergebnisdokument
<code>xsl:otherwise</code>	ist ein Element zur Steuerung bedingter Verarbeitung, das innerhalb eines <code>xsl:choose</code> -Elements den Fall definiert, dass keiner der <code>when</code> -Tests bestanden wird

Element	Beschreibung
<code>xsl:processing-instruction</code>	Zur Generierung einer Verarbeitungsanweisung in der Ausgabe
<code>xsl:sort</code>	Die <code>xsl:sort</code> sortiert die aktuelle Knotenliste
<code>xsl:text</code>	erzeugt Textelemente
<code>xsl:value-of</code>	gibt den Inhalt eines Ausdrucks an der aktuellen Position im Ausgabedokument aus
<code>xsl:when</code>	ermöglicht eine Bedingung innerhalb einer Auswahl
<code>xsl:with-param</code>	ermöglicht einen Parameter einen Wert zuzuweisen

<sup>a</sup>Ein TOP-Level-Element ist nur als Element auf oberster Ebene erlaubt, also als Kind zu `xsl:stylesheet` in der Datei stehen muss

## 4.2.3 Paradigmen der Informatik

XSLT bietet die Möglichkeit, zwei grundlegende Design-Paradigmen der Informatik erfolgreich umzusetzen. Die im folgenden näher erläuterten Prinzipien sind:

- Pipelines und Filter
- Separation of Concerns

### 4.2.3.1 Pipelines und Filter

Das Konzept der *Pipes und Filter* wurde von Douglas McIlroy entwickelt und 1973 zum ersten mal in einem Unix System umgesetzt. Die Idee dahinter ist, eine Kette (Pipeline) von einzelnen Programmen (Filter) aufzubauen, die jeweils den Output des Vorgängerprogrammes als Input erhalten und diesen dann entsprechend verarbeiten (filtern) und an das nachfolgende Programm weiterzugeben. Abbildung 11.6, „Das Konzept der Pipes and Filters“ veranschaulicht das Konzept.

**Abbildung 11.6. Das Konzept der Pipes and Filters**



Eine denkbare Anwendung des Konzepts aus der Unix-Welt wäre beispielsweise die Aufgabe: Wieviele Dateien, die "TODO" im Namen haben, befinden sich in einem Ordner? Mit Pipes und Filter lässt sich die Frage sehr einfach mit folgendem Skript lösen.

```
ls | grep "TODO" | wc -l
```

`ls` listet alle Dateien im aktuellen Ordner und gibt die List an `grep` weiter, welches die einzelnen Namen auf den regulären Ausdruck "TODO" prüft und nur die Listeneinträge zurück liefert, die eine Übereinstimmung aufweisen. Diese Liste wird dann an `wc` weitergegeben, welches mit der Option `-l` die Listeneinträge zählt. Das `|` Symbol steht hier für die Pipeverknüpfung der einzelnen Teilprogramme.

Am Beispiel wird ersichtlich, dass es mit Hilfe von Pipes möglich ist, für kleine Teilaufgabe einzelne Programme zu erstellen, welche die Aufgabe lösen und diese einzelnen Module in beliebiger Kombination zusammenzufügen um größere Probleme zu lösen.

Bei der Einordnung in XSLT entsprechen nun Templates Filtern, der zu verarbeitende Datenstrom ist ein XML Dokument und die Hintereinanderausführung einzelner Templates bildet die Pipe. Ein solche Pipe für die Veröffentlichung eines in XML vorliegendes Buches könnte z.B. so aussehen:

1. Erstelle die Seitennummerierung

2. Erstelle die Fußnoten
3. Löse Referenzen auf
4. Erstelle Kapitelnummerierung
5. Identifiziere Kontextabhängiges
6. Transformier nach HTML

Das Dokument wird an das erste Template übergeben, dieses wird ausgeführt und dessen Output an das zweite Template weitergereicht. So wird der Publikationsprozess in kleine Teilschritte modularisiert und kann somit (in Teilschritten) auch auf andere Dokumente angewandt werden. Inhalt und Format sind voneinander getrennt.

### 4.2.3.2 Separation of Concerns

*Separation of Concerns* wurde zum ersten mal 1974 von Dijkstra gebraucht und entwickelte sich von da ab zu einem der wichtigsten Paradigmen in der Informatik. Das Prinzip verlangt, dass ein Programm in funktional disjunkte Teilprogramme zerlegt wird. Die Eigenschaft "disjunkt" kann jedoch meistens nicht erfüllt werden und so wird versucht, möglichst kleine Schnittmengen zwischen den einzelnen Teilen zu erreichen. Ein gutes Beispiel für das Prinzip sind *monolithische Kernel* und *Mikrokernel*. Ein monolithischer Kernel enthält den Großteil aller Funktionalitäten, die notwendig sind, um mit der Hardware zu kommunizieren. Ein Mikrokernel hingegen lagert diese Funktionen aus (auf Benutzerebene) und macht sie so unabhängig. Abbildung 11.7, „Monolithischer Kernel und Mikrokernel, vereinfacht“ veranschaulicht das Konzept.

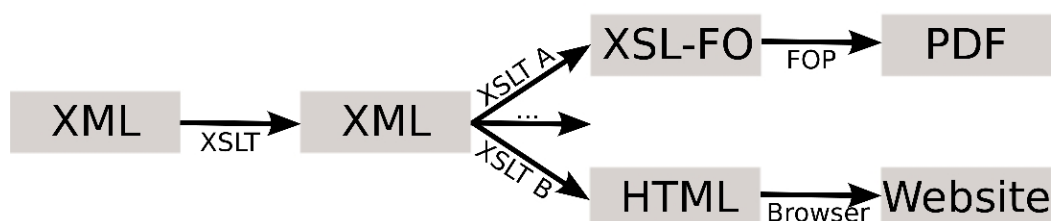
#### Abbildung 11.7. Monolithischer Kernel und Mikrokernel, vereinfacht

Im Zusammenhang mit XSLT wird die Separation of Concerns bei der Trennung von Format und Inhalt eingesetzt. Mit Hilfe eines Stylesheets ist es möglich, ein wohlgeformtes XML Dokument in ein beliebiges Ausgabeformat zu übersetzen, unabhängig von dessen Inhalt. Verschiedene Stylesheets können verschiedene Präsentationsformate erzeugen, jedoch wird der Inhalt immer derselbe bleiben und muss auch nicht, für ein anderes Format, geändert werden. Hierin zeigt sich wiederum die Fähigkeit von XML zum Single-Source-Publishing.

### 4.2.4 XSL Formatting Objects

Neben XSLT gehört auch *XSL Formatting Objects* (XSL-FO) zum XSL Standard. Während XSLT Programme Transformationen von XML (Dialekten) nach XML (Dialekten) beschreiben, wird XSL-FO dazu verwendet, ein Präsentationslayout zu definieren, in welchem das Dokument letztendlich, für den Benutzer aufbereitet, angezeigt wird. Die Formatierungsobjekte legen fest, wie einzelne Elemente (z.B. Textblöcke, Seitennummern, Kopf- oder Fußzeile) auf einer Seite angeordnet werden sollen. Um ein Dokument nach den Vorgaben zu formatieren, wird ein *Formatierer*, wie beispielsweise FOP von Apache oder RenderX (kommerziell), benötigt. Als Analogie hierzu kann eine HTML Seite mit Browser gesehen werden. Während der HTML Code festlegt, wie die Seite aufgebaut ist, sorgt der Browser dafür, dass die Layoutvorgaben umgesetzt und entsprechend angezeigt werden.

#### Abbildung 11.8. XSL Workflow





In Abbildung 11.8, „XSL Workflow“ ist ein Beispiel für einen XSL Workflow dargestellt. Ein XML-Dokument wird zunächst mit beliebigen XSLT Programmen transformiert und schließlich mit speziellen Transformationen in einen spezifischen XML Dialekt, wie z.B. XSL-FO, gebracht. Ausgehend von diesem Dokument wird dann das Layout mit Hilfe eines Formatierers erstellt. Dieses Kapitel beschäftigt sich mit dem dargestellte Weg, aus einem XML Dokument mittels XSL-FO ein formatiertes Ausgabedokument zu erstellen.

#### 4.2.4.1 Der XSL-FO Workflow

Zunächst wollen wir uns einmal ansehen, wie ein XSL-FO Dokument grundlegend aufgebaut ist und wie daraus ein formatiertes Dokument entsteht. In XSL-FO wird das Layout durch mehrere rechteckige *Areas* beschrieben. Diese Areas können sein:

- Seitenregion-Area
- Block-Area
- Inline-Area
- Glyphen

**Abbildung 11.9. XSL Areas**

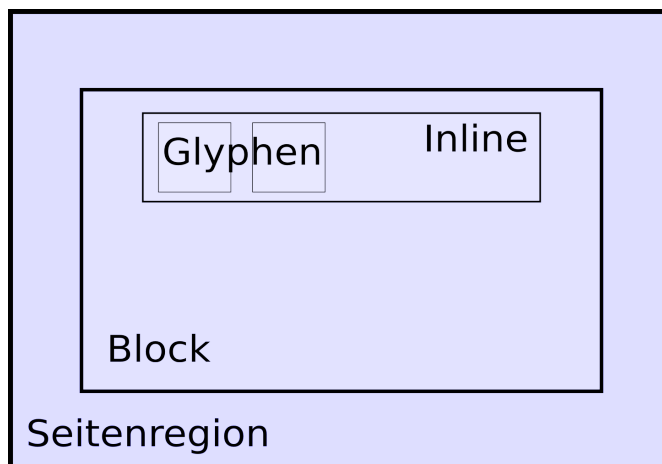


Abbildung 11.9, „XSL Areas“ zeigt einen schematischen Dokumentaufbau. Die Areas sind hier analog zu CSS aufgebaut, d.h. sie besitzen jeweils Margin, Border, Padding und Content. Nach der Anwendung des FO-Prozessors wird aus dem ursprünglichen Dokument ein *Area-Tree* erstellt, dass Dokument ist nun also vollkommen durch die verschiedenen (geschachtelten) Areas und deren Position beschrieben. Dieses Zwischenergebniss, als abstrakte Beschreibung der Formatierung, kann nun gerendert werden.

**Abbildung 11.10. XSL-FO Workflow**

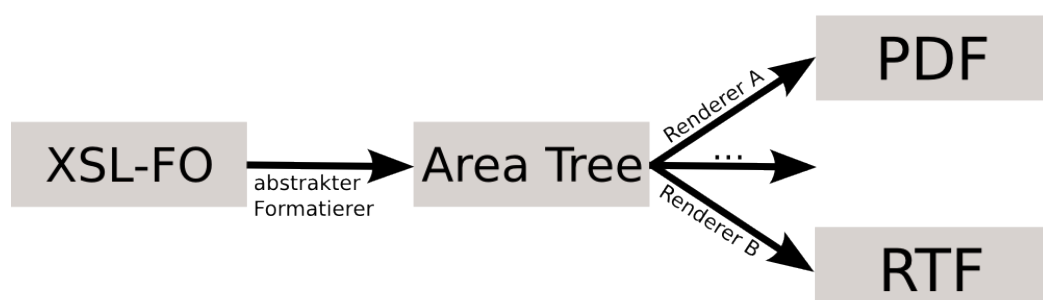


Abbildung 11.10, „XSL-FO Workflow“ visualisiert den Ablauf.

Auch hier kann man wiederum eine Umsetzung des Separation of Concerns Paradigmas erkennen. Der Renderprozess ist vollkommen unabhängig und arbeitet nur auf dem Area-Tree. Bis jetzt existieren nur Renderer für das PDF-Format, aufgrund des Designs ist es jedoch prinzipiell möglich auch Renderingsoftware für andere Formate bereitzustellen, ohne am eigentlichen Übersetzungsprozess von XSL-FO in die Area-Tree Darstellung etwas ändern zu müssen.

#### 4.2.4.2 Aufbau eines XSL-FO Dokuments

Sehen wir uns nun einmal an, wie mit Hilfe von XSL-FO PDF Dokumente erzeugt werden können. Wie der Workflow in Abbildung 11.8, „XSL Workflow“ bereits gezeigt hat, wird ein XSL-FO Programm nicht vom Benutzer implementiert, sondern mittels einer XSLT Transformation erzeugt. Wir werden nun ein Beispiel kennen lernen, wie wir aus der Datei aus Beispiel 11.1, „Doe.xml“ eine Plakette in PDF Format erstellen können. Beispiel 11.9, „badge.xsl“ zeigt das Transformationsprogramm, um Doe.xml in ein XSL-FO Dokument zu überführen.

##### Beispiel 11.9. badge.xsl

```
<?xml version="1.0" encoding="UTF-8" ?>
<xsl:stylesheet version="1.0" xmlns:fo=
 "http://www.w3.org/1999/XSL/Format"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="xml" version="1.0" indent="yes" />
<xsl:template match="registration">
 <fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">
 <fo:layout-master-set>
 <fo:simple-page-master page-height="297mm" page-width="210mm"
 margin="5mm 25mm 5mm 25mm" master-name="PageMaster">
 <fo:region-body margin="20mm 0mm 20mm 0mm"/>
 </fo:simple-page-master>
 </fo:layout-master-set>
 <fo:page-sequence master-reference="PageMaster">
 <fo:flow flow-name="xsl-region-body" >
 <fo:block>
 <xsl:apply-templates select="lastName"/>
 <xsl:text>, </xsl:text>
 <xsl:apply-templates select="firstName"/>
 <xsl:text>
 </xsl:text>
 <xsl:apply-templates select="affiliation"/>
 </fo:block>
 </fo:flow>
 </fo:page-sequence>
 </fo:root>
</xsl:template>
<xsl:template match="lastName"> <xsl:apply-templates />
</xsl:template>
<xsl:template match="firstName"> <xsl:apply-templates />
</xsl:template>
<xsl:template match="affiliation">
 <fo:block> <xsl:apply-templates /> </fo:block>
</xsl:template>

</xsl:stylesheet>
```

Das Programm enthält ausschließlich bekannte Elemente, die bereits in Abschnitt „4.2.2 XML Style Language Transformation“ vorgestellt wurden. Sehen wir uns also gleich die Ausgabedatei an, die bei Anwendung auf "Doe.xml" erzeugt wird.

### Beispiel 11.10. badge.fo

```
<?xml version="1.0"?>
<fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">
 <fo:layout-master-set>
 <fo:simple-page-master page-height="297mm"
 page-width="210mm" margin="5mm 25mm 5mm 25mm"
 master-name="PageMaster">
 <fo:region-body margin="20mm 0mm 20mm 0mm"/>
 </fo:simple-page-master>
 </fo:layout-master-set>
 <fo:page-sequence master-reference="PageMaster">
 <fo:flow flow-name="xsl-region-body">
 <fo:block>Doe, John
<fo:block>Technische Universität München</fo:block></fo:block>
 </fo:flow>
 </fo:page-sequence>
</fo:root>
```

Beispiel 11.10, „badge.fo“ zeigt die Ausgabe. Das Dokument beginnt wieder mit einer XML Deklaration welche anzeigt, dass es sich hierbei um ein XML Dokument handelt. Anschließend wird über `<fo:root >` das fo-Präfix mit dem Namensraum für XSL-FO verknüpft. Dieses Element bildet gleichzeitig die Wurzel des XSL-FO Baums. Die Wurzel enthält nun zwei Kindknoten `<fo:layout-master-set>` und `<fo:page-sequence>`. Über `layout-master-set` wird das Seitenlayout definiert, also Breite, Höhe, Ränder, etc. Innerhalb von `page-sequence` wird dann der eigentliche Inhaltsfluss Seitenweise dargestellt.

`<fo:flow>` sorgt nun für die korrekten Seitenumbrüche und stellt den Text flussweise dar. Unterelemente hier sind die möglichen Areas wie `<fo:block>` oder `<fo:inline>`. Die Ausgabe ist der folgende Text in einer PDF Datei:

Doe, John  
Technische Universität München

## 4.2.4.3 XSL-FO Vokabular

Dieser Abschnitt soll eine kurze Übersicht geben, was mit XSL-FO möglich ist. Die Übersicht dient lediglich als Einstieg und ist in keinsten Weise vollständig, stellt jedoch einen guten Startpunkt für einen tieferen Einstieg bereit.

**Text.** Mit den `fo:block` und `fo:inline` Elementen kann dargestellter Text in Form, Farbe und Schriftart formatiert werden. Hierzu existieren die Attribute `font-family`, `font-size`, `font-style`, `font-weight` `text-decoration` und `color`.

**Rahmen.** Block-Elemente können auch mit einem Rahmen versehen werden. Hierzu können `border-position-color`, `border-position-style` und `border-position-width` verwendet werden. *position* kann hierbei die Werte `top`, `bottom`, `left` oder `right` annehmen.

**Ausrichtung.** Innerhalb von Block-Elementen kann text mit `text-align="..."` ausgerichtet werden.

**Links.** Links innerhalb eines Dokuments oder auch nach extern, werden mit dem Element `<fo:basic-link>` definiert. Die Attribute `internal-destination` bzw. `external-destination` legen hierbei das Ziel fest.

**Listen.** Listen werden mit `<fo:list-block>` gestartet und Listenelemente können mit `<fo:list-item>` hinzugefügt werden.

**Graphiken.** Um Bilder einzufügen, verwendet man `<fo:external-graphic>` mit den Attributen `src`, `content-height` und `content-width`.

**Seitenlayout.** Wie bereits erwähnt, wird das Seitenlayout mittels `<fo:layout-master-set>` definiert. Das Unterelement `<simple-page-master>` legt die Standardeigenschaften fest. Hierzu stehen u.a. die Attribute `master-name`, `page-height`, `page-width` und `margin/padding-position` zur Verfügung. Zusätzlich ist ein Ausgabedokument in fünf Regionen gegliedert, welche mittels der Elemente `<fo:region-body>`, `<fo:region-before>`, `<fo:region-after>`, `<fo:region-start>` und `<fo:region-end>` angesprochen und gelayoutet werden können.

**Seitensequenzen.** Das allgemeine Seitenlayout, welches über `<fo:layout-master-set>` definiert wurde kann mit `<fo:page-sequence master-reference="...">` festgelegt werden. Zusätzlich gibt es die Möglichkeit, für einzelne Seiten, spezielle Layouts festzulegen (`<fo:single-page-master-reference master-name="..." />`) bzw. für mehrere aufeinander folgende Seiten ein spezielles Layout über `<fo:repeatable-page-master-reference master-name="..." maximum-repeats="..." />` zu definieren. Hiermit kann man den Ablauf der Seitenlayouts (wann welches Layout für welche Seite verwendet werden soll) steuern.

Wie gesagt gibt es noch eine Vielzahl von Elementen und zugehörigen Attributen, welche zum layouten eines Dokuments in XSL-FO herangezogen werden können. Die W3C-Referent unter <http://www.w3schools.com/xslfo/> bietet hierzu eine komplette Übersicht mit Tutorials um das volle Potential von XSL-FO ausschöpfen zu können.

---

# Kapitel 12. Zusammenfassung

In diesem Skript haben wir den Stand und die Erfordernisse elektronischen Publizierens, insbesondere des wissenschaftlichen Publizierens, dargestellt und analysiert. Das Skript gibt einen Überblick über Basistechnologie, Werkzeuge und Modelle elektronischen Publizierens, erörtert neue Möglichkeiten insbesondere des Online-Publizierens im Internet und zeigt auf, wo die heutigen Methoden und Werkzeuge zu kurz greifen. Im Zentrum der Darstellung standen die Extensible Markup Language (XML) und ihre Satellitenstandards.

XML und Ihre Satellitenstandards sind Sprachen, die leicht zu erlernen sind, auch ohne fortgeschrittene Programmierkenntnisse. Dieses Skript ist ein guter Einstiegspunkt, um XML und einigen Ihrer Satellitenstandards (DTD, XML-Schema, XPath, XQuery und XSLT) mit ihrer Konzepte und Besonderheiten zu erlernen.

Die folgende Auflistung gibt eine Übersicht über die Themen, die in dem Skript behandelt werden:

- Teil I: Strukturierte Dokumente und ihre Kodierung (Begriffe, Modell der strukturierten Dokumente, XML (nur Instanzen), XML Namespaces, Zeichenkodierung (Unicode)).
- Teil II: Schema-Ansätze für XML am Beispiel von DTDs und XML Schema.
- Teil III: Beziehungen und Abfragesprachen (Baummodell volles Datenmodell XDM, XPath, XQuery).
- Teil IV: Verarbeitung von XML-Daten (XSLT).

Dieses Skript ist als Leitfaden gedacht, den Sie in der Vorlesung kommentieren und ergänzen könnten, was Ihnen auch einiges an Mitschreiben während der Vorlesung ersparen lässt. Bitte beachten Sie, dass die Lektüre des Skripts den Besuch der Vorlesung nicht ersetzen kann. Prüfungsrelevant ist das, was in der Veranstaltung behandelt wird.

Dieses Skript wurde in Docbook erstellt und in PDF umgewandelt werden.

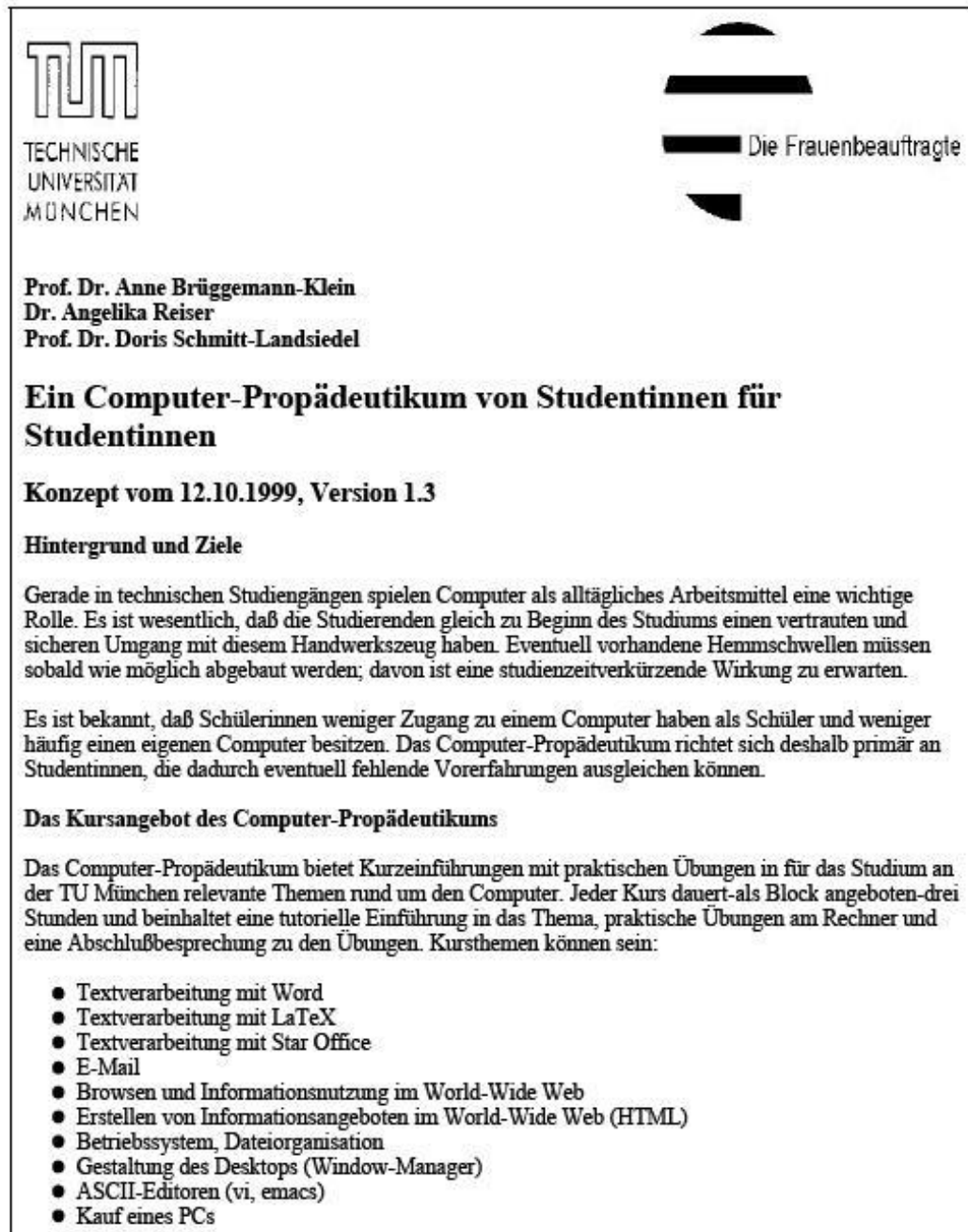
Prof. Dr. Brüggemann-Klein  
Dipl.-Inf. Univ. Marouane Sayih

---

# Anhang A. Beispiele

Formate am Beispiel



















Abbildung A.1. Ein formatiertes Dokument



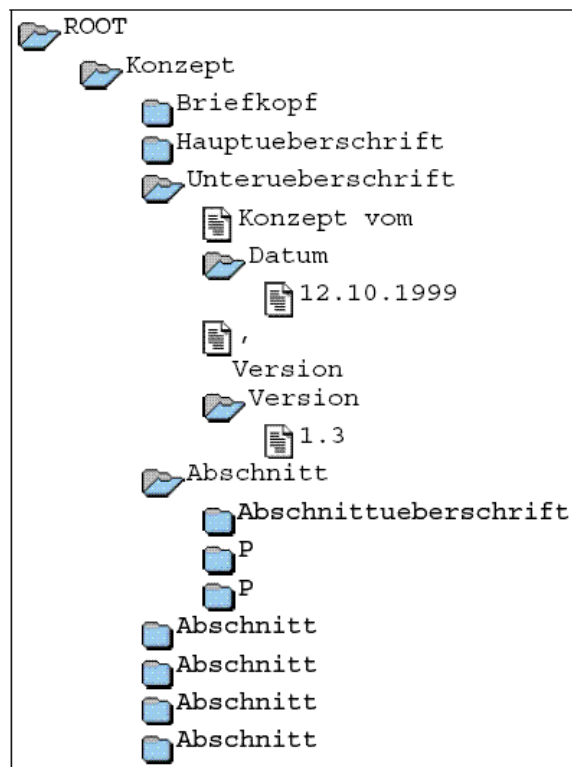
**Beispiel A.1. Ein unformatiertes Dokument mit reinem Text**

Prof. Dr. Anne Brueggemann-Klein Dr. Angelika Reiser Prof. Dr. Doris Schmitt-Landsiedel Ein Computer-Propaedeutikum von Studentinnen fuer Studentinnen Konzept vom 12.10.1999, Version 1.3 Hintergrund und Ziele Gerade in technischen Studiengaengen spielen Computer als alltaegliches Arbeitsmittel eine wichtige Rolle. Es ist wesentlich, dass die Studierenden gleich zu Beginn des Studiums einen vertrauten und sicheren Umgang mit diesem Handwerkszeug haben. Eventuell vorhandene Hemmschwellen muessen sobald wie moeglich abgebaut werden; davon ist eine studienzeitverkuerzende Wirkung zu erwarten. Es ist bekannt, dass Schuelerinnen weniger Zugang zu einem Computer haben als Schueler und weniger haeufig einen eigenen Computer besitzen. Das Computer-Propaedeutikum richtet sich deshalb primaer an Studentinnen, die dadurch eventuell fehlende Vorerfahrungen ausgleichen koennen. Das Kursangebot des Computer-Propaedeutikums Das Computer-Propaedeutikum bietet Kurzeinfuehrungen mit praktischen Uebungen in fuer das Studium an der TU Muenchen relevante Themen rund um den Computer. Jeder Kurs dauert-als Block angeboten-drei Stunden und beinhaltet eine tutorielle Einfuehrung in das Thema, praktische Uebungen am Rechner und eine Abschlussbesprechung zu den Uebungen. Kursthemen koennen sein: Textverarbeitung mit Word Textverarbeitung mit LaTeX Textverarbeitung mit Star Office E-Mail Browsen und Informationsnutzung im World-Wide Web Erstellen von Informationsangeboten im World-Wide Web (HTML) Betriebssystem, Dateioorganisation Gestaltung des Desktops (Window-Manager) ASCII-Editoren (vi, emacs) Kauf eines PCs Datenverbindungen von zu Hause (analog, ISDN) Fakultaetsspezifische Informationsinfrastrukturen Jeder Kurs sollte sich auf ein enges Thema konzentrieren. Die Themenliste ist offen fuer Ergaenzungen. Kriterium ist, dass der Kursinhalt propaedeutischer Natur fuer das Studium an der TU ist und eine Form der Computernutzung zum Inhalt hat. Ein Kurs kann fakultaetsuebergreifend oder fakultaetsspezifisch sein. Die Kurse sind in der Regel auf eine spezielle Rechnerplattform (PC/Windows, Unix) zugeschnitten. Die Zielgruppe des Computer-Propaedeutikums Das Kursangebot richtet sich an Studentinnen im ersten Semester. Wird ein Angebot von dieser Zielgruppe nicht ausgeschoept, steht es auch Studentinnen hoeherer Semester und, in dritter Prioritaet, maennlichen Studierenden offen. Die Leiterinnen und Leiter des Computer-Propaedeutikums Die Kurse werden vorzugsweise von Studentinnen hoeherer Semester konzipiert und durchgefuehrt. Zur Ergaenzung des Kursangebots koennen auch Mitarbeiterinnen und Mitarbeiter oder Studenten hoeherer Semester Kurse anbieten. Nur von Studierenden angebotene Kurse koennen im Rahmen des Programms finanziert werden. Die Kursleiterinnen und -leiter sind selbst fuer die Organisation von Seminar- und Rechnerraeumen und von Zugangsmoeglichkeiten zu den Raeumen am Wochenende fuer ihren Kurs verantwortlich. Wir gehen davon aus, dass die Studiendekaninnen und -dekane sie bei der Organisation unterstuetzen. Durchfuehrung Das Computer-Propaedeutikum ist eine Initiative der Frauenbeauftragten der TU Muenchen und wird vom Frauenbuero (Kerstin Hansen, Anja Quindeau) organisiert. Die TU Muenchen finanziert das Programm aus dem Tutorienprogramm zur Verkuerzung der Studiendauer. Fuer von Studierenden angebotene Kurse wird ein Werksvertrag abgeschlossen, der die Konzeption, Vorbereitung und Durchfuehrung eines dreistueendigen Kurses beinhaltet. Konzeption und Vorbereitung eines Kurses werden mit 200 DM und jede Durchfuehrung mit 100 DM honoriert. Das Frauenbuero erhaelt Hilfskraftmittel aus dem Tutorienprogramm fuer die Organisation des Computer-Propaedeutikums. Wir rechnen mit folgenden Kosten: 20 Kurse, jeder zweimal durchgefuehrt, zu 20 mal 200 DM plus 20 mal 2 mal 100 DM, also 8000 DM, plus 1000 DM fuer die Organisation im Frauenbuero. Eine erste Ausschreibung des Programms ist bereits erfolgt. Interessierte Kursleiterinnen und -leiter koennen ihr Kursangebot bis zum 5.11.1999 abgeben. Bitte benutzen Sie dazu unser Formular. Die ersten Kurse sollen noch im November 1999 stattfinden. Bei Bedarf erfolgt eine zweite Ausschreibung, zu der neue Kursangebote abgegeben werden koennen, Anfang Dezember 1999. Die Ausschreibung des Programms, die Gestaltung des Kursangebots sowie die Ankuendigung der Kurse erfolgt durch das Frauenbuero in Zusammenarbeit mit den Studierendenvertretungen, den Studiendekanen und den Frauenbeauftragten. Die Vertraege werden nach Absprache mit dem Frauenbuero von der Hochschulverwaltung abgeschlossen. Die Kurse sollen zu Randzeiten (Abends, Freitags Nachmittags oder am Wochenende) stattfinden.

Abbildung A.2. Eine Dokumentdarstellung mit geschachtelten Boxen

<b>Konzept</b>															
<b>Briefkopf</b>	<table border="1"> <tr> <td><b>Institution</b></td> <td> fbLogo</td> </tr> <tr> <td><b>Autorinnen</b></td> <td> <table border="1"> <tr> <td><b>Autorin (3)</b></td> <td></td> </tr> <tr> <td></td> <td></td> </tr> <tr> <td><b>1</b></td> <td>Prof. Dr. Anne Brüggemann-Klein</td> </tr> <tr> <td><b>2</b></td> <td>Dr. Angelika Reiser</td> </tr> <tr> <td><b>3</b></td> <td>Prof. Dr. Doris Schmitt-Landsiedel</td> </tr> </table> </td> </tr> </table>	<b>Institution</b>	 fbLogo	<b>Autorinnen</b>	<table border="1"> <tr> <td><b>Autorin (3)</b></td> <td></td> </tr> <tr> <td></td> <td></td> </tr> <tr> <td><b>1</b></td> <td>Prof. Dr. Anne Brüggemann-Klein</td> </tr> <tr> <td><b>2</b></td> <td>Dr. Angelika Reiser</td> </tr> <tr> <td><b>3</b></td> <td>Prof. Dr. Doris Schmitt-Landsiedel</td> </tr> </table>	<b>Autorin (3)</b>				<b>1</b>	Prof. Dr. Anne Brüggemann-Klein	<b>2</b>	Dr. Angelika Reiser	<b>3</b>	Prof. Dr. Doris Schmitt-Landsiedel
<b>Institution</b>	 fbLogo														
<b>Autorinnen</b>	<table border="1"> <tr> <td><b>Autorin (3)</b></td> <td></td> </tr> <tr> <td></td> <td></td> </tr> <tr> <td><b>1</b></td> <td>Prof. Dr. Anne Brüggemann-Klein</td> </tr> <tr> <td><b>2</b></td> <td>Dr. Angelika Reiser</td> </tr> <tr> <td><b>3</b></td> <td>Prof. Dr. Doris Schmitt-Landsiedel</td> </tr> </table>	<b>Autorin (3)</b>				<b>1</b>	Prof. Dr. Anne Brüggemann-Klein	<b>2</b>	Dr. Angelika Reiser	<b>3</b>	Prof. Dr. Doris Schmitt-Landsiedel				
<b>Autorin (3)</b>															
															
<b>1</b>	Prof. Dr. Anne Brüggemann-Klein														
<b>2</b>	Dr. Angelika Reiser														
<b>3</b>	Prof. Dr. Doris Schmitt-Landsiedel														
<b>Hauptueberschrift</b>	Ein Computer-Propädeutikum von Studentinnen für Studentinnen														
<b>Nebenueberschrift</b>	<table border="1"> <tr> <td></td> <td>Konzept vom</td> </tr> <tr> <td></td> <td>12.10.1999</td> </tr> <tr> <td></td> <td>Version</td> </tr> <tr> <td><b>Version</b></td> <td>1.3</td> </tr> </table>		Konzept vom		12.10.1999		Version	<b>Version</b>	1.3						
	Konzept vom														
	12.10.1999														
	Version														
<b>Version</b>	1.3														
<b>Abschnitt</b>	<table border="1"> <tr> <td><b>Abschnittueberschrift</b></td> <td>Hintergrund und Ziele</td> </tr> <tr> <td><b>P</b></td> <td>Gerade in technischen Studiengängen spielen Computer als alltägliches Arbeitsmittel eine wichtige Rolle. Es ist wesentlich, daß die Studierenden gleich zu Beginn des Studiums einen vertrauten und sicheren Umgang mit diesem Handwerkszeug haben. Eventuell vorhandene Hemmschwellen müssen sobald wie möglich abgebaut werden; davon ist eine studienzeitverkürzende Wirkung zu erwarten.</td> </tr> <tr> <td><b>P</b></td> <td>Es ist bekannt, daß Schülerinnen weniger Zugang zu einem Computer haben als Schüler und weniger häufig einen eigenen Computer besitzen. Das Computer-Propädeutikum richtet sich deshalb primär an Studentinnen, die dadurch eventuell fehlende Vorerfahrungen ausgleichen können.</td> </tr> </table>	<b>Abschnittueberschrift</b>	Hintergrund und Ziele	<b>P</b>	Gerade in technischen Studiengängen spielen Computer als alltägliches Arbeitsmittel eine wichtige Rolle. Es ist wesentlich, daß die Studierenden gleich zu Beginn des Studiums einen vertrauten und sicheren Umgang mit diesem Handwerkszeug haben. Eventuell vorhandene Hemmschwellen müssen sobald wie möglich abgebaut werden; davon ist eine studienzeitverkürzende Wirkung zu erwarten.	<b>P</b>	Es ist bekannt, daß Schülerinnen weniger Zugang zu einem Computer haben als Schüler und weniger häufig einen eigenen Computer besitzen. Das Computer-Propädeutikum richtet sich deshalb primär an Studentinnen, die dadurch eventuell fehlende Vorerfahrungen ausgleichen können.								
<b>Abschnittueberschrift</b>	Hintergrund und Ziele														
<b>P</b>	Gerade in technischen Studiengängen spielen Computer als alltägliches Arbeitsmittel eine wichtige Rolle. Es ist wesentlich, daß die Studierenden gleich zu Beginn des Studiums einen vertrauten und sicheren Umgang mit diesem Handwerkszeug haben. Eventuell vorhandene Hemmschwellen müssen sobald wie möglich abgebaut werden; davon ist eine studienzeitverkürzende Wirkung zu erwarten.														
<b>P</b>	Es ist bekannt, daß Schülerinnen weniger Zugang zu einem Computer haben als Schüler und weniger häufig einen eigenen Computer besitzen. Das Computer-Propädeutikum richtet sich deshalb primär an Studentinnen, die dadurch eventuell fehlende Vorerfahrungen ausgleichen können.														
<b>Abschnitt</b>	<table border="1"> <tr> <td><b>Abschnittueberschrift</b></td> <td>Das Kursangebot des</td> </tr> </table>	<b>Abschnittueberschrift</b>	Das Kursangebot des												
<b>Abschnittueberschrift</b>	Das Kursangebot des														



**Abbildung A.3. Eine Dokumentdarstellung mit eingerückten Ikonen**



**Beispiel A.2. Repräsentation der logischen Struktur mit Hilfe von eingebettetem Markup**

```
<?xml version="1.0"?>
<!DOCTYPE Konzept SYSTEM "Konzept.dtd">
<Konzept>
 <Briefkopf>
 <Institution Logo="fbLogo"/>
 <Autorinnen>
 <Autorin>Prof. Dr. Anne Brüggemann-Klein</Autorin>
 <Autorin>Dr. Angelika Reiser</Autorin>
 <Autorin>Prof. Dr. Doris Schmitt-Landsiedel</Autorin>
 </Autorinnen>
 </Briefkopf>
 <Hauptueberschrift>Ein Computer-Propädeutikum
 von Studentinnen für Studentinnen</Hauptueberschrift>
 <Nebenueberschrift>Konzept vom <Datum>12.10.1999</Datum>,
 Version <Version>1.3</Version></Nebenueberschrift>
 <Abschnitt>
 <Abschnittueberschrift>Hintergrund und Ziele
 </Abschnittueberschrift>
 <P>Gerade in technischen Studiengängen spielen
 Computer als alltägliches Arbeitsmittel eine wichtige Rolle.
 Es ist wesentlich, daß die Studierenden
 gleich zu Beginn des Studiums einen vertrauten und sicheren
 Umgang mit diesem Handwerkszeug haben. Eventuell vorhandene
 Hemmschwellen müssen sobald wie
 möglich abgebaut werden;
 davon ist eine studienzeitverkürzende Wirkung zu erwarten.
 </P>
 <P>Es ist bekannt, daß Schülerinnen weniger Zugang
 zu einem Computer haben als Schüler und weniger häufig
 einen eigenen Computer besitzen.
 Das Computer-Propädeutikum richtet sich deshalb primär an
 Studentinnen, die dadurch eventuell fehlende Vorerfahrungen
 ausgleichen können.</P>
 </Abschnitt>
 <Abschnitt>
 <Abschnittueberschrift>Das Kursangebot des
 Computer-Propädeutikums</Abschnittueberschrift>
 <P>Das Computer-Propädeutikum bietet Kurzeinführungen
 mit praktischen Übungen in für das Studium an
 der TU München relevante Themen rund um den Computer.
 Jeder Kurs dauert-als Block angeboten-drei Stunden
 und beinhaltet eine tutorielle Einführung in das Thema,
 praktische Übungen am Rechner und eine
 Abschlußbesprechung zu den Übungen.
 Kursthemen können sein:</P>

 Textverarbeitung mit Word
 Textverarbeitung mit LaTeX
 Textverarbeitung mit Star Office
 E-Mail
 Browsen und Informationsnutzung im World-Wide Web
 Erstellen von Informationsangeboten im World-Wide Web
 (HTML)
```

```
Betriebssystem, Dateiorganisation
Gestaltung des Desktops (Window-Manager)
ASCII-Editoren (vi, emacs)
Kauf eines PCs
Datenverbindungen von zu Hause (analog, ISDN)
Fakultätsspezifische Informationsinfrastrukturen

<P>Jeder Kurs sollte sich auf ein enges Thema konzentrieren.
Die Themenliste ist offen für Ergänzungen.
Kriterium ist, daß der
Kursinhalt propädeutischer Natur für das Studium an
der TU ist und eine Form der Computernutzung zum Inhalt hat.
Ein Kurs kann fakultätsübergreifend
oder fakultätsspezifisch sein. Die Kurse sind in der Regel
auf eine spezielle Rechnerplattform (PC/Windows, Unix)
zugeschnitten.
</P>
</Abschnitt>
<Abschnitt>
 <Abschnittueberschrift>Die Zielgruppe des
 Computer-Propädeutikums</Abschnittueberschrift>
 <P>Das Kursangebot richtet sich an Studentinnen im ersten Semester.
 Wird ein Angebot von dieser Zielgruppe nicht ausgeschöpft, steht
 es auch Studentinnen höherer Semester und, in dritter Priorität,
 männlichen Studierenden offen.</P>
</Abschnitt>
<Abschnitt>
 <Abschnittueberschrift>Die Leiterinnen und Leiter des
 Computer-Propädeutikums</Abschnittueberschrift>
 <P>Die Kurse werden vorzugsweise von Studentinnen höherer Semester
 konzipiert und durchgeführt. Zur Ergänzung des Kursangebots
 können auch Mitarbeiterinnen und Mitarbeiter
 oder Studenten höherer Semester Kurse anbieten.
 Nur von Studierenden angebotene Kurse können im Rahmen
 des Programms finanziert werden.</P>
 <P>Die Kursleiterinnen und -leiter sind selbst für die
 Organisation von Seminar- und Rechnerräumen und von
 Zugangsmöglichkeiten zu den Räumen am Wochenende
 für ihren Kurs verantwortlich. Wir gehen davon aus,
 daß die Studiendekaninnen und -dekane sie bei der
 Organisation unterstützen.</P>
</Abschnitt>
<Abschnitt>
 <Abschnittueberschrift>Durchführung</Abschnittueberschrift>
 <P>Das Computer-Propädeutikum ist eine Initiative der
 Frauenbeauftragten der TU München und wird vom
 Frauenbüro (Kerstin Hansen, Anja Quindeau) organisiert.
 Die TU München finanziert das Programm aus dem Tutorienprogramm
 zur Verkürzung der Studiendauer.
 Für von Studierenden angebotene Kurse
 wird ein Werksvertrag abgeschlossen, der
 die Konzeption, Vorbereitung und Durchführung
 eines dreistündigen Kurses beinhaltet.
 Konzeption und Vorbereitung eines Kurses werden mit 200 DM
 und jede Durchführung mit 100 DM honoriert.
 Das Frauenbüro erhält Hilfskraftmittel
 aus dem Tutorienprogramm für die
 Organisation des Computer-Propädeutikums.
```

Wir rechnen mit folgenden Kosten: 20 Kurse,  
jeder zweimal durchgeführt,  
zu 20 mal 200 DM plus 20 mal 2 mal 100 DM, also 8000 DM,  
plus 1000 DM für die Organisation im Frauenbüro.</P>  
<P>Eine erste <A HREF="Ausschreibung.html">Ausschreibung</A>  
des Programms ist bereits erfolgt.  
Interessierte Kursleiterinnen und -leiter  
können ihr Kursangebot bis zum 5.11.1999 abgeben.  
Bitte benutzen Sie dazu unser  
<A HREF="Formular.html">Formular</A>.  
Die ersten Kurse sollen noch im November 1999  
stattfinden. Bei Bedarf erfolgt eine zweite Ausschreibung,  
zu der neue Kursangebote abgegeben werden können,  
Anfang Dezember 1999.  
Die Ausschreibung des Programms, die Gestaltung des Kursangebots  
sowie die Ankündigung der Kurse erfolgt durch das Frauenbüro  
in Zusammenarbeit mit den Studierendenvertretungen,  
den Studiendekanen und den Frauenbeauftragten.  
Die Verträge werden nach Absprache mit dem Frauenbüro  
von der Hochschulverwaltung abgeschlossen.</P>  
<P>Die Kurse sollen zu Randzeiten (Abends, Freitags Nachmittags  
oder am Wochenende) stattfinden.</P>  
</Abschnitt>  
</Konzept>

## Abbildung A.5. Eine Verfeinerung durch Einrückungen und typographische Differenzierung

Seite 1 von 2

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE Konzept (View Source for full doctype...)>
<Konzept>
 <Briefkopf>
 <Institution Logo="fbLogo" />
 <Autorinnen>
 <Autorin>Prof. Dr. Anne Brueggemann-Klein</Autorin>
 <Autorin>Dr. Angelika Reiser</Autorin>
 <Autorin>Prof. Dr. Doris Schmitt-Landsiedel</Autorin>
 </Autorinnen>
 </Briefkopf>
 <Hauptueberschrift>Ein Computer-Propaedeutikum von Studentinnen fuer
 Studentinnen</Hauptueberschrift>
 <Nebenueberschrift>
 <Konzept vom
 <Datum>12.10.1999</Datum>
 , Version
 <Version>1.3</Version>
 </Nebenueberschrift>
 <Abschnitt>
 <Abschnittueberschrift>Hintergrund und Ziele</Abschnittueberschrift>
 <P>Gerade in technischen Studiengaengen spielen Computer als
 alltaegliches Arbeitsmittel eine wichtige Rolle. Es ist wesentlich, dass
 die Studierenden gleich zu Beginn des Studiums einen vertrauten und
 sicheren Umgang mit diesem Handwerkszeug haben. Eventuell
 vorhandene Hemmschwellen muessen sobald wie moeglich abgebaut
 werden; davon ist eine studienzeitverkuerzende Wirkung zu
 erwarten.</P>
 <P>Es ist bekannt, dass Schuelerinnen weniger Zugang zu einem
 Computer haben als Schueler und weniger haeufig einen eigenen
 Computer besitzen. Das Computer-Propaedeutikum richtet sich
 deshalb primaer an Studentinnen, die dadurch eventuell fehlende
 Vorerfahrungen ausgleichen koennen.</P>
 </Abschnitt>
 <Abschnitt>
 <Abschnittueberschrift>Das Kursangebot des Computer-
 Propaedeutikums</Abschnittueberschrift>
 <P>Das Computer-Propaedeutikum bietet Kurzeinfuehrungen mit
 praktischen Uebungen in fuer das Studium an der TU Muenchen
 relevante Themen rund um den Computer. Jeder Kurs dauert als Block
 angeboten drei Stunden und beinhaltet eine tutorielle Einfuehrung in
 das Thema, praktische Uebungen am Rechner und eine
 Abschlussbesprechung zu den Uebungen. Kursthemen koennen
 sein:</P>

 Textverarbeitung mit Word
 Textverarbeitung mit LaTeX
 Textverarbeitung mit Star Office
 E-Mail
 Browsen und Informationsnutzung im World-Wide Web
 Erstellen von Informationsangeboten im World-Wide Web
 (HTML)
 Betriebssystem, Dateiorganisation
 Gestaltung des Desktops (Window-Manager)
 ASCII-Editoren (vi, emacs)
 Kauf eines PCs
 Datenverbindungen von zu Hause (analog, ISDN)

 </Abschnitt>

```

file://A:\Konzept.xml

21.09.00

**Beispiel A.3. Ein Dokument im LaTeX-Format**

```
\documentclass{article}
\usepackage[german]{babel}
\usepackage{myPage}
\def\Datum{\today}
\def\Version#1{{\bf #1}}
\begin{document}
\author{Prof. Dr. Anne Bruggemann-Klein \and
Dr. Angelika Reiser \and
Prof. Dr. Doris Schmitt-Landsiedel}
\title{Ein Computer-Propädeutikum
von Studentinnen für Studentinnen
(Konzept vom \Datum, Version \Version{1.3})}
\maketitle
\section{Hintergrund und Ziele}
Gerade in technischen Studiengängen spielen
Computer als alltägliches Arbeitsmittel eine wichtige Rolle.
Es ist wesentlich, dass die Studierenden
gleich zu Beginn des Studiums einen vertrauten und sicheren Umgang
mit diesem Handwerkszeug haben. Eventuell vorhandene
Hemmschwellen müssen sobald wie möglich abgebaut werden;
davon ist eine studienzeitverkürzende Wirkung zu erwarten.
Es ist bekannt, dass Schülerinnen weniger Zugang zu einem Computer
haben als Schüler und weniger häufig einen eigenen Computer besitzen.
Das Computer-Propädeutikum richtet sich deshalb primär an
Studentinnen, die dadurch eventuell fehlende Vorerfahrungen
ausgleichen können.
\section{Das Kursangebot des Computer-Propädeutikums}
Das Computer-Propädeutikum bietet Kurseinführungen
mit praktischen Übungen für das Studium an
der TU München relevante Themen rund um den Computer.
Jeder Kurs dauert als Block angeboten drei Stunden
und beinhaltet eine tutorielle Einführung in das Thema,
praktische Übungen am Rechner und eine
Abschlussbesprechung zu den Übungen. Kursthemen können sein:
\begin{itemize}
\item Textverarbeitung mit Word
\item Textverarbeitung mit LaTeX
\item Textverarbeitung mit Star Office
\item E-Mail
\item Browsen und Informationsnutzung im World-Wide Web
\item Erstellen von Informationsangeboten im World-Wide Web (HTML)
\item Betriebssystem, Dateiorganisation
\item Gestaltung des Desktops (Window-Manager)
\item ASCII-Editoren (vi, emacs)
\item Kauf eines PCs
\item Datenverbindungen von zu Hause (analog, ISDN)
\item Fakultätsspezifische Informationsinfrastrukturen
\end{itemize}
Jeder Kurs sollte sich auf ein enges Thema konzentrieren.
Die Themenliste ist offen für Ergänzungen.
Kriterium ist, dass der
Kursinhalt propädeutischer Natur für das Studium an der TU ist
und eine Form der Computernutzung zum Inhalt hat.
```

Ein Kurs kann fakult"ats"ubergreifend oder fakult"atsspezifisch sein. Die Kurse sind in der Regel auf eine spezielle Rechnerplattform (PC/Windows, Unix) zugeschnitten.

\section{Die Zielgruppe des Computer-Prop"adeutikums}

Das Kursangebot richtet sich an Studentinnen im ersten Semester. Wird ein Angebot von dieser Zielgruppe nicht ausgesch"opft, steht es auch Studentinnen h"oherer Semester und, in dritter Priorit"at, m"annlichen Studierenden offen.

\section{Die Leiterinnen und Leiter des Computer-Prop"adeutikums}

Die Kurse werden vorzugsweise von Studentinnen h"oherer Semester konzipiert und durchgef"uhrt. Zur Erg"anzung des Kursangebots k"onnen auch Mitarbeiterinnen und Mitarbeiter oder Studenten h"oherer Semester Kurse anbieten. Nur von Studierenden angebotene Kurse k"onnen im Rahmen des Programms finanziert werden. Die Kursleiterinnen und -leiter sind selbst f"ur die Organisation von Seminar- und Rechnerr"äumen und von Zugangsm"oglichkeiten zu den R"äumen am Wochenende f"ur ihren Kurs verantwortlich. Wir gehen davon aus, da"s die Studiendekaninnen und -dekane sie bei der Organisation unterst"utzen.

\section{Durchf"uhrung}

Das Computer-Prop"adeutikum ist eine Initiative der Frauenbeauftragten der TU M"unchen und wird vom Frauenb"uro (Kerstin Hansen, Anja Quindeau) organisiert. Die TU M"unchen finanziert das Programm aus dem Tutorienprogramm zur Verk"urzung der Studiendauer. F"ur von Studierenden angebotene Kurse wird ein Werksvertrag abgeschlossen, der die Konzeption, Vorbereitung und Durchf"uhrung eines dreist"undigen Kurses beinhaltet. Konzeption und Vorbereitung eines Kurses werden mit 200 DM und jede Durchf"uhrung mit 100 DM honoriert. Das Frauenb"uro erh"alt Hilfskraftmittel aus dem Tutorienprogramm f"ur die Organisation des Computer-Prop"adeutikums. Wir rechnen mit folgenden Kosten: 20 Kurse, jeder zweimal durchgef"uhrt, zu 20 mal 200 DM plus 20 mal 2 mal 100 DM, also 8000 DM, plus 1000 DM f"ur die Organisation im Frauenb"uro. Eine erste Ausschreibung des Programms ist bereits erfolgt. Interessierte Kursleiterinnen und -leiter k"onnen ihr Kursangebot bis zum 5.11.1999 abgeben. Bitte benutzen Sie dazu unser Formular. Die ersten Kurse sollen noch im November 1999 stattfinden. Bei Bedarf erfolgt eine zweite Ausschreibung, zu der neue Kursangebote abgegeben werden k"onnen, Anfang Dezember 1999. Die Ausschreibung des Programms, die Gestaltung des Kursangebots sowie die Ank"undigung der Kurse erfolgt durch das Frauenb"uro in Zusammenarbeit mit den Studierendenvertretungen, den Studiendekanen und den Frauenbeauftragten. Die Vertr"age werden nach Absprache mit dem Frauenb"uro von der Hochschulverwaltung abgeschlossen. Die Kurse sollen zu Randzeiten (Abends, Freitags Nachmittags oder am Wochenende) stattfinden.

\end{document}



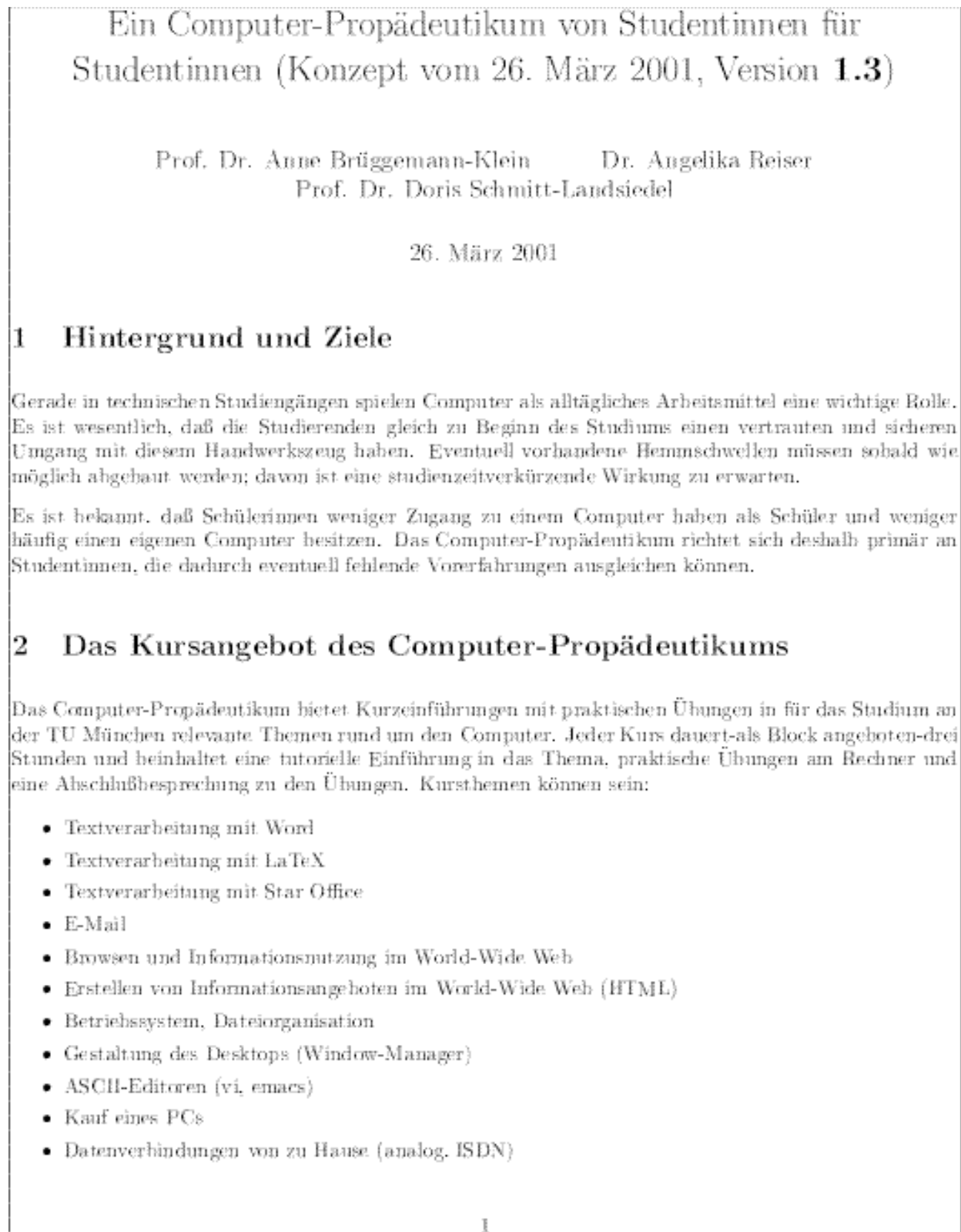
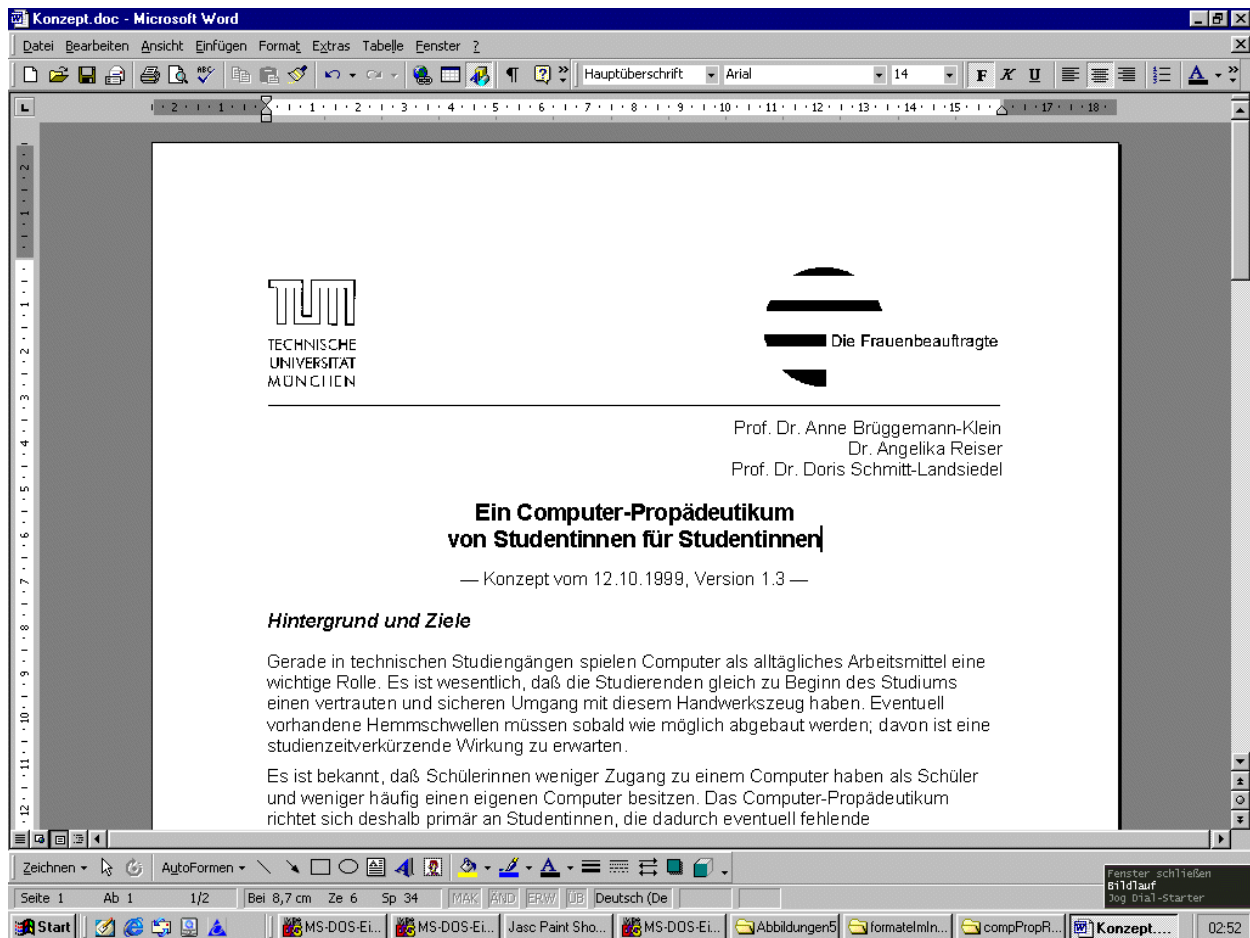
**Abbildung A.6. Eine LaTeX-formatierte Version des Dokuments**

Abbildung A.7. Ein Beispieldokument im Word-System



---

# Anhang B. Appendix: Primitive Datentypen<sup>1</sup>

string

boolean

decimal

float

double

duration

dateTime

time

date

gYearMonth

gYear

gMonthDay

gDay

gMonth

hexBinary

Base64Binary

anyURI

QNAME

NOTATION

xs:untyped

xs:untypedAtomic

xs:dayTimeDuration

xs:yearMonthDuration

---

<sup>1</sup>Siehe [BMP04] und [CR99-1]

---

# Anhang C. Appendix: Built-In Funktionen<sup>1</sup>

abs, add-dayTimeDuration-to-date, add-dayTimeDuration-to-time, add-dayTimeDurations, add-yearMonthDuration-to-date, add-yearMonthDuration-to-dateTime, add-yearMonthDurations, adjust-date-to-timezone, adjust-dateTime-to-timezone, adjust-time-to-timezone, avg, base-uri, base64Binary-equal, boolean, boolean-equal, boolean-greater-than, boolean-less-than, ceiling, codepoint-equal, codepoints-to-string, collection, compare, concat, concatenate, contains, count, current-date, current-dateTime, current-time, data, date-equal, date-greater-than, date-less-than, dateTime, dateTime-equal, dateTime-greater-than, dateTime-less-than, day-from-date, day-from-dateTime, days-from-duration, dayTimeDuration-greater-than, dayTimeDuration-less-than, deep-equal, default-collation, distinct-values, divide-dayTimeDuration, divide-dayTimeDuration-by-dayTimeDuration, divide-yearMonthDuration, divide-yearMonthDuration-by-yearMonthDuration, doc, doc-available, document-uri, duration-equal, empty, encode-for-uri, ends-with, error, escape-html-uri, exactly-one, except, exists, false, floor, gDay-equal, gMonth-equal, gMonthDay-equal, gYear-equal, gYearMonth-equal, hexBinary-equal, hours-from-dateTime, hours-from-duration, hours-from-time, id, idref, implicit-timezone, in-scope-prefixes, index-of, insert-before, intersect, iri-to-uri, is-same-node, lang, local-name, local-name-from-QName, lower-case, matches, max, min, minutes-from-dateTime, minutes-from-duration, minutes-from-time, month-from-date, month-from-dateTime, months-from-duration, multiply-dayTimeDuration, multiply-yearMonthDuration, name, namespace-uri, namespace-uri-from-QName, nilled, node-after, node-name, normalize-space, normalize-unicode, not, NOTATION-equal, number, numeric-add, numeric-divide, numeric-equal, numeric-greater-than, numeric-integer-divide, numeric-less-than, numeric-mod, numeric-multiply, numeric-subtract, numeric-unary-minus, numeric-unary-plus, one-or-more, position, QName, QName-equal, remove, replace, resolve-QName, resolve-uri, reverse, root, round, round-half-to-even, seconds-from-dateTime, seconds-from-duration, seconds-from-time, starts-with, static-base-uri, string, string-join, string-length, string-to-codepoints, subsequence, substring, substring-after, substring-before, substring-before, subtract-dates, subtract-dateTimes, subtract-dayTimeDuration-from-date, subtract-dayTimeDuration-from-dateTime, subtract-dayTimeDuration-from-time, subtract-dayTimeDurations, subtract-times, subtract-yearMonthDuration-from-date, subtract-yearMonthDuration-from-dateTime, subtract-yearMonthDurations, sum, time-equal, time-greater-than, time-less-than, timezone-from-date, timezone-from-dateTime, timezone-from-time, to, tokenize, trace, translate, true, union, unordered, upper-case, year-from-date, year-from-dateTime, yearMonthDuration-greater-than, yearMonthDuration-less-than, years-from-duration, zero-or-one

---

<sup>1</sup>Siehe [MMW07-1]

---

## B. Literaturverzeichnis

- [AFQ89] J. André, R. Furuta, V. Quint. (1989). *Structured Documents*. The Cambridge Series on Electronic Publishing. Cambridge University Press, Cambridge.
- [BD04] B. Brügge, A. H. Dutoit. (2004). *Objektorientierte Softwaretechnik mit UML, Entwurfsmustern und Java*. Pearson Studium.
- [BKST07] A. Brüggemann-Klein, T. Schöpf, K. Toni. (2007). *Principles, Patterns and Procedures of XML Schema Design – Reporting from the XBlog Project*. In: Extreme Markup Languages Conference - <http://www.idealliance.org/papers/extreme/proceedings/html/2007/BruggemannKlein01/EML2007BruggemannKlein01.html>.
- [C02] R. L. Costello. (2008). *XML Schemas: Best Practices*. <http://www.xfront.com/BestPracticesHomepage.html>.
- [DW00] M. Davis, K. Whistler. (2000). *Character encoding model*. Unicode Technical Report 17-3.1, Unicode, Inc.
- [DYIWFT02] M. J. Dürst, F. Yergeau, R. Ishida, M. Wolf, A. Freytag, and T. Texin. (2002). *Character model for the World Wide Web 1.0*. W3C Working Draft.
- [E63] D. Engelbart. (1963). *A conceptual framework for the augmentation of man's intellect*. In P. W. Howerton and D. C. Weeks, editors, *Vistas in Information Handling, volume 1, pages 1–29*, Washington, DC. Spartan Books.
- [FSS82] R. Furuta, J. Scofield, A. Shaw. (1982). *Document formatting: Survey, concepts, and issues*. Computing Surveys, 14(3):417–172.
- [GHJV95] E. Gamma, R. Helm, R. Johnson, J. Vlissides. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc.
- [HM04] E.R. Harold, W.S. Means. (2004). *XML in a Nutshell*. O'Reilly Media, Inc.
- [L86] L. Lamport. (1986). *LATEX: A Document Preparation System, User's Guide & Reference Manual*. Addison-Wesley Publishing Company, Reading, MA.
- [M02] E. Maler. (2002). *Schema Design Rules for UBL...and Maybe for You*. In: XML 2002 Conference.
- [MVD82] N. Meyrowitz, A. van Dam. (1982). *Interactive editing systems (Parts I and II)*. Computing Surveys, 14(3):321–415.
- [MS06] A. Møller, M. I. Schwartzbach. (2006). *An Introduction to XML and Web Technologies*. Addison-Wesley.
- [N00] T. H. Nelson. (2000). *Embedded markup considered harmful*. Published on XML.com (<http://www.xml.com/pub/a/w3j/s3.nelson.html>).
- [P08] D. Pagano. (2008). *Modellierung und Definition von XML-Anwendungen mit UML und XML Schema*. Fakultät für Informatik der Technischen Universität München.
- [RTW93] D. R. Raymond, F. W. T. Tompa, D. Wood. (1993). *Markup reconsidered*. Technical Report TR-356, Computer Science Department, University of Western Ontario. <http://www.csd.uwo.ca/tech-reports/356>.
- [RW99] D. A. Reid, P. Weverka. (1999). *Word 2000: The Complete Reference*. Osborne/McGraw-Hill, Berkeley.
- [VV02] E. van der Vlist. (2002). *XML Schema*. O'Reilly Media, Inc.
- [V05] H. Vonhoegen. (2005). *Einstieg in XML*. Galileo Computing.
- [BCFFRS07] Scott Boag, Don Chamberlin, Mary F. Fernandez, Daniela Florescu, Jonathan Robie, Jerome Simeon. *XQuery 1.0 Local Functions Namespace, referenziert auf <http://www.w3.org/TR/xquery/>*. W3C Recommendation: 2007, <http://www.w3.org/2005/xquery-local-functions/>.

- [BK07] Prof. Dr. Anne Brüggemann-Klein. (2007). *XML-Praktikum (Folien E\_XSLT.ppt)*. Technische Universität München.
- [BM06] Jim Melton and Steven Buxton. (2006). *Querying XML (Kapitel 3.2, Seite 46)*. Morgan Kaufmann, Elsevier: ISBN 1-55860-711-0.
- [BMP04] Paul V. Biron, Kaiser Permanente, Ashok Malhotra. *XML Schema Part 2: Datatypes Second Edition*. W3C Recommendation: 2004, <http://www.w3.org/TR/xmlschema-2/#built-in-primitive-datatypes>.
- [C99] James Clark. *XSL Transformations (XSLT) Version 1.0*. W3C Recommendation: 1999, <http://www.w3.org/TR/xslt>.
- [CR99-1] James Clark, Steve De Rose. *XML Path Language (XPath) Version 1.0, 5. Data Model*. W3C Recommendation: 1999, <http://www.w3.org/TR/xpath-datamodel/#types>.
- [CR99-2] James Clark, Steve De Rose. *XML Path Language (XPath) Version 1.0, 5. Data Model*. W3C Recommendation: 1999, <http://www.w3.org/TR/xpath-datamodel/#document-order>.
- [CT04] John Cowan, Richard Tobin. *XML Information Set (Second Edition)*. W3C Recommendation: 2004, <http://www.w3.org/TR/xml-infoset/>.
- [eXDB01] *eXist, Open Source Native XML Database, (aktuelle Version 1.2.5)*. <http://www.exist-db.org/>.
- [eXDB02] *eXist, Open Source Native XML Database, Documentation, Specifying the Input Document Set*. <http://exist.sourceforge.net/xquery.html#N10263>.
- [eXDB03] *eXist, Open Source Native XML Database, Documentation, Manipulating Database Contents*. <http://exist.sourceforge.net/xquery.html#N10474>.
- [G05] Joris Petrus Maria Graumans. (2005). *Usability of XML Query Languages*. Universität Utrecht: <http://igitur-archive.library.uu.nl/dissertations/2005-1018-200002/full.pdf>.
- [K09-1] Michael Kay. (2009). *XSLT and Xquery Procesing (aktuelle Version 9.1.0.6)*. Saxonica.com, <http://www.saxonica.com/index.html>.
- [K09-2] Michael Kay. (2009). *XSLT and Xquery Procesing, Saxon Documentation, Using XQuery*. Saxonica.com, <http://www.saxonica.com/documentation/using-xquery/intro.html>.
- [K09-3] Michael Kay. *XSL Transformations (XSLT) Version 2.0*. W3C Recommendation: 2007, <http://www.w3.org/TR/xslt20>.
- [M09] Dimitar Menkov. (2009). *Master's Thesis in Informatik: Entwicklung eines Tutorials für XQuery*. Technische Universität München (Betreuer: Prof. Dr. Anne Brüggemann-Klein).
- [MM07] Jim Melton, Subramanian Muralidhar. *XML Syntax for XQuery 1.0 (XQueryX)*. W3C Recommendation: 2007, <http://www.w3.org/TR/xqueryx>.
- [MMW07-1] Ashok Malhorta, Jim Melton, Norman Walsh. *XQuery 1.0 and XPath 2.0 Functions and Operators*. W3C Recommendation: 2007, <http://www.w3.org/TR/xpath-functions/#quickref-alpha>.
- [MMW07-2] Ashok Malhorta, Jim Melton, Norman Walsh. *XQuery 1.0 and XPath 2.0 Functions and Operators*. W3C Recommendation: 2007, <http://www.w3.org/TR/xpath-functions>.
- [S01] Kimbro Staken. (2001). *Introduction to Native XML Databases*. <http://www.xml.com/pub/a/2001/10/31/nativexml.html>.
- [W07-1] Priscilla Walmsley. (2007). *XQuery (Kapitel 2, Seite 21)*. O'Reilly: ISBN 0-596-00634-9.
- [W07-2] Priscilla Walmsley. (2007). *XQuery (Kapitel 5, Seite 69)*. O'Reilly: ISBN 0-596-00634-9.
- [W07-3] Priscilla Walmsley. (2007). *XQuery (Kapitel 6, Seite 72)*. O'Reilly: ISBN 0-596-00634-9.

[W07-4] Priscilla Walmsley. (2007). *XQuery (Kapitel 17, Working with Strings, Seite 223)*. O'Reilly: ISBN 0-596-00634-9.

[W07-5] Priscilla Walmsley. (2007). *XQuery (Kapitel 6, Joins, Seite 81)*. O'Reilly: ISBN 0-596-00634-9.

[W07-6] Priscilla Walmsley. (2007). *XQuery (Kapitel 3, Node Comparissons, Seite 34)*. O'Reilly: ISBN 0-596-00634-9.