Lutz Wegner

Einführung in XML

Universität Kassel

```
<?xml version="1.0"?>
<purchaseOrder orderDate="1999-10-20">
 <shipTo country="US">
  <name>Alice Smith</name>
  <street>123 Maple Street</street>
  <city>Mill Valley</city>
  <zip>90952</zip>
 </shipTo>
 <items>
  <item partNum="872-AA">
   <productName>Lawnmower</productName>
   <quantity>1</quantity>
   <price>148.95</price>
  </item>
 </items>
</purchaseOrder>
```

SKRIPTEN DER PRAKTISCHEN INFORMATIK

Einführung in XML

Lutz Wegner

Einführung in XML

Skriptum zur gleichnamigen Vorlesung
an der Universität Kassel

unter Mitarbeit von
Morad Ahmad und Kai Schweinsberg

Auflage Oktober 2014

Skriptum zur gleichnamigen Vorlesung an der Universität Kassel im Wintersemester 2014/2015.

Prof. Dr. Lutz Wegner
Universität Kassel
FB 16 – Elektrotechnik/Informatik
Wilhelmshöher Allee 73
D-34109 Kassel
wegner@db.informatik.uni-kassel.de

Vorwort

Vorwort zur Auflage 2005

In dieser zweistündigen Vorlesung (mit zwei SWS Übung) sollen die Grundlagen der *eXtensible Markup Language*, die sich als Datenaustauschsprache etabliert hat, erläutert werden. Im Gegensatz zu HTML erlaubt XML die semantische Anreicherung von Dokumenten. Neben XML sollen auch die *eXtensible Stylesheet Language* (XSL), *XML Schema*, die DOM-Schnittstelle, *XPath*, *XQuery* und andere Komponenten besprochen werden.

Das vorliegende Skript wurde erstmals zum WS 2004/05 in dieser Form herausgegeben. Änderungen können im Laufe der Veranstaltung noch einfließen. Als Grundlage diente mein älteres Skript aus dem Jahre 2000 und die Überarbeitung und Erweiterung von Dr. Morad Ahmad aus dem WS 2003/04.

Da XML eine relativ neue Technologie darstellt und die Standards z. T. noch in Entwicklung sind, bzw. täglich neue Software-Werkzeuge angeboten werden, bin ich für Hinweise und die Mitwirkungen der Studenten bei der Fehlersuche dankbar. Alle Mängel gehen zu meinen Lasten.

Kassel, im Oktober 2005

Lutz Wegner

Inhalt

1 Einführung

Die Entwicklung von XML [1] begann Ende der 90er Jahre. Hierzu schlossen sich mehrere Organisationen und führende Unternehmen der Softwareindustrie zusammen, um einen Standard für die Entwicklung von Markup-Sprachen zu definieren. XML selbst und viele darauf aufbauende Standards wurden vom *World Wide Web Consortium* (W3C)[1] verabschiedet.

Bevor wir mit der Einführung in XML starten, sollten zuerst einige weit verbreitete, falsche Vorstellungen über XML ausgeräumt werden. XML ist keine *Markup*-Sprache, sondern eine *Metabeschreibungssprache*, d. h. eine Sprache für die Definition anderer Markup-Sprachen wie etwa HTML. Dabei gilt XML als Vereinfachung der *Standard Generalized Markup Language* (SGML). Damit ist das zweite, häufig zu hörende Missverständnis ausgeräumt: XML wird HTML nicht ersetzen. Vielmehr ist das neuere XML-basierte XHTML eine „sauberere“ Version von HTML.

Die Verwendung von XML betrifft zwei große Bereiche. Als erstes lässt sich XML für die Entwicklung von unterschiedlichen Beschreibungssprachen einsetzen. Vorhandene Beispiele dafür sind z. B. HTML, SVG [2] und MathML [3]. Der zweite Anwendungsbereich liegt in der Definition von Austauschformaten zwischen verschiedenen Anwendungen in heterogenen Netzen. Hierzu dienen die Standards SOAP [4] [5] und XML-RPCs [6].

1. Siehe `http://www.w3.org`.

1.1 Begriffe und Definitionen

In dieser Vorlesung geht es um XML selbst sowie einige verwandte Technologien. Damit die vielen Abkürzungen und neuen Begriffe keine zu große Verwirrung stiften, folgt hier eine kurze Erläuterung der Terminologie:

- **Markup und Markup-Sprachen**
  „Markups sind Informationen, die einem Dokument hinzugefügt werden, um auf bestimmte Weise dessen Bedeutungsinhalt zu erweitern, indem es die einzelnen Teile kennzeichnet und festlegt, wie diese zueinander in Beziehung stehen. Eine *Markup-Sprache* ist eine Menge von Symbolen, die im Text des Dokuments plaziert werden können, um einzelne Teile dieses Dokuments zu benennen und sie voneinander abzugrenzen“ [7].
- **DTD und XML Schema**
  Neben der Einhaltung der Syntaxvorschriften der Markup-Sprache selbst kann man verlangen, dass Dokumente eigendefinierten Strukturregeln genügen. Dazu wurde in XML zuerst die sog. *Document Type Definition* (DTD) von SGML übernommen. Da DTDs einige Nachteile aufweisen (dazu später), wurde ein neuer XML-basierter Standard für die Modellierung eines Dokuments entwickelt: *XML Schema Language* [8].
- **XSL**
  Die *Extensible Stylesheet Language (XSL)* [9] besteht aus drei Hauptteilen: 1. *XSL Transformation Language* (XSLT [10] [11]) dient zur Definition von Transformationen von XML-Dokumenten in andere XML-basierte Sprachen. 2. *Formating Objects* (fo) bildet eine ausgereifte Formatierungsprache, die in ihrer Funktion etwa PDF ähnelt. 3. *XPath* [12] ist eine von XSLT verwendete Sprache für die Adressierung von Elementen innerhalb eines XML-Dokuments.
- **XLink**
  XML enthält keine festgelegten Elemente, d. h. auch keine speziellen Link-Elemente wie etwa das `A`-Element in HTML. Die *XML Linking Language* (XLink) ist eine Sprache zur Definition von Link-Elemen-

ten in einer Markup-Sprache. XLink verwendet dazu die beiden Sprachen XPath und XPointer.

- **XPointer**
  Die *XML Pointer Language* (XPointer [13]) ist der Teil von XLink zur Identifizierung von beliebigen Teilen eines Dokuments. XPointer basiert auf XPath und unterstützt Adressierungen bis in die internen Strukturen von XML-Dokumenten hinein.

- **DOM**
  Das *Document Object Model* (DOM) ist eine baumartige interne Darstellung eines XML-Dokuments (oder auch anderer Dokumentarten). Dabei definiert der DOM-Standard nicht, wie dies von einer Implementierung realisiert wird. Vielmehr wird eine *DOM-Schnittstelle* für die Bearbeitung eines Dokuments festgelegt. DOM ist in sog. Levels unterteilt. Der DOM-Level 1 [15] behandelt XML-Dokumente im allgemeinen. DOM-Level 2 [16] enthält weitere Teile für spezielle XML-Dokumente, etwa HTML oder SVG. Mithilfe der DOM-Schnittstelle können Anwendungen XML-Dokumente dynamisch verändern und haben einen wahlfreien, nicht sequentiellen Zugriff auf den Inhalt eines Dokuments.

- **XML-Parser**
  Ein XML-Parser ist ein Programm, das XML-Dokumente auf syntaktische Korrektheit überprüft. Ein sog. validierender XML-Parser validiert zusätzlich ein XML-Dokument anhand einer DTD oder eines XML Schemas. Meistens stellen XML-Parser weitere Funktionen zur Verfügung, z. B. das Lesen eines Dokuments als DOM-Objekt. Einige bekannte XML-Parser sind XT [18] und Xerces [19]. Xerces kann über Schnittstellen verschiedener Programmiersprachen angesprochen werden.

- **Stylesheet-Prozessor**
  Ein Stylesheet-Prozessor ist ein Programm, dass XML-Dokumente anhand der Angaben eines Stylesheets transformiert. Beispiel eines Stylesheet-Prozessors ist Xalan [20].

1.2 Warum XML?

Natürlich stellt sich bei der Einführung einer neuen Technologie die Frage nach dem Sinn und den mit der Einführung verbundenen Zielen. Zusammengefasst können für XML die folgenden Vorteile bzw. Ziele genannt werden:

- Durch XML können Auszeichnungssprachen anwendungsspezifisch, d. h. maßgeschneidert für die Bedürfnisse einer Anwendung, entwickelt werden. Dabei kann einem Auszeichnungselement eine semantische Bedeutung zukommen.
- XML-Dokumente müssen eindeutige Regeln und Strukturen einhalten, sodass ihre Verarbeitung durch Anwendungsprogramme einfacher und sicherer geschehen kann als z. B. die Verarbeitung von HTML-Dokumenten.
- Inhalt und Darstellung eines Dokuments werden voneinander getrennt, sodass unterschiedliche Darstellungen desselben Dokuments möglich sind. Somit können z. B. mehrere geräteabhängige Visualisierungen aus einem einzigen Dokument erzeugt werden.
- XML-Dokumente sind einfache, aber wohl strukturierte Textdokumente. Damit sind sie sowohl von menschlichen Lesern als auch von Anwendungsprogrammen einfach zu interpretieren und zu verarbeiten.
- XML geht mit Fehlerüberprüfung strenger um als z. B. HTML. Erfüllen Dokumente einige der Grundvoraussetzungen an die Syntaxregeln nicht, so muss ein *XML-Parser* das Dokument abweisen und eine entsprechende Fehlermeldung produzieren.

1.3 Entwicklung von Dokumenten

XML-Dokumente werden auf verschiedene Arten erzeugt. Sie können die Ausgabe von Anwendungsprogrammen sein, sie können durch spezielle grafische Editoren erzeugt werden[1] oder durch einfache Texteditoren per Hand eingetippt werden.

---

1. Siehe z. B. XMLSpy unter `http://www.xmlspy.softjam.co.uk`.

Da wir in dieser Vorlesung den Aufbau von XML-Dokumenten, Stylesheets, DTDs und Schematas erlernen wollen, bietet sich die Verwendung eines einfachen Texteditors an. (Sie haben bestimmt einen Lieblingseditor!). Dies hat weiterhin den Vorteil, sich nicht auf eine bestimmte Plattform oder Software festlegen zu müssen, auch wenn ein reiner Texteditor bei der Entwicklung größerer XML-basierter Projekte eher zu mühsam wäre.

In der Regel definiert man eine Sprache durch Festlegung auf Sprachregeln. Im nächsten Schritt kann man für Dokumente fordern, dass sie gewissen Formvorschriften genügen. Das geschieht mittels einer DTD oder eines XML Schemas. Zur Überprüfung eines Dokuments auf syntaktische Korrektheit bezüglich eines Schemas (zur sog. Validierung) wird ein XML-Parser aufgerufen. Dieser gibt in der Regel bei Fehlern detaillierte Fehlermeldungen mit Zeilennummern aus. Diese Fehler können dann mit einem Texteditor korrigiert werden, bis das Dokument fehlerfrei ist. Bei der Entwicklung eines Stylesheets geht man ähnlich vor. Zum Testen des Stylesheets kann ein Stylesheet-Prozessor verwendet werden, der eine Ausgabe in der Zielsprache erzeugt.

1.4 Publizieren von XML-Dokumenten

Für das Publizieren von XML-Dokumenten lassen sich drei Strategien verwenden:

- Das XML-Dokument wird mit dem Stylesheet an einen Client übertragen. Der Client (Browser) übersetzt das XML-Dokument in die Zielsprache (z. B. HTML) und stellt das Dokument dar.
- Das XML-Dokument wird dynamisch bei einer Anfrage auf der Serverseite mithilfe eines Stylesheet-Prozessors in die Zielsprache transformiert und an den Client übertragen.
- Das XML-Dokument wird statisch transformiert. Der Client fordert die transformierten Dokumente an.

Die dynamische Erzeugung von XML-Dokumenten auf der Serverseite scheint sich durchzusetzen, da diese die meisten Vorteile von XML vereint. Es werden damit keine Voraussetzungen an den Client gestellt. Wei-

terhin können vom Server verschiedene Ausgaben produziert werden, welche die Leistungsfähigkeit des jeweiligen Clients berücksichtigen.

Im einfachsten Fall kann man dazu *CGI-Skripte* verwenden. In der Praxis nutzt man oft PHP für diesen Zweck. Es existieren aber auch spezielle Umgebungen, die diese Aufgabe übernehmen. Sehr bekannt in diesem Bereich ist *Cocoon* von *Apache*. Falls es die Zeit zulässt, werden wir uns am Ende dieser Vorlesung kurz mit Cocoon beschäftigen.[1]

1.5 Beispiel

Um den Prozess der Entwicklung von XML-Dokumenten zu demonstrieren, betrachten wir zuerst ein einfaches Beispiel. Dabei gehen wir hier nicht auf die Details ein, die wir dann in den folgenden Kapiteln behandeln werden.

Programm 1–1 *Teilnehmerliste („teilnehmer0.xml“)*

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<!DOCTYPE TeilnehmerS SYSTEM "teilnehmer0.dtd">
<TeilnehmerS vorlesung='Einführung in XML'>
  <Teilnehmer matrNr='4711'>
    <name>Möller</name>
    <vorname>Alex</vorname>
    <semester>6</semester>
    <fachb>16</fachb>
  </Teilnehmer>
  <Teilnehmer matrNr="4722">
    <name>Schmidt</name>
    <vorname>Hans</vorname>
    <semester>8</semester>
    <fachb>17</fachb>
  </Teilnehmer>
</TeilnehmerS>
```

Dieses Dokument besteht, wie jedes XML-Dokument überhaupt, aus Inhalten, die mit Markup-Symbolen durchsetzt sind. Die spitzen Klammern (<>) und die von ihnen umschlossenen Namen heißen *tags*[2]. Tags begrenzen und bezeichnen Teile des Dokuments, die wir Elemente nen-

1. Siehe `http://xml.apache.org`.
2. Alternativ zum engl. Begriff verwenden wir das deutsche Wort *Markierung*.

nen, und fügen weitere Informationen hinzu, die zur Definition der Struktur beitragen. Wie man sieht, unterscheidet sich der *End-Tag* vom *Start-Tag* durch den Schrägstrich („/“) als erstes Zeichen nach der öffnenden spitzen Klammer. Der Text zwischen den Tags ist der Inhalt (*Content*) des Elements. Zusammen bilden sie der Inhalt des Dokuments, also die eigentliche Information, etwa den Inhalt eines Briefes oder den Namen des Absenders. XML-Dokumente setzen sich grundsätzlich aus diesen beiden Teilen - Strukturangaben und Inhalt - zusammen.

Die Anweisung `<?xml version='1.0'?>` in der ersten Zeile ist eine sog. *XML-Deklaration* (*XML Processing Instruction*). Sie teilt einem XML-Prozessor mit, dass dieses Dokument ein XML-Dokument ist. Zusätzlich wird die verwendete *XML-Version* `1.0` und der Zeichensatz `ISO-8859-1` für westeuropäische Sprachen angegeben.

Die Tags `<TeilnehmerS>` und `<Teilnehmer>` enthalten weitere *Unterelemente*. Sie dienen also der Strukturierung des Dokuments. Das `name`-Element dagegen enthält einen Text als Inhalt. In XML können Elemente auch gemischt Unterelemente und Texte als Inhalt haben, z. B. `<i><b>XML</b> in der Praxis</i>`.

Die Angaben `matrNr='...'` und `version="..."` sind sog. *Attribute*. Ein Element kann mehrere Attribute enthalten, die weitere Informationen über das Element angeben. Die Attributwerte sind in Hochkommata (einfache oder doppelte) eingeschlossen. In unserem Beispiel gibt z. B. das Attribut `matrNr` die Matrikelnummer eines Teilnehmers an.

1.5.1 Ein einfaches Stylesheet

Um eine Visualisierung dieses Dokuments z. B. in Form einer HTML-Tabelle zu erzeugen, entwickeln wir ein XSLT-Stylesheet. Es definiert für jedes Element, welche Ausgaben in der Zielsprache produziert werden. Es können aber auch weitere XSLT-Anweisungen angegeben werden.

Programm 1–2 *Stylesheet zur Transformation („teilnehmer0.xsl“)*

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<xsl:stylesheet version='1.0'
    xmlns:xsl='http://www.w3.org/1999/XSL/Transform'>
```

```
 <xsl:template match='/'>
  <html>
   <head>
    <title>
     <xsl:value-of select='TeilnehmerS/@vorlesung'/>
    </title>
   </head>
   <body>
    <h1 align='center'>
     Teilnehmerliste:
     <xsl:value-of select='TeilnehmerS/@vorlesung'/>
    </h1>
    <xsl:apply-templates/>
   </body>
  </html>
 </xsl:template>

 <xsl:template match='TeilnehmerS'>
  <table border='1' align='center' cellpadding='5'>
   <tr>
    <th>Vorname</th>
    <th>Name</th>
    <th>Matrikelnummer</th>
    <th>Semester</th>
    <th>Fachbereich</th>
   </tr>
   <xsl:apply-templates/>
  </table>
 </xsl:template>

 <xsl:template match='Teilnehmer'>
  <tr>
   <td><xsl:value-of select='vorname'/></td>
   <td><xsl:value-of select='name'/></td>
   <td><xsl:value-of select='@matrNr'/></td>
   <td><xsl:value-of select='semester'/></td>
   <td><xsl:value-of select='fachb'/></td>
  </tr>
 </xsl:template>

</xsl:stylesheet>
```

Da XSLT-Dokumente XML-Dokumente sind, beginnt das Dokument mit der Anweisung `<?xml version="1.0">`. Das Element `<xsl:styles-`

heet ...> ist das oberste Element[1] in jedem XSLT-Stylesheet. Das Element <xsl:template> verarbeitet ein Element (oder eine Gruppe von Elementen) aus dem XML-Dokument und produziert eine Ausgabe in der jeweiligen Zielsprache (hier: HTML). Im Beispiel verarbeitet das erste <xsl:template> das Wurzelelement des Dokuments ('/'). Es erzeugt daraus die Ausgabe <html>...</html> und ruft zwischen den HTML-Anweisungen <xsl:apply-templates> auf, um die *Unterelemente* der Wurzel an dieser Stelle zu verarbeiten.

Durch den Aufruf von Xalan kann das XML-Dokument mithilfe des obigen Stylesheets in HTML transformiert werden (siehe Abbildung 1.1).

```
xalan -in teilnehmer0.xml -xsl teilnehmer0.xsl \
      -out teilnehmer0.html
```

Programm 1–3 *Teilnehmerliste in HTML („teilnehmer0.html“)*

```
<html>
<head>
<META http-equiv="Content-Type"
      content="text/html; charset=UTF-8">
<title>Einf&uuml;hrung in XML</title>
</head>
<body>
<h1 align="center">
    Teilnehmerliste: Einf&uuml;hrung in XML</h1>
<table cellpadding="5" align="center" border="1">
<tr>
<th>Vorname</th>
<th>Name</th>
<th>Matrikelnummer</th>
<th>Semester</th>
<th>Fachbereich</th>
</tr>

<tr>
<td>Alex</td><td>M&ouml;ller</td>
<td>4711</td><td>6</td><td>16</td>
</tr>
```

1. Wir verwenden ab sofort die Begriffe *Element* und *Tag* synonym, d.h. statt genauer vom „Element mit Start-Tag <xyz>“ zu sprechen, sagen wir „das Element <xyz>“.

```
<tr>
<td>Hans</td><td>Schmidt</td>
<td>4722</td><td>8</td><td>17</td>
</tr>

</table>
</body>
</html>
```

Dieses HTML-Dokument kann nun in einem beliebigen Browser betrachtet werden.

Der Internet Explorer bietet ab Version 5 umfassende XML-Fähigkeiten. Der eingebaute XML-Parser des Internet Explorers kann XML-Dokumente validieren und in einer übersichtlichen, baumartigen Struktur darstellen. Zusätzlich können XML-Dokumente mithilfe eines XSLT-Stylesheets in HTML transformiert und das Ergebnis im Browser dargestellt werden. Dazu ist in das XML-Dokument ein Verweis auf das verwendende Stylesheet einzubauen:

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<?xml-stylesheet type="text/xsl"
                 href="teilnehmer0.xsl"?>
<TeilnehmerS>
...
</TeilnehmerS>
```

Der Browser Mozilla kann mittlerweile ebenfalls XML-Dokumente mithilfe von CSS-Stylesheets anzeigen und XSL-Transformationen durchführen.

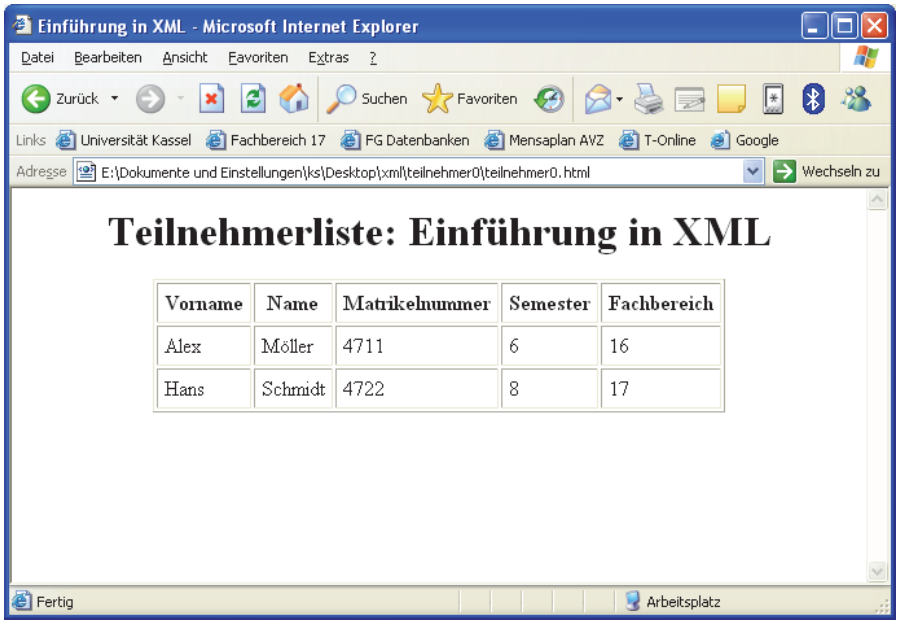


Abb. 1–1 *Darstellung als HTML-Tabelle*

1.5.2 Eine einfache DTD

Das obige XML-Dokument konnte vom Stylesheet-Prozessor bearbeitet und mithilfe eines Stylesheets in HTML übertragen werden. Es erfüllt also bestimmte *Grundvoraussetzungen* eines XML-Dokuments.[1] Doch welche Elemente kann man in einem solchen Dokument verwenden und welche Attribute haben sie? Anhand welcher *Grammatik* ist das Dokument aufgebaut? Zur Festlegung dieser Grammatik dienen sowohl *Document Type Definitions* (DTDs) als auch XML Schemata. Wir betrachten hier eine einfache DTD zur Definition von Dokumenten des Typs „Teilnehmerliste“.

1. Darauf gehen wir im nächsten Kapitel ein.

Programm 1–4 *DTD für Teilnehmerlisten („teilnehmer0.dtd“)*

```
<!ELEMENT TeilnehmerS (Teilnehmer*)>
<!ATTLIST TeilnehmerS vorlesung CDATA #REQUIRED>
<!ELEMENT Teilnehmer (name, vorname, semester, fachb)>
<!ATTLIST Teilnehmer matrNr NMTOKEN #REQUIRED>
<!ELEMENT name (#PCDATA)>
<!ELEMENT vorname (#PCDATA)>
<!ELEMENT semester (#PCDATA)>
<!ELEMENT fachb (#PCDATA)>
```

Zur Definition eines Elements dient die Deklaration

```
<!ELEMENT name definition>
```

Die Deklaration eines Attributes geschieht mit

```
<!ATTLIST elementname attributname definition>
```

Elemente mit Textinhalt werden mit dem Schlüsselwort `(#PCDATA)` angegeben (z. B. `name`). Elemente, die weitere Unterelemente enthalten, werden durch die Auflistung der erlaubten Unterelemente, getrennt durch Kommata, definiert (z. B. `Teilnehmer`).

Im XML-Dokument kann dann ein Verweis auf die verwendete DTD angegeben werden. Das geschieht wie folgt:

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<!DOCTYPE TeilnehmerS SYSTEM "teilnehmer0.dtd">
<TeilnehmerS>
...
</TeilnehmerS>
```

Bei kurzen DTDs bietet es sich an, diese direkt in das XML-Dokument wie folgt einzubauen:

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<!DOCTYPE TeilnehmerS [
<!ELEMENT TeilnehmerS (Teilnehmer*)>
...
]>
<TeilnehmerS>
...
</TeilnehmerS>
```

Zur Überprüfung, ob das Dokument den Angaben in der DTD genügt, in der XML-Terminologie also ein gültiges (*valides)*[1] XML-Dokument ist, verwenden wir den Xerces-XML-Parser. Er bietet für diesen Zweck unter anderem das Programm `Counter`.

```
kai@labserv:~$ alias xmlParse='java -cp \
/usr/share/doc/libxerces2-java-doc/examples dom.Counter \
-V -n -f'
kai@labserv:~$ xmlParse -v teilnehmer0.xml
teilnehmer0.xml: 250;35;0 ms (11 elems, 3 attrs,
                              33 spaces, 27 chars)
```

Um eine Fehlerausgabe zu produzieren, ändern wir in der drittletzten Zeile von Programm 1–1 das abschließende Tag `</fachb>` in `</fachB>` und rufen `xmlParse` erneut auf:

```
kai@labserv:~$ xmlParse -v teilnehmer1.xml
[Fatal Error] teilnehmer1.xml:14:14: The element type
"fachb" must be terminated by the matching end-tag
"</fachb>".
```

1.6 Zusammenfassung

Wir haben in diesem Kapitel einen ersten Eindruck von XML und den verwandten Technologien bekommen und dabei einige neue Begriffe und Definitionen kennengelernt. Dazu wurde ein erstes einfaches XML-Dokument entwickelt und mittels eines Stylesheets in HTML transformiert, womit die Entwicklung von XML-Dokumenten, XSLT-Stylesheets und DTDs veranschaulicht wurde.

In den nächsten Kapiteln gehen wir Schritt für Schritt auf diese Sprachen im Detail ein.

1. Im Gegensatz zu *wohlgeformt (well-formed)*, bei dem nur auf passende Start- und End-Tags geachtet wird.

2 Grundlagen

2.1 Aufbau von XML-Dokumenten

XML-Dokumente haben den folgenden Grundaufbau:

```
<?xml version='1.0' ...?>
<!DOCTYPE ...>
<rootElement attr="Wert" ...>

  <firstElement attr="Wert">
    Inhalt von firstElement
  </firstElement>

  <lastElement attr="Wert">
    Inhalt von lastElement
  </lastElement>

</rootElement>
```

Wir gehen im folgenden auf die einzelnen Teile dieses Dokuments ein.

2.1.1 Die XML-Deklaration

Die erste Zeile ist die sog. *XML-Deklaration*. Sie sollte in jedem XML-Dokument enthalten sein.[1] Sie kann die folgenden Attribute haben:

- **`version`**
  Spezifiziert die verwendete XML-Version im Dokument.
- **`encoding`**
  Definiert die im Dokument verwendete Zeichenkodierung. Einige ver-

1. Ab Version 1.1 des XML-Standards muss die XML-Deklaration angegeben werden.

breitete Zeichenkodierungen sind: US-ASCII, ISO-8859-1, UTF-8 und UTF-16. Falls `encoding` nicht angegeben wird, wird die Standard-Zeichenkodierung UTF-8 verwendet[1].

- **`standalone`**
  Sagt dem XML-Prozessor, ob irgendwelche andere Dateien geladen werden müssen.

Die Attribute „encoding“ und „standalone“ sind optional, „version“ muss angegeben werden.

Anmerkung: Die „Standalone Document Declaration“ ist als Hinweis für Applikationen gedacht, die ein Dokument vom XML-Prozessor erhalten und es nicht notwendigerweise vollständig parsen wollen. Der Hinweis besagt, ob externe „Markup Declarations“ existieren, die einzubinden wären, damit das Dokument „standalone“ vollständig zu interpretieren ist. Speziell betrifft das Default-Attribute und Entities (siehe unten), die im XML-Dokument benötigt werden und in einer externen DTD definiert werden. Ein validierender Parser prüft diese Notwendigkeit, wenn eine DTD angegeben ist, unabhängig davon ob `standalone` den Wert `"yes"` oder `"no"` hat! Der Standard weist darauf hin, dass man algorithmisch ein Dokument mit notwendigen externen Markup Declarations (`standalone="no"`) in ein solches mit bereits hereingeladenen Erweiterungen und der dann möglichen Angabe `standalone="yes"` wandeln kann. Falls externe Markup Declarations vorhanden sind, aber keine Standalone Document Declaration angegeben ist, wird `standalone="no"` angenommen.

2.1.2 Die Dokumenttyp-Deklaration

Falls eine DTD (Document Type Definition) verwendet werden soll, so muss diese deklariert werden. Die Instruktion kann gegenwärtig die zwei Formen

1. Zeichenkodierungen sind Teilbereiche vom Unicode-Zeichensatz, die mit 8 Bit dargestellt werden. Genauere Informationen kann man dem Dokument unter: `http://www.w3.org/Markup/html-spec/charset-harmful.html` entnehmen.

```
<!DOCTYPE root-element PUBLIC "name" "URL-of-DTD">
<!DOCTYPE root-element SYSTEM "URL-of-DTD">
```

annehmen.

Mit `PUBLIC` werden (in Zukunft!) öffentlich zugängliche DTDs spezifiziert. Diese sollen als registrierter Name in einem internen oder externen Repositorium bekannt sein. Schlägt die Ermittlung über `name` fehl, wird auf die folgende URL zugegriffen. Alternativ kann man sofort auf privat verfügbare DTDs zugreifen, dies geht mit der Variante `SYSTEM`.

Die Angabe `name` bei `PUBLIC` benutzt den sog. *Formal Public Identifier* (FPI)[1]. Diese Namenskonvention benutzt einen Namen mit vier Segmenten, die durch Doppelschrägstriche (slashes) getrennt sind, z. B.:

```
"-//O`Reilly//DTD//EN"
```

Das Minus deutet auf einen unregistrierten Namen hin, der ggf. nicht eindeutig ist. Ein Plus (+) wäre ein registrierter Eintrag. Das zweite Segment ist der Autor oder die Organisation, die den Namen veröffentlicht, das dritte Segment die Inhaltsform (hier DTD). Zuletzt folgt der Sprachcode, hier EN für Englisch.

2.1.3 Elemente

Elemente sind die Grundbausteine von XML-Dokumenten. In XML werden Elemente durch Tags in spitzen Klammern in der Form `<elementname> ... </elementname>` eingeklammert.

Der Elementname kann eine beliebige Anzahl von Buchstaben, Zahlen, Bindestrichen, Punkten und Unterstrichen enthalten, darf aber nicht mit Punkt, Bindestrich oder einer Ziffer beginnen.[2]

Das Doppelpunktzeichen (`:`) wird als Trenner für Namensräume verwendet (später in diesem Kapitel). Leerzeichen, Tabulatoren, Zeilenum-

1. ISO-8879
2. Sie müssen aber dabei beachten, dass jeder XML-Prozessor bzgl. der Länge des Namens Einschränkungen macht. Es existiert zwar keine Vorschrift, doch in der Praxis werden Elementnamen mit bis zu 40 Zeichen akzeptiert.

brüche, Gleichheitzeichen und Anführungszeichen sind *Separatoren* und dürfen deshalb in Element- und Attributnamen nicht verwendet werden. Sonderzeichen wie `&` oder `?` dürfen ebenfalls nicht in Elementnamen enthalten sein.[1]

Zwischen der ersten spitzen Klammer und dem Elementnamen darf kein Leerzeichen stehen (z. B. `< Buch>`), dagegen können beliebige Trennzeichen nach dem Elementnamen, und zwischen den Attributen stehen. Damit kann man ein Element auf mehreren Zeilen verteilen:

```
<Buch
  ISBN="X-3-89721-286-2" titel='Einführung in XML'>
  ...
</Buch>
```

Da XML Unicode als Zeichensatz verwendet, können Elementnamen in einer *beliebigen*, von Unicode unterstützten Sprache geschrieben werden.

Beispiele für gültige Elementnamen in den Start-Tags sind
`<_>`, `<_a>`, `<b4711>`, `<VON-BIS>`, `<Name.Vorname>`.

Beispiele für nicht gültige Elementnamen in Tags sind etwa
`<4711>`, `<.punkt>`, `<a&b>`, `<verstanden?>`.

Start- und Endtags

Ein Element besteht aus einem *Start-Tag*, einem *Inhalt* und einem *End-Tag*. Der Inhalt kann aus reinem Text bestehen, aus weiteren Unterelementen oder einer Mischung aus beidem. Die Ende-Markierung (End-Tag) wird durch `</elementname>` angegeben, und darf nicht, wie in HTML, weggelassen werden. Ist der Inhalt leer (*leeres Element*), so kann man Start- und End-Tag zu einem Tag zusammenziehen indem man ein '/'-Zeichen an das Ende des Tags vor die schließende spitze Klammer schreibt:

```
<leeresElement attr1='...' attrn='...'/>
```

1. Die genaue Beschreibung in EBNF-Form kann man in der Standardspezifikation nachlesen.

Alternativ kann man ein leeres Elemente aber auch mit abschließendem eigenen End-Tag wie üblich angeben. Daher sind `<br></br>` und `<br/>` äquivalente Schreibweisen.

Der Start- und End-Tag eines Elements müssen sich beide im gleichen Eltern-Element befinden. So ist z. B.
`<a>So <b>geht es</a> nicht</b>`
nicht erlaubt.

Da das Kleinerzeichen (<) für den Beginn eines Tags steht, darf es nicht direkt im Text verwendet werden. Dazu können die *vordefinierten Entities* für < (<) und für > (>) verwendet werden.

Sehr wichtig bei der Texteingabe ist darauf zu achten, dass XML alle Leerzeichen, Zeilenumbrüche und Tabulatorzeichen beibehält. Dies ist bei Programmiersprachen und anderen Auszeichnungssprachen, wo Leerzeichen in der Regel ignoriert werden (siehe `xml:space` weiter unten), anders.

2.1.4 Attribute

Mit Attributen kann man Elementen zusätzliche oder genauere Informationen hinzufügen. Man kann z. B. Elementen eindeutige Bezeichner durch ein Attribut geben, oder eine Eigenschaft über dieses Element beschreiben (Beispielsweise das `href`-Attribut für das `A`-Element in HTML).

Eine Attributangabe besteht aus einem Attributnamen, einem Gleichheitszeichen und einem Wert in Anführungszeichen. Dabei können einfache oder doppelte Anführungszeichen verwendet werden:
`attrname="Wert von attrname"` oder
`attrname='Wert von attrname'`.

Für Attributnamen gelten die gleichen Regeln wie für Elementnamen. Ein Element darf ein Attribut nur einmal enthalten.
`<x a="wert 1" a="wert 2"/>` ist nicht erlaubt.
Ein Element kann eine beliebige Anzahl von Attributen haben, solange jedes über einen eindeutigen Namen verfügt.

Reservierte Attribute

Die folgenden Attributnamen sind im XML-Standard mit einer festen Bedeutung reserviert:

- **`xml:lang`**
  Mit diesem Attribut wird die Sprache des Elements angegeben. Dies ist nützlich für die Erstellung von mehrsprachigen Dokumenten. Die Angabe verwendet einen zweibuchstabigen Sprachcode (Kleinschreibung wie `de`, `en` ist üblich). Ein weiterer *Qualifier* kann verwendet werden, um eine Sprach-Variante wie `en-US` zu spezifizieren. Vereinbarungsgemäß besteht dieser Qualifier aus zwei Großbuchstaben.
  Zur Definition von eigenen Sprachen kann der Sprach-Code `x` verwendet werden. Beispiele könnten etwa sein: `x-pascal`, `x-java` usw.
- **`xml:space`**
  `xml:space="default | preserve"`
  Bestimmt, ob die Whitespaces (Leerzeichen, Neue-Zeile, Tabulator) wie angegeben erhalten bleiben sollen (`preserve`) oder ob beliebige Ersetzungen erlaubt sind.
- **`xml:link`** und **`xml:attribute`**
  Diese beiden Attribute wurden ursprünglich im Zusammenhang mit Link-Elementen eingesetzt. Seit der Einführung des XLink-Namensraums sind sie veraltet und werden nicht mehr verwendet.

2.1.5 Namensräume

In XML-Dokumenten können Elemente aus verschiedenen Sprachen verwendet werden. Sie können z. B. eine mathematische Formel als MathML-Element in einem HTML-Dokument einbauen. Damit der XML-Prozessor (HTML-Browser) zwischen den Elementen beider Sprachen unterscheiden kann und diese unterschiedlich behandelt (z. B. das Plugin für die Darstellung von MathML aufruft), wurden in XML Namensräume eingeführt.

Bevor ein Namensraum verwendet werden kann, muss dieser zuerst im Dokument deklariert werden. Die Deklaration erfolgt in der Form eines Attributes innerhalb eines Elements:

`<namensraum:elementName xmlns:namensraum="url">`

Man kann in einem Dokument mehrere Namensräume verwenden. Der deklarierte Namensraum kann in dem Element selbst und allen Nachkommen genutzt werden. Zur Verwendung von Namensräumen werden sie durch ein *Namensraum-Präfix* gefolgt von einem Doppelpunkt gefolgt vom lokalen Namen des Elements angegeben.

Der Wert des `xmlns:`-Attributs ist eine URL, die gewöhnlich zu der Organisation gehört, die den Namensraum unterhält. Es ist jedoch nicht erforderlich, dass der XML-Prozessor diese URL benutzt. Die Angabe der URL dient lediglich der eindeutigen Bezeichnung des Namensraums. Durch die Wahl einer URL für diesen Zweck könnte jedoch eine eventuelle weitere Verwendung dieser Angabe in der Zukunft möglich sein.[1]

Im folgenden Beispiel werden die Namensräume `vorlesung` und `skript` deklariert:

```
<?xml version="1.0" standalone="yes"?>
<vorlesung:beispiel xmlns:vorlesung=
   "http://www.informatik.uni-kassel.de/~wegner/">
  <vorlesung:titel>Namensraum</vorlesung:titel>
  <vorlesung:code>4711-9999-XYZ</vorlesung:code>
  <vorlesung:anfang>
    Auch Zwerge haben mal klein angefangen
  </vorlesung:anfang>
  <skript:inhaltsverz xmlns:skript=
     'http://www.informatik.uni-kassel.de/skript/'>
    <skript:kap nr='1'>
      <skript:titel>Einführung</skript:titel>
    </skript:kap>
  </skript:inhaltsverz>
</vorlesung:beispiel>
```

Wir können einen Namensraum als *Standard-Namensraum* deklarieren, indem wir den Doppelpunkt (`:`) und den Namen im `xmlns`-Attribut weglassen. In unserem Beispiel können wir z. B. `vorlesung` als Standard-Namensraum benutzen:

---

1. Vorstellbar wäre das Validieren des Elements anhand einer DTD oder eines XML Schemas, die sich unter der angegeben URL befinden.

```
<?xml version="1.0" standalone="yes"?>
<beispiel xmlns=
   "http://www.informatik.uni-kassel.de/~wegner/">
  <titel>Namensraum</titel>
  <code>4711-9999-XYZ</code>
  <anfang>
    Auch Zwerge haben mal klein angefangen
  </anfang>
  <skript:inhaltsverz xmlns:skript=
     'http://www.informatik.uni-kassel.de/skript/'>
    <skript:kap nr='1'>
      <skript:titel>Einführung</skript:titel>
    </skript:kap>
  </skript:inhaltsverz>
</beispiel>
```

Die Verwendung von Namensräumen in Verbindung mit DTDs ist problematisch, und in dem jetzigen Standard noch nicht sauber gelöst. Möchten Sie ein Dokument mit Namensräumen anhand einer DTD validieren, so werden die Elemente aus den „fremden“ Namensräumen nicht etwa ignoriert, sondern müssen in der DTD definiert sein. Aus diesen und anderen Gründen sind Namensräume unter XML-Planern umstritten. Es ist aber klar, dass beide Konzepte, *Strukturerzwingung* und *Namensräume*, prinzipiell erforderlich sind und unter einen Hut gebracht werden müssen.

2.1.6 Wohlgeformte XML-Dokumente

Im Laufe der Entwicklung von HTML haben sich gewisse tolerierte Vereinfachungen und Unsauberkeiten eingeschlichen. So darf man bei Tabellen das schließende Element eines Tabellenfeldes `</TD>` weglassen, allerdings auf die Gefahr hin, dass das letzte Feld nicht richtig zugeordnet wird.

Dies ist in XML nicht zulässig. Alle Elemente müssen paarweise auftreten und "*wohlgeschachtelt*" (*properly nested*) sein, außer natürlich bei leeren Elementen wie etwa `<picture src='...' />`. Damit ist

```
<Abschnitt>
<Para>
Blablabla
<Para>
```

```
noch mehr bla
</Abschnitt>
```

ungültiges XML (es fehlen jeweils die `</Para>` Endmarken).

Genauso wird

```
<b><i>Achtung: gekauft wie gesehen</b></i>
```

zwar in HTML akzeptiert, in XML aber zurückgewiesen.

Andersherum ist ein leeres XML-Element in HTML eigentlich ein Fehler, z. B. die waagerechte Linie `<HR>` (horizontal ruler). In „XML-Form“ muss stattdessen `<HR/>` geschrieben werden.

Argumente von Attributen müssen in XML immer in Hochkommata eingeschlossen werden. HTML Browser erlauben auch das Weglassen, sofern der Wert keine Leerstellen enthält. Es sind sowohl einzelne als auch doppelte Hochkommata erlaubt.

Zuletzt sei darauf hingewiesen, dass HTML bekanntlich nicht auf Groß- und Kleinschreibung achtet, XML dagegen case sensitive ist! `<h2>Überschrift</H2>` wird deshalb als XML-Dokument zurückgewiesen, ginge in HTML aber durch.

Weiterhin ist zu beachten:

- Ein XML-Dokument besitzt ein eindeutiges *Wurzelelement*.
- Die Markierungszeichen `<` und `&` dürfen nicht als Textzeichen auftreten, sondern müssen durch sog. *Entity-Referenzen* ersetzt werden. Genauso muss `]]>` durch `]]>` ersetzt werden (`>` darf direkt angegeben werden).

Dokumente, die *korrekt geschachtelt* sind und die obigen Punkte beachten, sind *wohlgeformt* (*well-formed*). Genügt ein Dokument auch einer zugehörigen DTD, dann nennt der Standard es *gültig* (*valid*).

2.1.7 Prozessor-Instruktionen

Prozessor-Instruktionen (PI, *Processing Instructions*) sind dazu gedacht, Anweisungen an externe Anwendungen in einem XML-Dokument einzu-

bauen. Diese werden nur von der adressierten Anwendung verstanden und interpretiert. Sie werden von anderen Anwendungen ignoriert.

Eine PI sieht wie folgt aus

```
<?anwendung attr1='Wert 1' ...?>
```

Ein Beispiel für eine Prozessor-Instruktion ist der Hinweis auf die Verwendung eines Stylesheets:

```
<?xml-stylesheet type="text/xsl" href="filename.xsl"?>
```

Dies teilt einem Stylesheet-Prozessor mit, das XML-Dokument anhand des Stylesheets `filename.xsl` zu transformieren.

2.1.8 Entities

Ähnlich wie bei Makros lassen sich Zeichenfolgen mit einem Bezeichner versehen. Einige Standardvorgaben (z. B. `&`) werden von XML vordefiniert und müssen nicht gesondert angegeben werden. Andere können in einer DTD definiert werden.

Eine Entity *ename* wird in einem Dokument durch `&`*ename*`;` aufgerufen.

2.1.9 Kommentare

Kommentare werden in XML durch `<!-- ... -->` angegeben. In einem Kommentar können beliebige Texte vorkommen. Innerhalb eines Kommentars dürfen aber keine doppelten Bindestriche (*hyphens*) enthalten sein.

Kommentare dürfen überall im Dokument stehen, nur *nicht vor der XML-Deklaration und innerhalb von Tags*. Sie werden von einem XML-Prozessor ignoriert.

2.1.10 CDATA-Abschnitte

Mit dieser sog. *Character-Data* Sektion (nicht zu verwechseln mit der PCDATA-Angabe in DTDs) lassen sich Zeichenfolgen angeben, die der

Parser nicht interpretiert. Die Folge muss mit `<![CDATA[` anfangen und mit `]]>` aufhören. Als Beispiel könnte man schreiben

```
<![CDATA[In diesem XML-Kapitel besprechen wir
"Deklarationen", z. B. <?xml ...?> und <!DOCTYPE ...>,
CDATA[...]-Sektionen & vieles mehr
]]>
```

Andererseits sollte man innerhalb einer CDATA-Sektion keine Entity-Referenzen wie `&`, `<`, `>`, `&quot;` und `&apos;` verwenden, da diese nicht aufgelöst werden.

3 Document Type Definitions

3.1 Elementdeklarationen

Entschließt man sich, ein Dokumentenschema mittels der älteren, weit verbreiteten Document Type Definition (DTD) anzugeben, dann gilt: Jedes in einem gültigen XML-Dokument auftretende Element muss in der DTD deklariert sein. Dies geschieht mit der Elementdeklaration

```
<!ELEMENT elementname rule>
```

wobei der Elementname ohne spitze Klammern angegeben wird. Was hier als `rule` bezeichnet wird, legt fest, was zwischen der Start- und Endmarkierung des Elements stehen darf.

3.1.1 ANY und #PCDATA

Im einfachsten Fall erlaubt man mit einer Angabe

```
<!ELEMENT book ANY>
```

beliebige Daten, auch andere Tags.

Will man andererseits nur „Nutzdaten“ ohne Metazeichen (sog. *parsed character data*) erlauben, dann lautet die Angabe #PCDATA.

```
<!ELEMENT title (#PCDATA)>
```

Damit wäre in einem XML-Dokument

```
<title></title>
<title>Skriptum XML-Vorlesung &#xA9; L. Wegner</title>
```

legal, nicht jedoch

```
<title><emphasis>Skriptum XML-Vorlesung</emphasis></title>
```

Soll ein Element in einem anderen Element auftreten, gibt man das mit runden Klammern an. Muss in unserem Beispiel ein Buchtitel in einem Buch erscheinen, lauten die Deklarationen:

```
<!ELEMENT book (title)>
<!ELEMENT title (#PCDATA)>
```

3.1.2 Mehrfache Elemente

Sollen mehrere Elemente in einer vorgegebenen Reihenfolge auftreten, gibt man diese durch Komma getrennt an:

```
<!ELEMENT book (title, authors)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT authors (#PCDATA)>
```

Damit kann man in einem XML-Dokument schreiben:

```
<book>
<title>Skriptum XML-Vorlesung</title>
<authors>Wegner</authors>
</book>
```

Ein Vertauschen oder Weglassen eines Elements wäre aber nicht möglich. Dies erreicht man mit der Alternative, allerdings muss genau *eine* der Alternativen erscheinen!

```
<!ELEMENT book (title | authors)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT authors (#PCDATA)>
```

3.1.3 Gruppierung und Wiederholung

Wie man dies aus den kontextfreien Grammatiken in EBNF-Notation und von der Shell-Programmierung kennt, kann man Gruppierungen und (optionale) Wiederholungen angeben. Soll ein Buch entweder einen Titel gefolgt von Autoren, oder nur eine Beschreibung enthalten, schreibt man:

```
<!ELEMENT book ((title, authors) | description)>
```

```
<!ELEMENT title (#PCDATA)>
<!ELEMENT authors (#PCDATA)>
<!ELEMENT description (#PCDATA)>
```

Das optionale Auftreten gibt man mit `?`, das mindestens einmalige mit `+`, und das null- oder mehrmalige Auftreten mit `*` an. Die Metazeichen stehen unmittelbar nach den Elementnamen oder den Gruppen, auf die sie sich beziehen sollen. Das Beispiel in [21] lautet:

```
<!ELEMENT reviews (rating, synopsis?, comments+)*>
<!ELEMENT rating ((tutorial | reference)*, overall)>
<!ELEMENT synopsis (#PCDATA)>
<!ELEMENT comments (#PCDATA)>
<!ELEMENT tutorial (#PCDATA)>
<!ELEMENT reference (#PCDATA)>
<!ELEMENT overall (#PCDATA)>
```

3.1.4 Leere Elemente

Auch das Auftreten leerer Elemente in einem XML-Dokument muss angegeben werden.

```
<!ELEMENT elementname EMPTY>
```

Damit kann durch `<!ELEMENT statuscode EMPTY>` im XML-Dokument entweder

```
<statuscode></statuscode>
```

oder

```
<statuscode/>
```

stehen.

3.2 Entities und Notationen

Ähnlich wie bei Makros lassen sich Zeichenfolgen mit einem Bezeichner versehen. Einige Standardvorgaben (z. B. `&`) haben wir schon gesehen, diese müssen nicht gesondert angegeben werden. Andere gibt man wie folgt an:

```
<!ENTITY copyright "&#xA9;">
```

Im Dokument schreibt man dann

```
<title>XML-Skript &copyright; L. Wegner 2000</title>
```

und erzeugt damit die Ausgabe `XML-Skript © L. Wegner 2000`.

Wie bei Makros üblich, dürfen die Angaben nicht zirkulär sein. Auch dürfen in den Substitutionszeichen keine weiteren Metazeichen auftauchen, da die Ersetzung erst im XML-Dokument erfolgt, nicht bereits in der DTD. Dies ist bei den folgenden parametrischen Entities anders.

3.2.1 Parameter-Entities

Diese werden nur in DTDs verwandt und können als deklarierte Elemente nicht in XML-Dokumenten auftreten. Sie stellen abkürzende Schreibweisen für DTDs dar. Vor dem Namen erscheint ein Prozentzeichen.

```
<!ENTITY % name "replacement characters">
```

Das Beispiel in [21] lautet:

```
<!ENTITY % pcdata "(#PCDATA)">
<!ELEMENT authortitle %pcdata;>
```

Wie zuvor dürfen die Deklarationen nicht zirkulär sein und ein Parameter-Entity muss zuerst deklariert sein, bevor es verwendet werden kann.

3.2.2 Externe Parameter-Entities

Externe Parameter-Entities dienen dazu, große DTDs auf verschiedene Dateien aufteilen zu können. Mit der Parameter-Entity-Angabe kann man **Teile** einer DTD in eine andere importieren.

Die Deklaration ähnelt der Deklaration einer Parameter-Entity, doch statt einen Ersatztextes gibt es hier eine `PUBLIC`- oder `SYSTEM`-Angabe. Beispiel:

```
<!ENTITY % format-elemente SYSTEM "formats.mod">
```

Durch den Aufruf von `%format-elemente;` in einer DTD wird der Inhalt der Datei `formats.mod` an der Stelle des Aufrufs eingebaut.

3.2.3 Externe Entities

Hier sollen Daten, die von einer anzugebenden URI stammen, dynamisch in das XML-Dokument geladen werden.

```
<!ENTITY fremd SYSTEM "fremd.xml">
```

Befinden sich in `fremd.xml` (ggf. an einem anderen Ort) XML-Daten, zum Beispiel

```
<einbau>Wer&apos;s glaubt wird seelig</einbau>
```

wird dieser Text zur Laufzeit in das XML-Dokument aufgenommen, das die Referenz `&fremd;` enthält.[1]

3.2.4 Notationen

Der Zweck einer „Notations-Vereinbarung“ (notation declaration) ist es, ein Format für eine externe Datei anzugeben, die kein XML-Dokument ist. Dies könnte ein Bild oder eine Audiodatei sein. Mit der Formatangabe kann man jetzt auf diese Datei verweisen.

Die zwei allgemeinen Formen einer Notationsvereinbarung sind:

```
<!NOTATION nname PUBLIC "std">
<!NOTATION nname SYSTEM "url">
```

wobei *nname* der selbstvergebene Name für die Notation ist, *std* ist der veröffentlichte Name einer „public notation“, und *url* ist ein Hinweis auf ein Programm, das die Datei, deren Notation hier angegeben wird, darstellen kann.

In vier Schritten verbindet man ein Attribut mit einer Notation:

1. Die Notation vereinbaren. Beispiel:
   ```
   <!NOTATION jpeg PUBLIC "JPG 1.0">
   ```

1. In `fremd.xml` darf `&fremd;` natürlich nicht verwendet werden.

2. Das Entity vereinbaren (NDATA steht für *notation data*). Z. B.:
   ```
   <!ENTITY bogie-pic SYSTEM
             "http://stars.com/bogart.jpg" NDATA jpeg>
   ```
3. Das Attribut vereinbaren als vom Typ `ENTITY`. Zum Beispiel:
   ```
   <!ATTLIST star-bio pin-shot ENTITY #REQUIRED>
   ```
4. Das Attribut verwenden:
   ```
   <star-bio pin-shot="bogie-pic">...</star-bio>
   ```

Anmerkung: Beispiel aus
`http://infohost.nmt.edu/tcc/help/pubs/dtd/notation-sect.html`

Andere Beispiele für den in

```
<!NOTATION Name Identifier>
```

stehenden *`Identifier`* folgen. Es ist jedoch nicht genau festgelegt, wie *`Identifier`* anzugeben ist.

Beispiel	**Bedeutung**
`SYSTEM "application/x-troff"`	MIME-Type
`SYSTEM "ISO 8601:1988"`	Internationale Standards
`SYSTEM "http://www.w3.org/TR/-NOTE-datetime"`	Die URL über ein technisches Dokument im Internet
`SYSTEM "/usr/local/bin/xv"`	Ein Softwareprogramm auf dem lokalen Rechner
`PUBLIC "-//FPI"`	Eine URI

XML macht keine genauen Angaben über den Umgang mit nicht analysierten Daten. Die Interpretation bleibt also der Anwendung überlassen, die das Dokument bearbeitet (XML-Prozessor). Es ist daher ratsam, Notationen zu vermeiden und stattdessen XML-Instruktionen zu verwenden.

Diese werden bekanntlich von anderen Anwendungen ignoriert, falls sie nicht „verstanden“ werden.

3.2.5 Unparsed Entities

Wie oben gezeigt lassen sich Daten, die kein XML darstellen, in XML-Dokumente einbauen. Üblich sind z. B. Bilder. Beispiel:

```
<!ENTITY meinkopf SYSTEM "test.jpg" NDATA jpg>
```

Speziell wird damit durch das Schlüsselword `NDATA` (notation data) angegeben, um welche Art von Rohdaten es sich handelt, hier ein jpg-Pixelbild. Dabei soll `jpg` als Notation dem XML-Prozessor bekannt sein. Beispiel:

```
<?xml version='1.0'?>
<!DOCTYPE rootElement [
<!ELEMENT rootElement (img)>
<!ELEMENT img EMPTY>
<!ATTLIST img src ENTITY #REQUIRED>
<!NOTATION jpg SYSTEM "image/jpeg">
<!ENTITY meinkopf SYSTEM "test.jpg" NDATA jpg>
]>
<rootElement>
  <img src='meinkopf'/>
</rootElement>
```

Man achte darauf, dass hier `meinkopf` statt `&meinkopf;` steht, da das Attribut `src` als `ENTITY` definiert wurde. Die Aufgabe, ungeparste Entities aufzulösen, übernimmt der XML-Prozessor, der das Dokument verarbeitet. In XSLT z. B. steht dafür die Funktion `unparsed-entity-uri` zur Verfügung. Damit lassen sich Entities anhand ihrer Spezifikation in der DTD auflösen. Somit wird das `<img>`-Element im Stylesheet wie folgt verarbeitet:

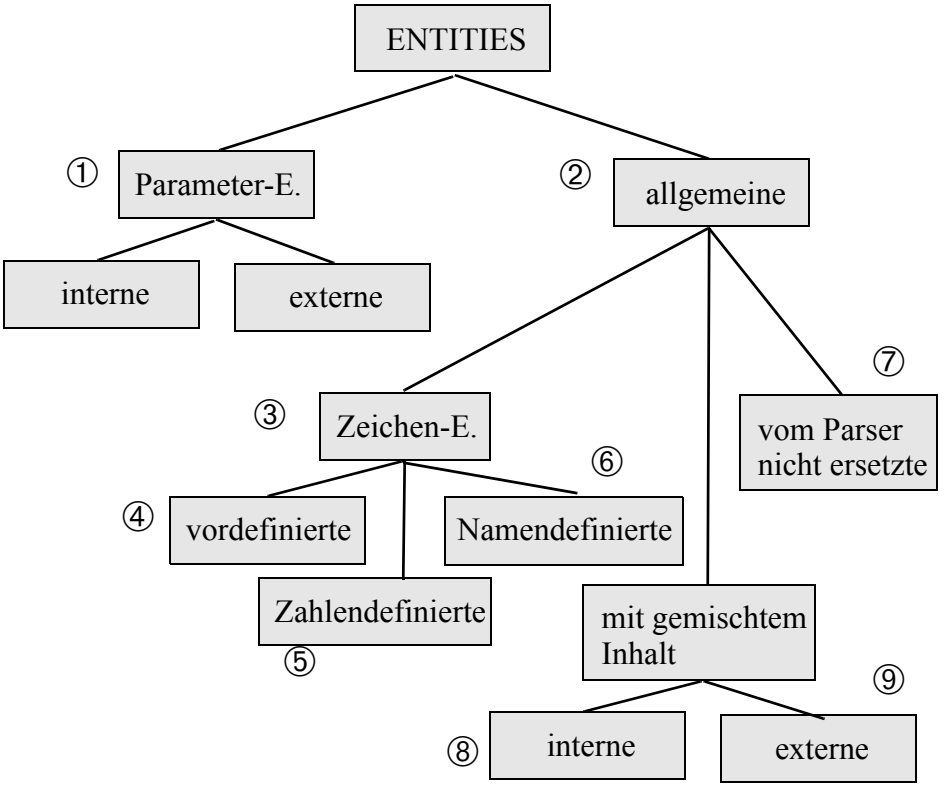
```
<xslt:template match='img'>
  <xslt:element name='img'>
    <xslt:attribute name='src'>
      <xslt:value-of select='unparsed-entity-uri(@src)'/>
    </xslt:attribute>
  </xslt:element>
</xslt:template>
```

Wobei `@src` in XSLT den Wert des Attributes `src` ausgibt. Xalan produziert die folgende Ausgabe in HTML:

```
<img src="file:///home/staff/morad/ ... /XML/test.jpg"/>
```

3.2.6 Zusammenfassung Entities

In [7, p. 52] werden Entities mit der folgenden Abbildung klassifiziert.



In [28, p. 57] werden Entities wie folgt zusammengefasst (die Nummern in Kreisen stellen den Bezug zu der Klassifikation oben her):

- **Standard entities** ④, etwa `<`
  werden intern deklariert, keine Vereinbarung nötig.
- **Character entities** ⑤, etwa `&#x202A;` als Zeichenersatz für Zeichen außerhalb des 7-bit ASCII-Zeichensatzes oder für ein Zeichen, das nicht auf der Tastatur erscheint. **Zusatz:** Werden dafür Namen vergeben, etwa wie für den ISO-8879 Zeichensatz, dann trifft dies den Fall ⑥
- **Internal Entities** ⑧ für längere Texte, die selber wieder Markup enthalten können und die den Ersatztext in der Entity-Deklaration selbst stehen haben. Geeignet für häufige Ersetzungen in *einem* Dokument.
- **External Entities** ⑨ für längere Texte inkl. Markup, wenn die Ersetzungen konsistent über *mehrere* Dokumente erfolgen sollen; daher werden sie in einer externen Datei gespeichert (sog. XML-Fragmente); das nach dem Import zusammengesetzte XML-Dokument muss insgesamt den Wohlgeformtheitsregeln genügen.
- **External unparsed entities** ⑦ für Inhalte nichttextueller Art, z.B. Graphiken, die der Parser nicht verarbeiten kann.
- **Parameter entities** ① für Ersetzungen innerhalb einer DTD, etwa weil mehrere Element-Deklarationen gemeinsame Teile haben; gibt es als interne oder externe (aus externer Datei) Variante. Verwendet das **%-Zeichen** bei Deklaration und Aufruf.

3.3 Attributlisten

Nach der Deklaration eines Elements können dessen Attribute als *Attributliste* deklariert werden. Dabei kann man für ein Element mehrere Attributlisten spezifizieren. Die Vereinbarung hat die folgende Syntax:

```
<!ATTLIST Name
          Attname1 Attype1 Attbeschr1
          Attname2 Attype2 Attbeschr2
...
>
```

Die Deklaration beginnt mit der Zeichenkette `<!ATTLIST`. Als zweites kommt der Name des Elements, für den die Attribute definiert werden. Danach folgt eine beliebige Anzahl von *Attributdeklarationen*.

Eine Attributdeklaration besteht aus einem Attributnamen, dessen Datentyp oder aufgezählten Werten und der Beschreibung des Attributverhaltens. Im folgenden Beispiel werden z. B. die drei Attribute `id` vom Typ `ID`, `name` vom Typ `CDATA` und `gruppe` als Aufzählungstyp für das Element `benutzer` definiert:

```
<!ATTLIST benutzer
          id       ID                          #REQUIRED
          name     CDATA                       #IMPLIED
          gruppe   (student | mitarb | gast) "gast"
>
```

Es folgt eine Liste der möglichen Attribut-Typen:

- **`CDATA` (Zeichendaten)**
  Attribute mit diesem Typ können beliebige Zeichendaten als Werte bekommen.[1] Beispiel dafür:
  ```
  titel="In 80 Tagen um die Welt"
  erklaerung="4! = 4*3! = 4*3*2 = 24"
  signatur="023 25 Wir OU 5114"
  ```
  Anführungszeichen müssen durch Zeichen-Entity ersetzt werden oder man verwendet einzelne Anführungszeichen (`'...'`) in doppelten (`"..."`) bzw. umgekehrt. Beispiel:
  ```
  veranstaltung='Diskussion über die "Agenda 2010"'
  ```
  Tabulatoren und Newline-Zeichen werden in Leerzeichen umgewandelt. Eine Kette von Leerzeichen bleibt dagegen erhalten.
- **`NMTOKEN` (Namens-Token)**
  Ein Wert bei `NMTOKEN` ist eine **Zeichenkette** (eine einzelne), die den Regeln für Element- und Attributnamen entspricht, allerdings keine Einschränkungen bzgl. des ersten Zeichens macht. Alle Leerzeichen werden vom XML-Prozessor entfernt. Beispiele:

1. In allen Attribut-Typen dürfen Elemente oder Prozessorinstruktionen nicht verwendet werden.

```
datei='install.sh'
rechner='141.51.8.4'
```

- **NMTOKENS (Namens-Token-Listen)**
  Eine Liste von NMTOKEN-Werte, die durch Leerzeichen getrennt werden. Zusätzliche Leerzeichen werden vom XML-Prozessor entfernt. Beispiel:
  ```
  artikel='die der das'
  farben='rot gelb blau'
  ```
- **ID (Eindeutige Bezeichner)**
  Ein Wert dieses Typs verleiht seinem Element einen Label, der innerhalb des ganzen Dokuments eindeutig ist. Es darf daher kein anderes Element mit demselben Attributwert im Dokument geben. Der Inhalt folgt den Formvorschriften für Element- und Attributnamen.
  Üblicherweise wird ID mit dem Schlüsselwort #REQUIRED verwendet. Zwingend ist dies allerdings nicht. Beispiel:
  ```
  urn="urn:nbn:de:hebis:34-593"
  buchnr="b17010808"
  ```
- **IDREF (Verweis)**
  Ähnlich wie ID, dient allerdings zum Verweis auf andere Elemente im Dokument durch die Angabe ihrer ID. Existiert im Dokument kein Element mit dieser ID, so muss der XML-Prozessor einen Fehler produzieren. Beispiel:
  ```
  author-id="a4711"
  ```
- **IDREFS (Verweisliste)**
  Die Syntax ist ähnlich wie NMTOKENS. Die aufgelisteten Werte müssen aber auf im Dokument existierende Element mit diesen ID-Werten verweisen:
  ```
  authors-id="a4711 a5711 a6711"
  ```
- **ENTITY (Entity-Name)**
  Dieser Typ akzeptiert den Namen einer nicht-geparsten Entity als Wert. Beispiel:
  ```
  <!ENTITY meinkopf SYSTEM "test.jpg" NDATA jpg>
  <!ATTLIST img src ENTITY #REQUIRED>
  ```
  Im XML-Dokument wird dieser Attribut wie folgt verwendet:
  ```
  <img src="meinkopf">
  ```

- **`ENTITIES` (Entity-Namenliste)**
  Der Wert dieses Attributs ist eine Liste von Entity-Namen.
- **Aufzählungsliste**
  Hier spezifiziert man eine Liste von Namens-Token, aus denen das Attribut einen Wert annehmen kann. Die Werte der Liste werden durch | voneinander getrennt, z. B.:

```
<!ATTLIST veranstaltung
     tag (Mo | Di | Mi | Do | Fr | Sa | So) "Sa">
```

  Ein Attribut kann nur einen dieser Werte bekommen. Falls ein *Standardwert* spezifiziert ist, so muss dieser auch in der Liste auftreten.
- **`NOTATION` (Notationsliste)**
  Der Wert eines `NOTATION`-Attributs besteht aus einem Namenstoken aus einer bei der Deklaration des Typs angegebenen Liste von möglichen Werten. Ein Beispiel aus [7] lautet:

```
<?xml version="1.0"?>
<!DOCTYPE doc [
  <!ELEMENT doc (listing*)>
  <!ELEMENT listing (#PCDATA)>
  <!NOTATION schema-lisp SYSTEM "IEEE 1178-1990">
  <!NOTATION ansi-c SYSTEM "ISO/IEC 8899:1999">
  <!ATTLIST listing format NOTATION
                   (schema-lisp | ansi-c) #REQUIRED>
]>
<doc>
  <listing format='schema-lisp'>
       (defun fact (lambda (n)
              (if (= n 1) 1 (fact (- n 1)))))
  </listing>
  <listing format='ansi-c'>
       int fact(int n) {
              if (n == 1) return 1;
              return n * fact (n - 1);
       }
  </listing>
</doc>
```

  Dabei beschreibt hier die Notation, wie Textdaten interpretiert werden

sollen. Die externen Identifier stammen von den internationalen Standardinstitutionen IEEE und ISO.

Nach der Spezifikation des Typs eines Attributes kann sein *Verhalten* spezifiziert werden. Dazu existieren die folgenden Möglichkeiten:

- **Standardwert zuweisen**
  Wenn dieses Attribut im XML-Dokument nicht angegeben wird, wird der Standardwert vom XML-Prozessor angenommen. Dies macht Sinn, wenn ein Wert sehr häufig für ein Element vorkommt. Zum Beispiel:
  ```
  <!ATTLIST article language (en | de | fr) "en">
  ```
- **Das Attribut ist optional (`#IMPLIED`)**
  Mit `IMPLIED` wird ausgedruckt, dass das Attribut keinen Standardwert besitzt und dass das Attribut optional ist. Zum Beispiel:
  ```
  <!ATTLIST article pages CDATA #IMPLIED>
  ```
- **Attribut muss angegeben werden (`#REQUIRED`)**
  Das Weglassen des Attributes ist nicht erlaubt. Zum Beispiel:
  ```
  <!ATTLIST article title CDATA #REQUIRED>
  ```
- **Der Wert ist bereits gesetzt, und der Benutzer kann ihn nicht ändern (`#FIXED`)**
  Für den Fall, dass ein Attribut stets denselben Wert hat, kann man damit einiges an Schreibarbeit sparen. Zum Beispiel:
  ```
  <!ATTLIST article license CDATA #FIXED "free">
  ```
  Das Attribut `license` kann weggelassen werden, gibt man es an, muss es den Wert `free` haben.

3.4 Modularisierung

DTDs können ziemlich unübersichtlich werden. Daher wurden Möglichkeiten geschaffen, eine DTD auf mehrere Module aufzuteilen. Damit kann man

- eine DTD leichter warten,
- die Module einer DTD auf mehrere Entwickler verteilen,
- die DTD durch bedingte Abschnitte konfigurieren und

- Fehler in einer DTD besser finden und korrigieren.

XML stellt zwei Möglichkeiten zur Modularisierung einer DTD zur Verfügung:

- Aufspalten der DTD auf mehrere *Module*. Die Module werden mit externen Parameter-Entities importiert.
- Bedingte Abschnitte.

3.4.1 Importieren von Modulen

Man kann eine DTD auf mehrere Module aufteilen. Diese können dann durch Parameter-Entities in einer anderen DTD *importiert* werden. Beispiel:

```
<!ELEMENT script (titel | autor | inhalt)>
<!ENTITY % script.formats SYSTEM "formats.mod">
<!ENTITY % script.struct SYSTEM "struct.mod">
<!-- Lokale Deklarationen -->
...
<!-- hier importieren -->
%script.formats;
%script.struct;
...
```

Die Endung `.mod` ist keine Vorschrift, jedoch die übliche Bezeichnung, um die Datei als eine Sammlung von Deklarationen und nicht als eigenständige DTD zu markieren.

Da es in DTDs keine Möglichkeit für lokale Definitionen mit Sichtbarkeitsbereichen gibt, sondern nur einen globalen Kontext, in dem alle Deklarationen in der gesamten DTD sichtbar sind, wird mit einem externen Parameter-Entity der *gesamte* Text aus der Datei importiert.

Bei Importieren von Modulen müssen die folgenden Regeln für das mehrfache Vorkommen einer Deklaration beachtet werden:

- Existieren mehrere Attributlistendeklarationen für ein Element, so werden diese miteinander verkettet.

- Entitäts- und Attributdeklarationen:
  Existieren für eine Entität oder ein Attribut mehrere Deklarationen, so wird jeweils die erste verwendet; alle folgenden werden ignoriert.
- Element- und Notationsdeklarationen:
  Elemente und Notationen dürfen nicht mehr als einmal deklariert werden; mehrfaches Vorkommen ist ein Gültigkeitsfehler.

3.4.2 Bedingte Abschnitte

Man kann mit einem sog. *bedingten Abschnitt* bestimmte Teile eines Moduls oder einer *öffentlichen* DTD zum Einbinden in anderen DTDs bzw. zum Ausschluss aus der DTD markieren. Bedingte Abschnitte werden wie folgt deklariert:

```
<![INCLUDE[
<!-- Die folgenden Deklarationen werden einbezogen-->
<!ELEMENT kreis (x, y, r)>
...
]]>
<![IGNORE[
<!-- Die folgenden Deklarationen werden nicht einbezogen-->
<!ELEMENT rechteck (x1, y1, x2, y2)>
...
]]>
```

Mit Hilfe von Parameter-Entities kann man in einer DTD bestimmen, welche Teile aus anderen DTDs übernommen werden. Dazu werden in der zu importierenden DTD die `INCLUDE` und `IGNORE` Abschnitte durch Entities deklariert. Zum Beispiel:

```
<!ENTITY % formen "INCLUDE">
<!ENTITY % rechtecke "IGNORE">
<![%formen;[
<!-- Die folgenden Deklarationen werden einbezogen-->
<!ELEMENT kreis (x, y, r)>
...
]]>
<![%rechtecke;[
<!-- Die folgenden Deklarationen werden nicht einbezogen-->
<!ELEMENT rechteck (x1, y1, x2, y2)>
...
]]>
```

In einer DTD, die dieses Modul verwendet, können dann die beiden Entities überschrieben werden, womit man bestimmt, ob die Deklarationen für Rechtecke oder Formen übernommen werden. Im folgenden Beispiel werden die Standardeinstellungen aus der importierten Datei `geometrie.mod` vertauscht:

```
<!ENTITY % formen "IGNORE">
<!ENTITY % rechtecke "INCLUDE">
<!ENTITY % geometrie SYSTEM "geometrie.mod">
%geometrie;
...
```

Dadurch kann man DTDs in logisch zusammenhängende Blöcke aufteilen und diese durch Parameter-Entities mit `INCLUDE` oder `IGNORE` vorbelegen. Bevor die DTD importiert wird, kann man diese Entities überschreiben und so genau bestimmen, welche Blöcke aus dieser DTD verwendet werden. Dies ist ein mächtiges Werkzeug, das besonders beim Erstellen von öffentlichen DTDs hilfreich ist.

3.4.3 Verwenden der internen Teilmenge

In der Einführung haben wir zwei Formen der `DOCTYPE`-Deklaration in einem XML-Dokument kennengelernt:

```
<?xml version='1.0'?>
<!DOCTYPE rootElement SYSTEM "dateiname.dtd">
```

und

```
<?xml version='1.0'?>
<!DOCTYPE rootElement[
interne Deklarationen
]>
```

Eine weitere Möglichkeit besteht darin, eine externe DTD einzubinden und zusätzlich eine *interne Teilmenge* zu spezifizieren:

```
<?xml version='1.0'?>
<!DOCTYPE rootElement SYSTEM "dateiname.dtd" [
interne Deklarationen
]>
```

In diesem Teil kann man z. B. die Parameter-Entities setzen, um zu bestimmen, welche Teile aus der importierten DTD verwendet werden. Zum Beispiel:

```
<?xml version='1.0'?>
<!DOCTYPE rootElement SYSTEM "script.dtd" [
<!ENTITY % BiBTeXStyle.Modul "INCLUDE">
<!ENTITY % BiBWordStyle.Modul "IGNORE">
]>
```

3.5 Tipps zum Erstellen von DTDs

Wie Programmcode können unkommentierte und schlecht formatierte DTDs nur bedingt nachvollzogen werden. Daher die folgenden Tipps:

- Deklarieren Sie logisch zusammenhängende Elemente in voneinander getrennten Abschnitten.
- Verwenden Sie Leerräume, um Ihre DTD zu formatieren und damit lesbarer zu machen. In DTDs werden Leerzeichen, Tabulatoren und Newline-Zeichen ignoriert. DTDs wie

  ```
  <!ATTLIST buch titel CDATA #REQUIRED autor
    NMTOKEN #REQUIRED isbn ID #REQUIRED>
  ```

  sind schwer lesbar. Die folgende äquivalente Deklaration sieht jedenfalls viel besser aus:

  ```
  <!ATTLIST buch
     titel    CDATA     #REQUIRED
     autor    NMTOKEN   #REQUIRED
     isbn     ID        #REQUIRED
  >
  ```

- Kommentieren Sie ihre DTDs, damit andere Benutzer die Deklarationen verstehen. Auch der Autor selbst könnte nach einiger Zeit nur mit Mühe sein eigenes Werk ohne Kommentare verstehen.

  ```
  <!-- buch hat ein Titel, Autor und ISBN-Nummer. -->
  <!-- Die ISBN ... -->
  <!ATTLIST buch ...
  ```

- Versuchen Sie wiederkehrende Teile durch Parameter-Entities zu definieren:

```
<!ENTITY % standelements "title, author, year">
<!ELEMENT book (%standelements;, isbn, ...)>
<!ELEMENT article (%standelements;, pages, ...)>
<!ELEMENT thesis (%standelements;, school, ...)>
```

3.6 Ein Datenbankbeispiel

In diesem Beispiel betrachten wir eine geschachtelte Tabelle, die wir aus zwei flachen Tabellen `Abteilung` und `Mitarbeiter` mit einer *1:n*-Beziehung (jeder Mitarbeiter ist in genau einer Abteilung) herstellen.

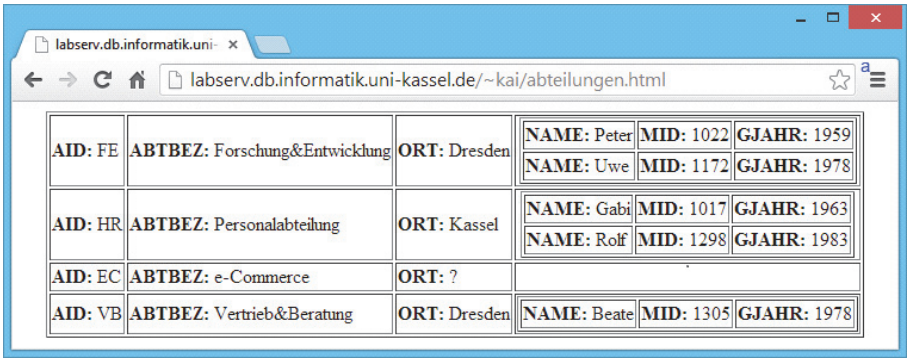
AID	ABTBEZ	ORT
FE	Forschung&Entwicklung	Dresden
HR	Personalabteilung	Kassel
EC	e-Commerce	?
VB	Vertrieb&Beratung	Dresden

Tab. 3–1 *Abteilungen*

NAME	MID	GJAHR	AID
Peter	1022	1959	FE
Gabi	1017	1963	HR
Beate	1305	1978	VB
Rolf	1298	1983	HR
Uwe	1172	1978	FE

Tab. 3–2 *Mitarbeiter*

Mithilfe eines Stylesheet (`nf2tohtml.xsl`) erzeugen wir z. B. die folgende geschachtelte Darstellung:



AID: FE	**ABTBEZ:** Forschung&Entwicklung	**ORT:** Dresden	**NAME:** Peter **MID:** 1022 **GJAHR:** 1959 **NAME:** Uwe **MID:** 1172 **GJAHR:** 1978
AID: HR	**ABTBEZ:** Personalabteilung	**ORT:** Kassel	**NAME:** Gabi **MID:** 1017 **GJAHR:** 1963 **NAME:** Rolf **MID:** 1298 **GJAHR:** 1983
AID: EC	**ABTBEZ:** e-Commerce	**ORT:** ?	
AID: VB	**ABTBEZ:** Vertrieb&Beratung	**ORT:** Dresden	**NAME:** Beate **MID:** 1305 **GJAHR:** 1978

Abb. 3–1 *Geschachtelte Darstellung von Abteilungen und Mitarbeitern*

Eine DTD für die geschachtelte Form wäre offensichtlich (`abteilungen.dtd`):

```
<!-- DTD fuer Abteilung und Mitarbeiter -->
<!ENTITY % nf2types.mod SYSTEM "nf2types.mod">
%nf2types.mod;

<!ELEMENT ABTEILUNGEN (ABTEILUNG*)>
<!ATTLIST ABTEILUNGEN
  %settype;
  %idtype;>

<!ELEMENT ABTEILUNG (AID, ABTBEZ, ORT, MITARBEITER)>
<!ATTLIST ABTEILUNG
  %tupletype;
  %idtype;>

<!ELEMENT AID %atomictype;>
<!ATTLIST AID %idtype;>
<!ELEMENT ABTBEZ %atomictype;>
<!ATTLIST ABTBEZ %idtype;>
<!ELEMENT ORT %atomictype;>
<!ATTLIST ORT %idtype;>

<!ELEMENT MITARBEITER (MITARB*)>
<!ATTLIST MITARBEITER
```

```
 %settype;
 %idtype;>

<!ELEMENT MITARB (NAME, MID, GJAHR)>
<!ATTLIST MITARB
 %tupletype;
 %idtype;>

<!ELEMENT NAME %atomictype;>
<!ATTLIST NAME %idtype;>
<!ELEMENT MID %atomictype;>
<!ATTLIST MID %idtype;>
<!ELEMENT GJAHR %atomictype;>
<!ATTLIST GJAHR %idtype;>
```

Wobei das Modul `nf2types.mod` in diese DTD eingebunden wurde:

```
<!ENTITY % settype "nf2type CDATA #FIXED 'sett'">
<!ENTITY % listtype "nf2type CDATA #FIXED 'listt'">
<!ENTITY % tupletype "nf2type CDATA #FIXED 'tuplet'">
<!ENTITY % atomictype "(#PCDATA)">
<!ENTITY % idtype "id ID #REQUIRED">
```

Ein Beispieldokument nach dieser DTD wäre (`abteilungen.xml`):

```
<?xml version='1.0' standalone="no" ?>
<?xml-stylesheet type="text/xsl" href="nf2tohtml.xsl"?>
<!DOCTYPE ABTEILUNGEN SYSTEM "abteilungen.dtd">
<ABTEILUNGEN id="RID-4000" >
 <ABTEILUNG id="RID-4001">
  <AID id="RID-4003">FE</AID>
  <ABTBEZ id="RID-4004">
   Forschung&amp;amp;Entwicklung
  </ABTBEZ>
  <ORT id="RID-4005">Dresden</ORT>
  <MITARBEITER id="RID-4006">
   <MITARB id="RID-4007">
    <NAME id="RID-4008">Peter</NAME>
    <MID id="RID-4009">1022</MID>
    <GJAHR id="RID-4010">1959</GJAHR>
   </MITARB>
   <MITARB id="RID-4011">
    <NAME id="RID-4012">Uwe</NAME>
    <MID id="RID-4013">1172</MID>
    <GJAHR id="RID-4014">1978</GJAHR>
   </MITARB>
```

```
    </MITARBEITER>
  </ABTEILUNG>
  <ABTEILUNG id="RID-4020">
    <AID id="RID-4021">HR</AID>
    <ABTBEZ id="RID-4022">
      Personalabteilung
    </ABTBEZ>
    <ORT id="RID-4023">Kassel</ORT>
    <MITARBEITER id="RID-4024">
      <MITARB id="RID-4025">
        <NAME id="RID-4026">Gabi</NAME>
        <MID id="RID-4027">1017</MID>
        <GJAHR id="RID-4028">1963</GJAHR>
      </MITARB>
        <MITARB id="RID-4029">
        <NAME id="RID-4030">Rolf</NAME>
        <MID id="RID-4031">1298</MID>
        <GJAHR id="RID-4032">1983</GJAHR>
      </MITARB>
    </MITARBEITER>
  </ABTEILUNG>
  <ABTEILUNG id="RID-4040">
    <AID id="RID-4041">EC</AID>
    <ABTBEZ id="RID-4042">
      e-Commerce
    </ABTBEZ>
    <ORT id="RID-4043">?</ORT>
    <MITARBEITER id="RID-4044">
    </MITARBEITER>
  </ABTEILUNG>
  <ABTEILUNG id="RID-4060">
    <AID id="RID-4061">VB</AID>
    <ABTBEZ id="RID-4062">
      Vertrieb&amp;amp;Beratung
    </ABTBEZ>
    <ORT id="RID-4063">Dresden</ORT>
    <MITARBEITER id="RID-4064">
      <MITARB id="RID-4065">
        <NAME id="RID-4066">Beate</NAME>
        <MID id="RID-4067">1305</MID>
        <GJAHR id="RID-4068">1978</GJAHR>
      </MITARB>
    </MITARBEITER>
  </ABTEILUNG>
</ABTEILUNGEN>
```

4 XML Schema

Im vorigen Kapitel haben wir die Möglichkeit kennengelernt, ein XML-Dokument mit Hilfe einer DTD zu beschreiben. Die Nachteile sind:

- DTDs unterstützen keine Namensräume.
- DTDs haben eine sehr beschränkte Anzahl an Datentypen. Z. B. gibt es keine Datentypen für Zahlen, Datum etc.
- DTDs sind selbst keine XML-Dokumente.

XML Schema (kurz **XSchema**) ist ein Standard des W3C, der ursprünglich DTDs ersetzen sollte. XSchema selbst ist eine XML-basierte Sprache, wodurch die Verarbeitung von XSchema-Dokumenten mit den gleichen Werkzeugen wie für XML-Dokumente erfolgen kann (Parser, DOM-Schnittstelle etc.).

Mit XSchema wird also ein in sich abgeschlossenes System geschaffen; es ist sogar möglich, ein XSchema für beliebige XSchema-Dokumente anzugeben, ein sog. *Metaschema*, das alle Schemadokumente inklusive sich selbst beschreibt.

Die Standardisierung erfolgt in zwei Teilen: *XML Schema Part 1: Structures* [22] und *XML Schema Part 2: Datatypes* [23]. Diese beiden Quellen sind schwer zu verstehen. Der sog. *Primer* [8] enthält eine übersichtliche Einführung.

XSchema ist sehr umfangreich. Wir behandeln hier nur die Grundlagen. Für weitere Details siehe [24].

4.1 Grundlagen

XSchema-Dokumente haben üblicherweise die Endung `*.xsd`. Andere Endungen können ebenfalls verwendet werden. Üblich ist natürlich auch `*.xml`! Als Namensraum wird `http://www.w3.org/2001/XMLSchema` benutzt. Üblicherweise werden die Abkürzungen `xsd` oder `xs` verwendet.

Ein XSchema hat den folgenden Aufbau:

```
<?xml version='1.0'?>
<xsd:schema
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:annotation>
    <xsd:documentation xml:lang="de-DE">
      Dokumentation über das Schema
    </xsd:documentation>
  </xsd:annotation>
  <xsd:element name="x" type="xType"/>
  ...
</xsd:schema>
```

Der Element `xsd:annotation` ist optional und dient zur Angabe von allgemeinen Informationen über das Schema.

Wenn man Namensräume verwendet, kann man durch die Angabe des Attributes `xsi:schemaLocation` für das Wurzelelement auf das verwendete Schema verweisen. Verwendet man keine Namensräume so wird das Attribut `xsi:noNamespaceSchemaLocation` verwendet. Beispiel:

```
<document
  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xsi:noNamespaceSchemaLocation='mydocument.xsd'>
    ...
</document>
```

Ein Dokument, das ein Attribut `schemaLocation` verwendet, folgt hier.

```
<p:Person xmlns:p="http://contoso.com/People"
   xmlns:v="http://contoso.com/Vehicles"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://contoso.com/People
     http://contoso.com/schemas/people.xsd
     http://contoso.com/Vehicles
     http://contoso.com/schemas/vehicles.xsd">
```

```
   <name>John</name>
   <age>28</age>
   <height>59</height>
   <v:Vehicle>
     <color>Red</color>
     <wheels>4</wheels>
     <seats>2</seats>
   </v:Vehicle>
</p:Person>
```

Ein XML-Dokument, das nach einem XSchema aufgebaut ist, wird als *Instanzdokument* dieses Schemas bezeichnet. Als erstes Beispiel betrachten wir ein Schema für das Dokument `teilnehmer0.xml` aus Kapitel 1 (`teilnehmer0.xsd`):

```
<?xml version='1.0'?>
<xsd:schema
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="TeilnehmerS"
               type="TeilnehmerSType"/>
  <xsd:complexType name="TeilnehmerSType">
    <xsd:sequence>
      <xsd:element name="Teilnehmer"
                   type="TeilnehmerType"
                   minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="vorlesung" type="xsd:string"/>
  </xsd:complexType>
  <xsd:complexType name="TeilnehmerType">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="vorname" type="xsd:string"/>
      <xsd:element name="semester"
                   type="xsd:positiveInteger"/>
      <xsd:element name="fachb" type="xsd:string"/>
    </xsd:sequence>
    <xsd:attribute name="matrNr"
                    type="xsd:positiveInteger"/>
  </xsd:complexType>
</xsd:schema>
```

4.1.1 Deklaration von Elementen

Elemente werden mit `<xsd:element>` deklariert. Man kann die *Typdefinition* eines Elements in der Deklaration angeben (*anonyme Typdefinition*) oder einen Typnamen angeben. Der Typname muss entweder *vordefiniert* sein (z. B. `xsd:string`) oder im Schema weiter ausgeführt sein:

```
<xsd:element name="elementName" type="typName"/>
...
<xsd:complexType name="typName">
  Definition von typName
</xsd:complexType>
```

oder

```
<xsd:element name="elementName">
  <xsd:complexType>
    anonyme Typdefinition
  </xsd:complexType>
</xsd:element>
```

Dabei sind die Deklarationen innerhalb einer anonymen Typdefinition nur innerhalb dieses Bereichs und seiner Unterelemente sichtbar, nicht aber in Elementen, die außerhalb der Deklaration liegen. Zum Beispiel deklariert die folgende Zeile ein Element `TeilnehmerS` vom Typ `TeilnehmerSType`, wobei dieser Typ später im Schema definiert wird:

```
<xsd:element name="TeilnehmerS"
             type="TeilnehmerSType"/>
```

Äquivalent könnte man schreiben:

```
<xsd:element name="TeilnehmerS">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="Teilnehmer"
                   type="TeilnehmerType"
                   minOccurs="0"
                   maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="vorlesung" type="xsd:string"/>
  </xsd:complexType>
</xsd:element>
```

Häufigkeitseinschränkungen bei Elementen

<xsd:element> bekommt zwei Attribute, die festlegen, wie oft ein Element vorkommen darf. Mit xsd:minOccurs wird das Minimum, mit xsd:maxOccurs das Maximum festgelegt. Für optionales Vorkommen setzt man minOccurs auf 0. Für beliebiges Vorkommen dient das Schlüsselwort unbounded. Sonst sind beliebige Integerwerte zugelassen. Standard ist bei beiden Attributen 1. Zum Beispiel ist Teilnehmer im folgenden Beispiel optional, darf aber beliebig oft vorkommen:

```
<xsd:element name="Teilnehmer" type="TeilnehmerType"
             minOccurs="0" maxOccurs="unbounded"/>
```

4.1.2 Attributdeklaration

Attribute für ein Element werden innerhalb der Definition seines Typs deklariert. Z. B. deklarieren wir für das Element TeilnehmerS das Attribut vorlesung vom Typ xsd:string wie folgt:

```
<xsd:complexType name="TeilnehmerSType">
  ...
  <xsd:attribute name="vorlesung" type="xsd:string"/>
</xsd:complexType>
```

Attribute dürfen nur sog. *einfache Datentypen* (*simple types*) haben.

Häufigkeitseinschränkungen bei Attributen

Ein Attribut tritt in einem Element genau einmal oder garnicht auf. Angaben, ob ein Attribut optional, verpflichtend oder fest vorgegeben ist, werden in der Attributdefinition <xsd:attribute> durch die Besetzung der folgenden drei Attribute geregelt:

- use,
- default und
- fixed.

Das Attribut use von <xsd:attribute> kann die Werte required, optional oder sogar prohibited (verboten!) annehmen.

Mit `default` kann ein Standardwert angegeben werden. Dabei macht die Angabe eines Standardwerts nur bei `use='optional'` Sinn und ist in XSchema **nur in dieser Kombination** erlaubt.

Das Attribut `fixed` wird in Deklarationen von Elementen und Attributen verwendet, um sicherzustellen, dass Elemente oder Attribute einen bestimmten Wert haben. Die Verwendung von `fixed` und `default` in derselben Deklaration ist sinnlos und deshalb **verboten**. Jede Kombination aus diesen drei Attributen hat eine festgelegte Bedeutung. Wir geben hier nur zwei Beispiele an:

```
<xsd:element name='book'>
 <xsd:complexType>
  <!-- Das Attribut muss mit dem Wert 'de' vorkommen -->
  <xsd:attribute name='lang' type='xsd:string'
                 use='required' fixed='de'/>
  <!-- Optional: Standardwert 'latin' falls nicht
       angegeben.-->
  <xsd:attribute name='script' type='xsd:string'
                 use='optional' default='latin'/>
 </xsd:complexType>
</xsd:element>
```

4.1.3 Globale Elemente und Attribute

Globale Element- und Attribut-Deklarationen treten als Kind des Wurzelelements `xsd:schema` auf. Globale Deklarationen sind überall im Schema sichtbar und können an beliebigen Stellen mit dem Attribut `ref` referenziert werden. Globale **Elemente** haben weiterhin die besondere Eigenschaft, Wurzelelemente eines Instanzdokuments sein zu können. Sie haben jedoch die folgenden Einschränkungen:

- Sie müssen einfache wie komplexe Typen direkt benennen. D. h. die Verwendung des `ref`-Attributs ist nicht erlaubt (siehe weiter unten).
- Die Kardinalität darf in der Deklaration der globalen Elemente nicht eingeschränkt werden, d.h. die Attribute `minOccurs` und `maxOccurs` dürfen dort nicht auftreten, wohl aber in den Elementen, in denen sie mit `ref` verwendet werden.

Beispiel (aus XML Schema Primer):

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:po="http://www.example.com/PO1"
        targetNamespace="http://www.example.com/PO1">
  <element name="purchaseOrder"
           type="po:PurchaseOrderType"/>
  <element name="shipTo" type="po:USAddress"/>
  <element name="billTo" type="po:USAddress"/>
  <element name="comment" type="string"/>
  <element name="name" type="string"/>
  <element name="street" type="string"/>
  <complexType name="PurchaseOrderType">
    <sequence>
      <element ref="po:shipTo"/>
      <element ref="po:billTo"/>
      <element ref="po:comment" minOccurs="0"/>
      <!-- etc. -->
    </sequence>
  </complexType>
  <complexType name="USAddress">
    <sequence>
      <element ref="po:name"/>
      <element ref="po:street"/>
      <!-- etc. -->
    </sequence>
  </complexType>
  <!-- etc. -->
</schema>
```

Namenskonflikte

Es stellt sich die Frage: Was passiert, wenn man zwei Dingen denselben Namen gibt? Ein *Konflikt* liegt vor, wenn zwei Typen mit demselben Namen definiert werden. Dagegen liegt **kein** Konflikt vor, wenn:

- ein Element und ein Typ mit demselben Namen definiert werden.
- zwei Elemente innerhalb verschiedener Datentypdefinitionen denselben Namen haben. Z. B. ein `title`-Element innerhalb einer `bookType`-Definition und einer `personType`-Definition mit unterschiedlichem Inhalt bei beiden Angaben (man nennt solche Deklarationen auch *lokale Element-Deklarationen*).

- ein vordefinierter Datentyp und ein neu definierter Datentyp denselben Namen haben (da vordefinierte Datentypen zum XSchema-Namensraum gehören).

4.2 Vordefinierte Datentypen

Abb. 4–1 wurde dem Standard entnommen und enthält alle vordefinierten Datentypen und ihre Ableitungshierarchie in XSchema. Wir gehen auf einige ein.

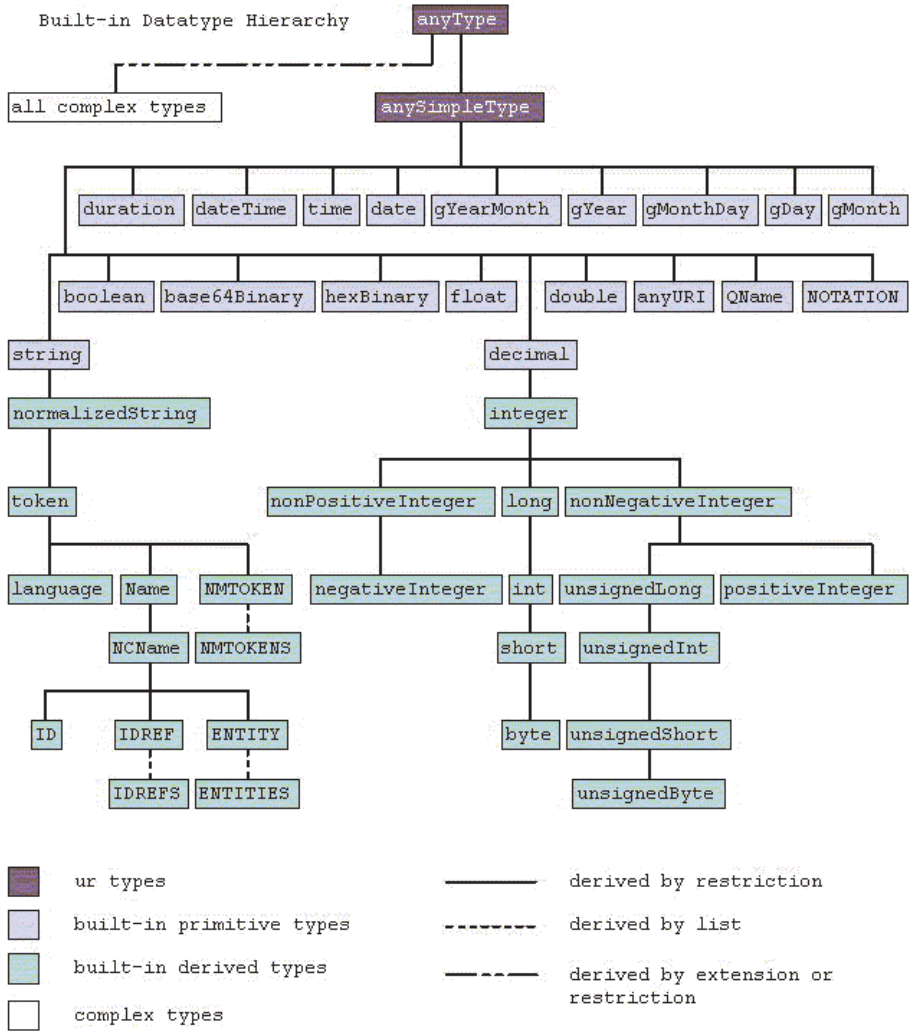


Abb. 4–1 *Vordefinierte Datentypen in XML Schema*

4.2.1 Strings

- **`string`**
  Beschreibt beliebige Zeichenketten, in denen **alle** Whitespaces[1] erhalten bleiben.
- **`normalizedString`**
  Hier werden Zeilenumbrüche und Tabulatoren in Leerzeichen umgewandelt.
- **`token`**
  Whitespaces werden jeweils zu einem einzigen Leerzeichen zusammengezogen. Whitespaces am Anfang und Ende werden entfernt.
- **`language`**
  Akzeptiert alle Abkürzungen für Sprachangaben wie `en-US` oder `de-DE`.

`NMTOKEN` und `NMTOKENS` haben die gleiche Bedeutung wie in DTDs. `Name` ist ähnlich wie `NMTOKEN`, darf aber nicht mit einer Ziffer, Bindestrich oder Punkt beginnen (wie `ID` in DTDs). `NCName` ähnlich wie `Name`, darf aber keinen Doppelpunkt enthalten (no colon). `QName` ist ein qualifizierter Name, der aus einem `NCName` besteht, dem ein Namensraumpräfix vorangestellt werden kann; zwischen beiden Teilen steht ein Doppelpunkt.

Die Datentypen `ID`, `IDREF`, `IDREFS`, `ENTITY`, `ENTITIES` und `NOTATION` wurden in XML Schema aus DTDs übernommen.

Weiterhin sind die folgenden Datentypen definiert:

- `anyURI` dient zur Aufnahme von URI-Angaben.
- `base64Binary` dient zur Aufnahme von binären Daten in Base64-Kodierung.
- `hexBinary` für hexadezimal kodierte Daten.

4.2.2 Zahlen

Für ganze Zahlen sind (unter anderem) die folgenden Datentypen vorgesehen:

1. Zeilenumbrüche, Tabulatoren und Leerzeichen.

- integer
- positiveInteger
- negativeInteger
- long
- unsignedLong

Dabei ist zu beachten, dass z. B. integer keine Einschränkung bzgl. der Länge einer Zahl macht. Von integer sind weitere Datentypen wie unsignedInt, unsignedShort usw. abgeleitet.

Für Gleitkommazahlen ist vorgesehen:

- decimal für Gleitkommazahlen beliebiger Länge!
- float für 32-Bit-Gleitkommazahlen und
- double für 64-Bit-Gleitkommazahlen.

Weiterhin dient der Typ boolean für Wahrheitswerte. Diese können entweder numerisch (0 oder 1) oder durch vordefinierte Bezeichner (true oder false) spezifiziert werden.

4.2.3 Zeit und Datum

Für Zeit- und Datumsangaben existieren z. B.: time, dateTime, date, gYear, gYearMonth, gDay, gMonth und gMonthDay.

Dabei werden bei einigen Datentypen für Zeit- und Datumsangaben vorhandene Standards unterstützt. Z. B. ist dateTime ein Datum nach dem ISO 8601 Standard. Das g steht für Gregorianischer Kalender.

4.3 Einfache Datentypen

Einfache Datentypen sind solche, die keine Elemente oder Attribute bekommen. Andere Datentypen sind komplex.

Man kann einfache Datentypen durch *Erweiterung* oder *Einschränkung* vorhandener Datentypen definieren und somit den Wertebereich eines Elements ziemlich genau spezifizieren. Die Einschränkung eines einfachen Datentyps neuerTyp geschieht mit:

```
<xsd:simpleType name="neuerTyp">
  <xsd:restriction base="basisTyp">
    Facetten zur Einschränkung des Basistyps
  </xsd:restriction>
</xsd:simpleType>
```

Datentypen bieten verschiedene Möglichkeiten für die Einschränkung (sog. *Facetten*). Diese sind im Standard für **jeden Datentyp** angegeben. Die Semantik einer Facette wechselt ggf. von Datentyp zu Datentyp. Es folgen hier ein paar Beispiele.

4.3.1 Einschränkung des Wertebereichs

Zur Einschränkung eines Wertebereichs dienen die Facetten:

- **xsd:minInclusive**,
- **xsd:maxInclusive**,
- **xsd:minExclusive** und
- **xsd:maxExclusive**.

Beispiel:

```
<xsd:simpleType name="lottoZahl">
  <xsd:restriction base="xsd:integer">
    <xsd:minExclusive value="0"/>
    <xsd:maxInclusive value="49"/>
  </xsd:restriction>
</xsd:simpleType>
```

4.3.2 xsd:enumeration

Damit wird für verschiedene Datentypen eine Liste von zulässigen Werten angegeben.

```
<xsd:simpleType name="bundesLaenderType">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="Baden-Württemberg"/>
    <xsd:enumeration value="Bayern"/>
    <xsd:enumeration value="Berlin"/>
    <xsd:enumeration value="Brandenburg"/>
    <xsd:enumeration value="Bremen"/>
    <xsd:enumeration value="Hamburg"/>
```

```
    <xsd:enumeration value="Hessen"/>
    <xsd:enumeration value="Mecklenburg-Vorpommern"/>
    <xsd:enumeration value="Niedersachsen"/>
    <xsd:enumeration value="Nordrhein-Westfalen"/>
    <xsd:enumeration value="Rheinland-Pfalz"/>
    <xsd:enumeration value="Saarland"/>
    <xsd:enumeration value="Sachsen"/>
    <xsd:enumeration value="Sachsen-Anhalt"/>
    <xsd:enumeration value="Schleswig-Holstein"/>
    <xsd:enumeration value="Thüringen"/>
  </xsd:restriction>
</xsd:simpleType>
```

4.3.3 xsd:length, xsd:maxLength und xsd:minLength

Spezifizieren die Länge einer Zeichenkette. Bei anderen Datentypen weicht die Bedeutung ab (z. B. Anzahl der Bytes bei `base64Binary`). `xsd:length` legt die Länge fest, während `maxLength` und `minLength` die maximale bzw. minimale Länge angeben. Beispiel:

```
<xsd:simpleType name="pwdType">
  <xsd:restriction base="xsd:string">
    <xsd:minLength value="5"/>
    <xsd:maxLength value="8"/>
  </xsd:restriction>
</xsd:simpleType>
```

4.3.4 xsd:pattern

Damit kann man einen String (auch für andere Datentypen wie `date` und `time`!) durch einen *regulären Ausdruck* einschränken. Reguläre Ausdrücke bieten ein sehr mächtiges Werkzeug zur genauen Beschreibung eines Wertebereichs. Im folgenden Beispiel definieren wir den Typ `bookID`, der mit dem Buchstaben `b` beginnen soll, gefolgt von 9 Ziffern, gefolgt von einer Ziffer oder `X`, also eine Zeichenkette der Art `b123456789X`:

```
<xsd:simpleType name="bookID">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="b[0-9]{9}[0-9X]"/>
  </xsd:restriction>
</xsd:simpleType>
```

4.3.5 Erweiterung zu Listentypen

Zusätzlich zu den vordefinierten Listentypen, können Listentypen für vorhandene Datentypen definiert werden. Beispiel:

```
<xsd:simpleType name="BundesLaenderListe">
  <xsd:list itemType="bundesLaenderType"/>
</xsd:simpleType>
```

Beispiel für ein Element von diesem Typ ist etwa:

```
<blaender>
  Baden-Württemberg Bayern Berlin Brandenburg Bremen
</blaender>
```

Listentypen können durch Facetten eingeschränkt werden, etwa im folgenden Beispiel für die Länge der Liste (genau sechs Einträge):

```
<xsd:simpleType name="SechsBundesLaenderListe">
  <xsd:restriction base="BundesLaenderListe">
    <xsd:length value="6"/>
  </xsd:restriction>
</xsd:simpleType>
```

4.3.6 Erweiterung durch Vereinigung

Mit `xsd:union` können mehrere einfache Datentypen zu einem neuen Datentyp vereinigt werden.

Im folgenden Beispiel kann die Angabe eines Bundeslandes entweder durch den Namen oder durch eine Postleitzahl aus diesem Bundesland erfolgen:

```
<xsd:simpleType name="postleitzahlType">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="[0-9]{5}"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="land_oder_plzType">
  <xsd:union
    memberTypes="postleitzahlType bundesLaenderType"/>
</xsd:simpleType>
```

Dabei bekommt das Attribut `memberTypes` eine Liste der Datentypen, aus denen der zu definierende Datentyp bestehen soll.

4.4 Komplexe Datentypen

Komplexe Datentypen sind solche, die Elemente oder Attribute haben.

Sie werden mit der Anweisung `xsd:complexType` definiert:

```
<xsd:complexType name="typeName">
  Deklaration der Elemente
  Deklaration der Attribute
</xsd:complexType>
```

Komplexe Typen können wiederum *komplexen* oder aber *einfachen Inhalt* haben (*complex content, simple content*).

4.4.1 Definition von einfachem Inhalt

Ein Element hat einen komplexen Typ mit einfachem Inhalt, wenn für das Element Attribute definiert werden, der Inhalt aber einfach ist (kein Markup). Beispiel:

```
<name anrede="Herr">Lehmann</name>
```

Zur Definition von solchen Datentypen kommt innerhalb von `xsd:complexType` die Anweisung `xsd:simpleContent`. Innerhalb von `xsd:simpleContent` wird ein einfacher Datentyp durch `xsd:extension` um die erlaubten Attribute erweitert. Beispiel:

```
<xsd:element name="name" type="nameType"/>
<xsd:complexType name="nameType">
  <xsd:simpleContent>
    <xsd:extension base="xsd:string">
      <xsd:attribute name="anrede" type="xsd:string"/>
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>
```

Für diese gängige Kombination ist die Deklaration in XML-Schema leider sehr langatmig.

4.4.2 Definition von komplexem Inhalt

Innerhalb der Definition (`xsd:complexType`) werden die Elemente und Attribute angegeben, die der zu definierende Typ enthalten darf oder muss, das sog. *Contentmodell*. Zusätzlich zu der Möglichkeit zur Festlegung der Häufigkeit (die Attribute `minOccurs` und `maxOccurs` von `xsd:element`) kann die Reihenfolge der Elemente festgelegt werden. Dazu existieren die folgenden drei Unterelemente von `xsd:complexType`:

- **`<xsd:sequence>`**
  Die angegebene Reihenfolge ist zwingend. Als Beispiel betrachten wir die Definition von `AbteilungType` zur Beschreibung einer Abteilung aus dem vorigen Kapitel:

```
<xsd:complexType name="AbteilungType">
 <xsd:sequence>
    <xsd:element name="AID" type="atomicType"/>
    <xsd:element name="ABTBEZ" type="atomicType"/>
    <xsd:element name="ORT" type="atomicType"/>
    <xsd:element name="MITARBEITER"
                 type="MitarbeiterType"/>
 </xsd:sequence>
 <xsd:attribute name="id" type="xsd:ID" use="required"/>
 <xsd:attribute name="nf2type" type="nf2typeType"
                use="optional" fixed="tuplet"/>
</xsd:complexType>
```

  Hier ist die Reihenfolge vorgeschrieben. Die Angabe im Instanzdokument:

```
<ABTEILUNG id="RID-4001">
  <AID id="RID-4003">FE</AID>
  <ORT id="RID-4005">Dresden</ORT>
  <ABTBEZ id="RID-4004">
    Forschung&amp;Entwicklung
  </ABTBEZ>
  ...
</ABTEILUNG>
```

  wäre damit ungültig, da `ORT` und `ABTBEZ` vertauscht wurden.

- **<xsd:choice>**
  Mit choice wählt man Elemente aus einer Gruppe. Durch Angabe von minOccurs (Vorgabe 1, Werte >= 0) und maxOccurs (Vorgabe 1, Werte >= 0, unbounded für beliebig häufiges Auftreten) kontrolliert man die Anzahl der Auswahlschritte.
  Im folgenden einfachen Beispiel hat ein Buch entweder ein authors- oder author-Element:

```
<xsd:complexType name="bookType">
  <xsd:sequence>
    <xsd:element name="title" type="xsd:string"/>
    <xsd:choice>
      <xsd:element name="authors"
        type="authorsType"/>
      <xsd:element name="author"
        type="authorType"/>
    </xsd:choice>
  </xsd:sequence>
</xsd:complexType>
```

  In dem Beispiel sehen wir auch, dass choice neben anderen Elementen in einer Sequenz auftauchen darf. Das ist auch der Fall im folgenden *HausTyp*.

```
<xsd:complexType name="HausTyp">
  <xsd:sequence>
    <xsd:element name="Lage" type="xsd:string"/>
    <xsd:element name="Grundstueck"
           type="GrundstueckT"/>
    <xsd:choice minOccurs="1" maxOccurs="100">
      <xsd:element name="Wohnen" type="WohnenT"/>
      <xsd:element name="Praxis" type="PraxisT"/>
    </xsd:choice>
    <xsd:element name="Bemerkung" type="xsd:string"/>
  </xsd:sequence>
  <xsd:attribute name="Preis"
       type="xsd:positiveInteger"/>
</xsd:complexType>
```

Er erlaubt in einem Haus bis zu 100 Wohnungen und Praxen (auch gemischt). In einer DTD würde die Angabe (ohne die nicht formulierbare Beschränkung auf 100 Wohn-/Praxiseinheiten) wie folgt lauten:

```
<!ELEMENT Haus (Lage, Grundstueck, (Wohnen | Praxis)+, Bemerkung)>
```

Neben dem Auftreten innerhalb eines `sequence`-Elements kann `choice` wiederum Unterelement von `choice` sein, d. h. eine Schachtelung ist erlaubt, allerdings nicht in Verbindung mit dem folgenden `all`-Element.

- **`<xsd:all>`**

  Die mit `all` geklammerten Unterelemente dürfen in beliebiger Reihenfolge, jedes davon höchstens einmal, vorkommen. Für die Attribute `minOccurs` und `maxOccurs` der `element`-Elemente innerhalb von `all` sind jeweils die Werte 0 und 1 zulässig. Sollen Unterelemente auch fehlen dürfen, muss bei den zugehörigen `element`-Elementen explizit `minOccurs="0"` gesetzt werden, der Vorgabewert ist 1.

  Für die Attribute `minOccurs` und `maxOccurs` des Elements `all` gilt: für `minOccurs` sind die Werte 0 und 1 erlaubt, für `maxOccurs` ist nur der Vorgabewert 1 zulässig, sodass man `maxOccurs` üblicherweise nicht angeben wird.

  Als Vaterelemente für `all` können dienen: `group`, `complexType`, `restriction` (sowohl `simpleContent` als auch `complexContent`), `extension` (sowohl `simpleContent` als auch `complexContent`), **nicht** zulässig sind `all`, `choice` und `sequence`, weil dies zu Mehrdeutigkeiten führen könnte.

  Beispiele:

```
<xsd:complexType name="authorType">
  <xsd:all>
    <xsd:element name="nachname" type="xsd:string" />
    <xsd:element name="vorname" type="xsd:string" />
  </xsd:all>
</xsd:complexType>
```

  `authorType` verlangt genau einen Vor- und Nachnamen in beliebiger Reihenfolge.

```
<xsd:complexType name="authorType">
  <xsd:all>
    <xsd:element name="nachname" type="xsd:string"
                 minOccurs="0"/>
    <xsd:element name="vorname" type="xsd:string"
                 minOccurs="0"/>
  </xsd:all>
</xsd:complexType>
```

Wieder können Vor- und Nachname in beliebiger Reihenfolge erscheinen, jeder höchstens einmal oder auch gar nicht.

4.4.3 Erweiterung von komplexen Elementen

Ähnlich der Einschränkung von einfachen Datentypen, können neue komplexe Datentypen durch Erweiterung oder Einschränkung definiert werden. Hierzu dienen die Anweisungen `xsd:extension` und `xsd:restriction`:

```
<xsd:complexType name='typName'>
  <xsd:complexContent>
    Einschränkung oder Erweiterung
  </xsd:complexContent>
</xsd:complexType>
```

Die Erweiterung von komplexen Typen mit komplexem Inhalt basiert auf der Zusammenfügung von Gruppen von Elementen und Attributen zu einer neuen Gruppe, die den Inhalt eines Datentyps definiert. Zur Erweiterung dient `xsd:extension`. In dem Beispiel erweitern wir den `authorType` um die Angabe einer Anrede:

```
<xsd:complexType name='newauthorType'>
  <xsd:complexContent>
    <xsd:extension base="authorType">
      <xsd:sequence>
        <xsd:element name="anrede" type="anredeType"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Die Einschränkung von komplexen Datentypen geschieht mit `xsd:restriction`, ist aber nicht sinnvoll zu verwenden, da der Datentyp im Prinzip neu definiert werden muss. Wir geben trotzdem ein Beispiel:

```
<xsd:complexType name='personType'>
  <xsd:sequence>
    <xsd:element name="Name" type="xsd:string"/>
    <xsd:element name="GebDatum" type="xsd:date"/>
    <xsd:element name="Beruf" type="berufType"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name='authorType'>
  <xsd:complexContent>
    <xsd:restriction base="personType">
      <xsd:sequence>
        <xsd:element name="Name" type="xsd:string"/>
        <xsd:element name="GebDatum" type="xsd:date"/>
      </xsd:sequence>
    </xsd:restriction>
  </xsd:complexContent>
</xsd:complexType>
```

4.4.4 Gemischter Inhalt

Treten im XML-Dokument Elemente und „normaler“ Text gemischt auf, etwa wie in

```
<Anrede>
  Sehr geehrte Frau, sehr geehrter Herr
  <Nachname>Meier</Nachname>
  !
</Anrede>
```

dann spricht man von gemischtem Inhalt. In XML Schema setzt man dafür dann das optionale Attribut `mixed` (default `false`) auf `true`:

```
<xsd:element name="Anrede">
  <xsd:complexType mixed="true">
    <xsd:sequence>
      <xsd:element name="Nachname" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

Man beachte aber, dass XML Schema weiterhin bei Verwendung des Attributs `mixed` Reihenfolge und Anzahl des Auftretens von Unterelementen genau kontrolliert. Freigestellt ist nur der eingestreute Text, im Beispiel vor und hinter `Nachname`, wo genau, lässt sich aber nicht vorschreiben.

4.4.5 anyType

`anyType` ist der oberste Typ in der XML-Schema-Typhierarchie. Elemente von diesem Typ haben keinerlei Einschränkung bezüglich ihres Inhalts. Man sollte deshalb vermeiden, Elemente von diesem Typ zu definieren.

`anyType` ist der Standardwert für das `type`-Attribut in Elementdeklaration, darf also weggelassen werden.

```
<xsd:element name="irgendwas"/>
```

ist also äquivalent zu:

```
<xsd:element name="irgendwas" type="xsd:anyType"/>
```

4.4.6 Leerer Inhalt

Datentypen für Elemente mit leerem Inhalt werden wie folgt definiert (abgekürzte Schreibweise):

```
<xsd:element name="preisInternational">
  <xsd:complexType>
    <xsd:attribute name="währung" type="xsd:string"/>
    <xsd:attribute name="wert" type="xsd:decimal"/>
  </xsd:complexType>
</xsd:element>
```

4.4.7 Referenzieren von Elementen und Attributen

Man kann innerhalb von Typdefinitionen auf Elemente oder Attribute verweisen, um diese zum Inhalt hinzuzufügen. Dazu kann man Elemente, die häufiger vorkommen, einmal **global** definieren und dann an mehreren Stellen auf diese Typdefinition verweisen. Beispiel:

```
<xsd:element name="title" type="xsd:string"/>
<xsd:element name="author" type="xsd:string"/>
<xsd:complexType name="bookType">
  <xsd:sequence>
    <xsd:element ref="title"/>
    <xsd:element ref="author"/>
    <xsd:element name="isbn" type="isbnType"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="articleType">
  <xsd:sequence>
    <xsd:element ref="title"/>
    <xsd:element ref="author"/>
    <xsd:element name="journal" type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>
```

5 XSLT und XPath

Nachdem wir die Grundlagen XML-basierter Sprachen und der XML-Dokumente gelernt haben, wenden wir uns der ersten Anwendungsmöglichkeit für solche Sprachen zu, der Umwandlung in andere Dokumente oder andere Sprachen mittels XSLT-Stylesheets. Dabei ist XSLT ein Teil der *Extensible Stylesheet Language* (XSL). XSL besteht aus drei Teilen (vergl. Kapitel 1):

- Formatting Objects (kurz fo),
- XPath [12] und
- XSLT [10] (XSL-Transformationen).

In diesem Kapitel behandeln wir XSLT und XPath.

5.1 XSLT als XML-Dokument

XSLT-Stylesheet-Dateien haben meistens die Endung „`.xsl`“. XSLT-Stylesheets sind XML-Dokumente und beginnen deshalb mit der XML-Instruktion:

```
<?xml version='1.0' ...?>
```

XSLT verwendet den Namensraum `http://www.w3.org/1999/XSL/Transform`. Meistens wird dafür die Abkürzung „`xsl:`“ verwendet.

Das Rootelement eines Stylesheets ist `<xsl:stylesheet>`. Alternativ kann man `<xsl:transform>` verwenden. Wichtige Attribute des Rootelements sind:

- **`version`**
  Die Version eines Stylesheets. Wir verwenden die Version 1.0.[1]
- **`xmlns:xsl`**
  Angabe des Namensraums.
- **`id`**
  Damit kann eine eindeutige ID für das Stylesheet vergeben werden.

Aus einem XML-Dokument wird wie folgt auf ein Stylesheet, z. B. auf `mystylesheet.xsl`, verwiesen:

```
<?xml-stylesheet type='text/xsl' href='mystylesheet.xsl'?>
```

5.1.1 Erstes Beispiel: Hallo Welt

Bevor wir in die Details einsteigen, schauen wir uns ein sehr einfaches Beispiel an (`hallo.xml`):

```
<?xml version='1.0' standalone='yes'?>
<?xml-stylesheet type='text/xsl' href='hallo.xsl'?>
<greeting>
  Hallo Welt!
</greeting>
```

Die Datei `hallo.xsl` enthält das folgende Stylesheet, das dieses XML-Dokument in HTML transformiert:

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<xsl:stylesheet version='1.0'
  xmlns:xsl='http://www.w3.org/1999/XSL/Transform'>
  <xsl:template match='/'>
    <html>
      <head>
        <title>Beispiel</title>
      </head>
      <body>
        <xsl:apply-templates/>
      </body>
    </html>
  </xsl:template>
  <xsl:template match='greeting'>
```

1. XSLT Version 2.0 erschien 2007 als W3C Recommendation.

```
   <h1><xsl:value-of select='.'/></h1>
  </xsl:template>
</xsl:stylesheet>
```

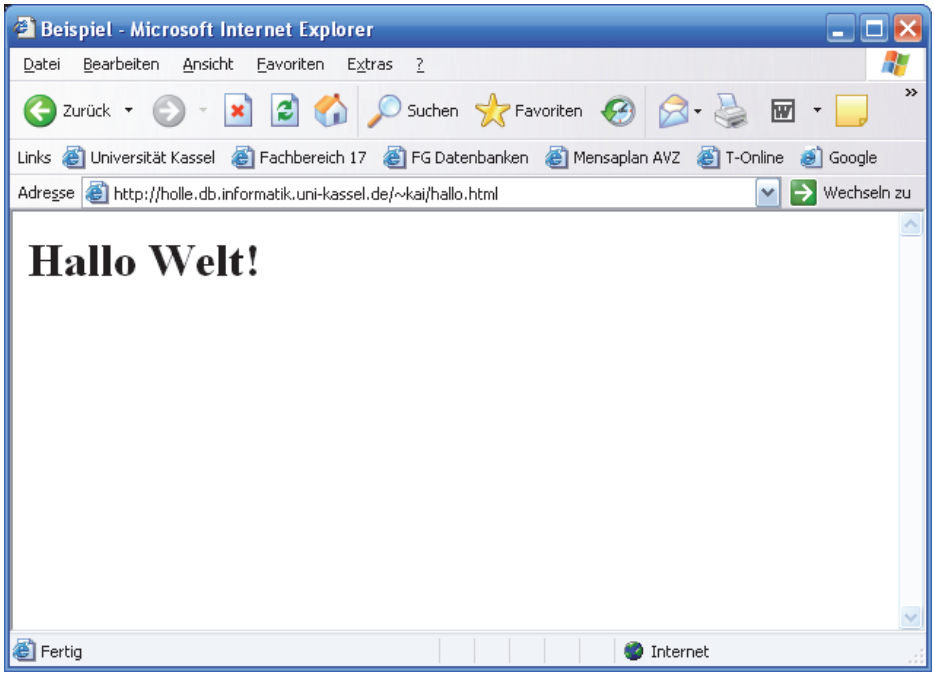


Abb. 5–1 *Darstellung von „hallo.html“ mit Internet Explorer 6*

5.2 XPath-Datenmodell

Grundlage für die Transformierung bildet das XPath-Datenmodell. Ein XML-Dokument wird in diesem Datenmodell zu einem *Baum*. Eine Besonderheit ist, dass die Wurzel des Baums im Datenmodell nicht dem Rootelement des XML-Dokuments entspricht, sondern als abstrakter Knoten eine Stufe höher liegt. Der Baum enthält sieben unterschiedliche Arten von Knoten:

- **Element**
  Ein Element kann andere Elemente und zusätzlich jeden anderen Knotentyp mit Ausnahme des Wurzelelements enthalten.

- **Attribut**
  Attribute haben keine weiteren Kinder. Sie werden als Blatt-Knoten dargestellt.
- **Text**
  Texte sind immer Kinder anderer Knoten. In einem Element können beliebig viele Textknoten vorkommen.
- **Kommentar**
  Kommentare bilden in XPath auch eigene Baumknoten.
- **Steuerungsanweisung**
  Steuerungsanweisungen werden in XPath auch als Baumknoten betrachtet.
- **Namensraum**
  Namensräume werden als besondere Knoten betrachtet, da sie über das ganze Dokument Bedeutung haben.
- **Wurzel**
  Der Wurzelknoten ist ein abstrakter Knoten über dem Wurzelelement. Er enthält das ganze Dokument mit allen Deklarationen vor dem Wurzelelement.

Wir schauen uns ein einfaches Dokument an und betrachten den zugehörigen Baum:

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<!-- So könnten die nächsten E-Mails aussehen! -->
<nachricht>
  <?interpbilder?>
  <subj>XML-Dokumente als Bäume</subj>
  <text format="text">heute haben wir gelernt ...</text>
</nachricht>
```

Abb. 5–2 enthält alle Knotentypen in diesem Datenmodell. Diese Baumsicht ist für das Verständnis von XSLT von zentraler Bedeutung. Man bezeichnet das Eingabedokument als *Quellbaum* und das Ausgabedokument als *Ergebnisbaum*. Der Stylesheet-Prozessor (als Teil eines Browsers oder getrennt wie Xalan) durchläuft nun den XML-Baum (Quellbaum) und versucht, an jedem Knoten eine passende Regel zu finden, wobei bei mehreren passenden die **spezifischste** (genauest passende,

mehr dazu später) genommen wird. Der **Inhalt** des `<xsl:template>`-Elements bestimmt, welche Ausgabe produziert wird.

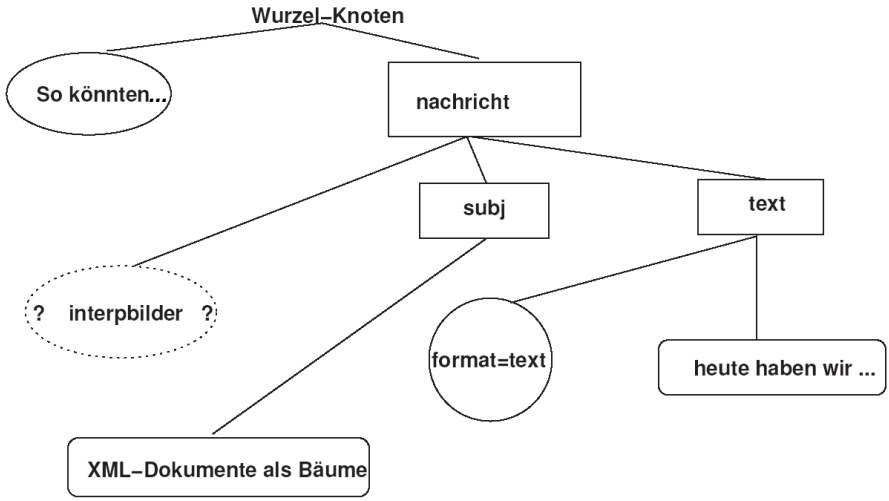


Abb. 5–2 *Baumdarstellung eines Dokuments*

5.3 Funktionsprinzip von Stylesheets

Wie bei anderen Stylesheet-Sprachen besteht ein XSLT-Stylesheet aus *Regeln* (`<xsl:template>`), die zu Elementen passen. Jede Regel beschreibt, welche Ausgabe produziert werden soll, bzw. welche weiteren Anweisungen ausgeführt werden. Zum Beispiel besagt die Regel

```
<xsl:template match='greeting'>
  <h1><xsl:value-of select="."/></h1>
</xsl:template>
```

dass aus dem Element `greeting` im Eingabedokument, die Ausgabe

```
<h1><xsl:value-of select="."/></h1>
```

zu erzeugen ist, wobei `value-of` den Wert eines Knotens liefert, hier einen Text, und zwar für den ausgewählten Knoten „.“, d. h. den aktuellen Knoten.

Innerhalb eines Templates werden weitere Regeln auf Sohnelemente mit `<xsl:apply-templates select='node-set-expression'/>` angewandt. Dabei kann man mit dem optionalen Attribut `select` die zu verarbeitenden Knoten bestimmen. Standardmäßig werden **alle** Unterknoten genommen. Im folgenden Beispiel wird innerhalb des Templates für die Wurzel `<xsl:apply-templates/>` aufgerufen, um die Unterknoten zu verarbeiten:

```
<xsl:template match='/'>
  <html>
    <head>
      <title>Beispiel</title>
    </head>
    <body>
      <xsl:apply-templates/>
    </body>
  </html>
</xsl:template>
```

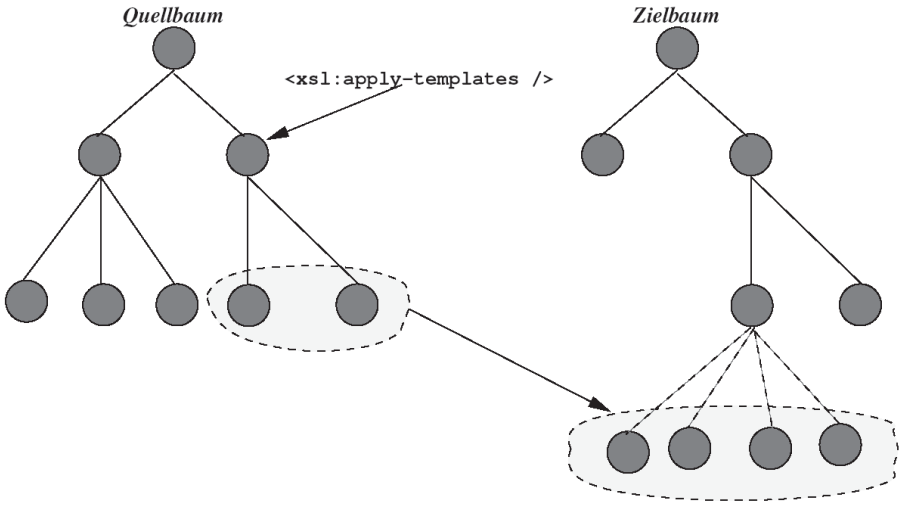


Abb. 5–3 *Quell- und Zielbaum*

Die XSLT-Transformation beginnt mit einer aktuellen Knotenliste, die einen einzigen Eintrag enthält: den Wurzelknoten (`/`). Die Verarbeitung geht dann **rekursiv** wie folgt weiter:

- Für jeden Knoten `X` in der aktuellen Knotenliste sucht der Prozessor nach allen Templates, die *potentiell* auf diesen Knoten passen. Aus dieser Liste wird die **passendste** Regel genommen.
- Das ausgewählte Template wird mit diesem Knoten ausgeführt. Der erzeugte Teilbaum wird als Teilbaum des Knotens im Zielbaum an der Stelle eingefügt, wo `<xsl:apply-templates>` aufgerufen wurde.
- Wenn das Template ein Element `<xsl:apply-templates>` enthält, so wird eine neue aktuelle Knotenliste aus den Söhnen des aktuellen Knotens erzeugt, und der Prozess wiederholt sich.

Man kann `<xsl:template>` und `<xsl:apply-templates>` mit Methodendefinition und Methodenaufruf vergleichen.

5.4 Einfache XSLT-Elemente

Wir betrachten erst ein paar einfache XSLT-Elemente, die in der Regel für die Erstellung von einfachen Stylesheets ausreichen.

5.4.1 xsl:template und xsl:apply-templates

```
<xsl:template match='pattern'> ... </xsl:template>
<xsl:apply-templates select='node-set-expression'/>
```

Die `<xsl:template>`-Elemente sind die eigentlichen Verarbeitungsregeln. Das im Starttag angegebene Muster `pattern` bestimmt, auf welche XML-Elemente das Template angewandt wird. Dabei bestehen solche *pattern* aus sog. *Lokalisierungspfaden*, die eine Untermenge der XPath-Ausdrücke (*expressions*) bilden. Wie die Ausgabeformatierung aussehen soll, steht zwischen Start- und Endtag:

```
<xsl:template match='/'>
  <html>
    <head>
      <title>Beispiel</title>
    </head>
    <body>
      <xsl:apply-templates/>
    </body>
  </html>
</xsl:template>
```

Wichtig an diesem Beispiel ist auch die Anweisung `<xsl:apply-templates/>`, die den Prozessor zwingt, im zu formatierenden Element alle auftretenden **Sohnelemente** ebenfalls dem Template-Matching zu unterziehen.

5.4.2 xsl:value-of

```
<xsl:value-of select='expression'/>
```

`<xsl:value-of>` dient zur Ausgabe von Knotenwerten. Das Attribut `select` bekommt als Wert einen XPath-Ausdruck, der den auszugebenden Wert berechnet.

Der *Wert eines Elements* besteht aus allen geparsten Texten im Element zusammen mit den Werten der Nachfolgerelemente (rekursiv!). Anders ist es, wenn der Knoten, der gerade zur Verarbeitung ansteht oder für den die `<xsl:value-of>`-Formatieranweisung gilt, kein Element ist, sondern ein Attribut, ein Namensraum, eine Prozessorinstruktion oder ein Kommentar. Dann ist der Wert, der ausgegeben wird, der Attributwert, die URI für den Namensraum, der Datenteil (ohne Target und ohne „`<?`“ und „`?>`“) der Prozessorinstruktion bzw. der Text des Kommentars ohne „`<!--`“ und „`-->`“.

`<xsl:value-of>` ist nicht nur für die Ermittlung von Knotenwerten nützlich, sondern auch für Berechnungen. In dem folgenden Beispiel geben wir die Anzahl der Teilnehmer aus einer Teilnehmerliste aus (vergl. Beispiel aus Kapitel 1):

```
<xsl:template match='TeilnehmerS'>
  <p>Gesamt: <xsl:value-of select='count(//Teilnehmer)'/>
     Teilnehmer</p>
  <table>
    ...
  </table>
</xsl:template>
```

5.5 Auswahl von Knoten

Die Wahl der zu verarbeitenden Knoten wird durch XPath-Ausdrücke angegeben. Einige Elemente bekommen einen sog. Lokalisierungspfad,

eine Untermenge der XPath-Ausdrücke. Im Standard wird von *pattern* und *expression* gesprochen. Das Attribut `match` des Elements `<xsl:template>` bekommt z. B. einen Lokalisierungspfad als Wert (*pattern*), dagegen erhält das Attribut `select` von `<xsl:value-of>` einen XPath-Ausdruck (*expression*).

5.5.1 Lokalisierungspfade

Einige der Beispiele werden sich auf das folgende Dokument beziehen (`kunden.xml`):

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<kunden>
  <kunde id='0001'>
    <titel>Dipl.-Ing.</titel>
    <name>Müller</name>
    <vorname>Udo</vorname>
    <adresse>
      <strasse>Lindenstraße</strasse>
      <hnr>12</hnr>
      <plz>56001</plz>
      <ort>Köln</ort>
    </adresse>
    <wunschliste>
      <buecher>
        <buch id='4711'>
          <titel>Die Zauberflöte</titel>
        </buch>
        <buch id='4712'>
          <titel>Die Schachnovelle</titel>
        </buch>
      </buecher>
      <cds>
        <cd id='6601'>
          <titel>Die Zauberflöte</titel>
        </cd>
      </cds>
    </wunschliste>
  </kunde>
  <kunde id='0002'>
    ...
  </kunde>
  <kunde id='0003'>
```

```
    ...
  </kunde>
  <kunde id='1000'>
    ...
  </kunde>
</kunden>
```

Lokalisierungspfade sind XPath-Ausdrücke, die einen oder mehrere Knoten in einem XML-Dokument lokalisieren.

Der Kontext

Ähnlich wie die Angabe eines Verzeichnis- oder Dateinamens im Dateisystem, wird ein XPath-Ausdruck in einem sog. *Kontext* ausgewertet. Dabei besteht dieser Kontext im wesentlichen aus dem *aktuellen Knoten* im Quellbaum.

Weiterer wichtiger Bestandteil des Kontexts sind zwei Integerwerte, die sog. *Kontextposition* und *Kontextgröße*. Diese sind wichtig, wenn man eine Gruppe von Elementen verarbeitet. Man könnte z. B. einen XPath-Ausdruck schreiben, der alle `<li>`-Elemente in einem Dokument behandelt. Die Kontextgröße wäre hier die Anzahl dieser Elemente, die Kontextposition das gerade zu verarbeitende Element innerhalb dieser Menge.

Absolute und relative Lokalisierungspfade

Man unterscheidet weiterhin zwischen *absoluten* und *relativen* Lokalisierungspfaden. Absolute Pfade beginnen mit dem Wurzelknoten, relative mit dem Kontextknoten:

```
<!--Relativ: Titel des aktuellen Buches-->
<xsl:template match="buch">
  <xsl:value-of select="titel"/>
</xsl:template>
<!--Absolut: Name des ersten Kunden im Dokument -->
<xsl:template match="/">
  <xsl:value-of select="/kunden/kunde[1]/name"/>
</xsl:template>
```

Aufbau von Lokalisierungspfaden

Ein Lokalisierungspfad besteht aus einer Reihe von *Schritten*, die den Pfad vom Ausgangspunkt immer weiterführen und durch `/` voneinander getrennt werden. Ein einzelner Schritt hat drei Teile:

- eine *Achse*, die beschreibt in welche Richtung es geht,
- einen *Knotentest*, der spezifiziert, auf welche Arten von Knoten der Schritt zutrifft,
- einen Satz von **optionalen** *Prädikaten*, die also Boolesche Tests darstellen, um Kandidaten auszuwählen.

Die Tabellen 5–1 und 5–2 zeigen Knotenachsen und Knotentest in XPath.

Achsentyp	Bedeutung
`self`	Der Kontextknoten selbst.
`child`	Alle direkten Kinder des Kontextknotens.
`descendant`	Alle Unterknoten des Kontextknotens. D. h. alle Kinder und Kindeskinder. Allerdings ohne die Attribute und Namensräume.
`descendant-or-self`	Der Kontextknoten selbst zusätzlich zu `descendant`.
`parent`	Vaterknoten
`ancestor`	Alle Vaterknoten bis zur Wurzel.
`ancestor-or-self`	Alle Vaterknoten inklusive des Kontextknotens selbst.
`following`	Alle Knoten, die dem Kontextknoten auf einer beliebigen Ebene im Dokument folgen.

Achsentyp	Bedeutung
`following-sibling`	Wie `following`, aber auf derselben Ebene (gleicher Vaterknoten).
`preceding`	Alle Knoten, die dem Kontextknoten auf einer beliebigen Ebene im Dokument vorhergehen.
`preceding-sibling`	Wie `preceding`, aber auf derselben Ebene (gleicher Vaterknoten).
`attribute`	Alle Attributknoten des Kontextknotens.
`namespace`	Alle Namensraumknoten des Kontextknotens.
`.`	Abkürzung für `self`
`@`	Abkürzung für `attribute`
`..`	Abkürzung für `parent`
`//`	Abkürzung für `/descendant-or-self::node()`, also beliebiges Vorkommen im Dokument. Z. B. steht `//titel` für alle `titel`-Elemente im Dokument.

Tab. 5–1 *Knotenachsen in XPath*

Knotentest	Bedeutung
`QName`	Die Angabe eines Namens (z. B. `book`) bedeutet bei einer Attributachse Attribute mit diesem Namen, in einer Namensraumachse Namensräume mit diesem Namen, in allen anderen Fällen Elemente mit diesem Namen.
`*`	In einer Attributachse jedes Attribut, in einer Namensraumachse jeder Namensraum, in allen anderen Achsen jedes Element.
`node()`	Jeder Knoten.
`text()`	Jeder Textknoten.
`processing-instruction()`	Jede Steueranweisung.
`comment()`	Jeder Kommentarknoten.
`/`	Der Wurzelknoten.
`@*`	Jedes Attribut.

Tab. 5–2 *Knotentests in XPath*

Die Angabe eines Pfades kann in der *langen Schreibweise* in der Form

`Achse``::`*`Knotentest`*`[`*`Prädikat1`*`]...[`*`PrädikatN`*`]`

erfolgen oder in der *kurzen Schreibweise*, wo die Achsenangabe verkürzt wird und `::` wegfällt. In den meisten Fällen kommt man mit der kurzen Schreibweise aus.

Das Weglassen der Achsenangabe ist gleichbedeutend mit `child`, also Kinder des aktuellen Knotens. Somit sind `titel`, `./titel` und `child::titel` drei äquivalente Pfade für das Element `titel` des aktuellen Knotens.

Wir geben ein einfaches Beispiel an, bevor wir Prädikate behandeln. Das im Beispiel enthaltene Element `<xsl:text>` stellt den Textinhalt in die Ausgabe.

```
<!-- Kunden, die direkte Nachfahren des XML-
     Wurzelelements sind -->
<xsl:template match='/kunden/kunde'>
  <p>
    <!-- Gib deren ID-Attribut aus -->
    <xsl:value-of select="@id"/><xsl:text> : </xsl:text>
    <!-- Und den Namen -->
    <xsl:value-of select="name"/>
  </p>
</xsl:template>
```

Prädikate werden innerhalb eckiger Klammern angegeben und können in jedem Schritt vorkommen. Jedes Prädikat wird ausgewertet und liefert einen Booleschen Wert. Jeder Knoten, der den Test in einem Prädikat besteht (zusätzlich zu Knotentest und Achsenangabe), wird in den Knotensatz einbezogen.

Eine Zahl als Prädikat wählt Knoten, die eine *Position* gleich dieser Zahl haben. Zum Beispiel gibt die folgende Zeile den Titel des ersten Buchs innerhalb eines Elements `lit` aus:

```
<xsl:value-of select='/lit/book[1]/titel'/>
```

Sonst können beliebige XPath-Ausdrücke innerhalb eines Prädikates vorkommen. Dabei werden die folgenden Vergleichsoperatoren verwendet:

Operator	Rückgabe
expr1 = *expr2*	Wahr, wenn *expr1* = *expr2* (Zeichenkette oder numerisch).

Operator	Rückgabe
expr1 != *expr2*	ungleich
expr1 < *expr2*	kleiner
expr1 <= *expr2*	kleinergleich
expr1 > *expr1*	größer
expr1 >= *expr1*	größergleich

Im folgenden Beispiel wählen wir alle Kunden aus Kassel:

```
<xsl:for-each
     select="/kunden/kunde[adresse/ort='Kassel']">
  ...
</xsl:for-each>
```

Ebenfalls können die Booleschen Operatoren `and`, `or`, `not(`*expr*`)`, `true()` und `false()` verwendet werden. Beispiel:

```
<!--Alle Mengen im Schema -->
<xsl:if test="//xsd:complexType/xsd:attribute
               [@name='nf2type' and
                @use='optional' and
                @default='sett']">
  ...
</xsl:if>
```

Wir verweisen zuletzt auf die sog. Attributwertvorlage. Mit

```
<table border="{@size}"/>
```

kann man den Wert des Attributs `size` aus dem Quelldokument als Wert des Attributs `border` im Zieldokument setzen.

Ausdrücke

Innerhalb eines Prädikates können XPath-Funktionen und Operatoren aufgerufen werden. Wir erwähnen hier einige wichtige. Weitere Operatoren können dem Standard entnommen werden.

- **Knotensätze**
  Auf Knotensätze können folgende Funktionen angewandt werden:

Funktion	Bedeutung
`count(`*`Knotensatz`*`)`	Die Anzahl der Knoten im Knotensatz.
`sum(`*`Knotensatz`*`)`	Summe der numerischen Werte aller Knoten im Knotensatz.
`position()`	Die Nummer des Kontextknotens im **Kontext-Knotensatz**.
`last()`	Die Nummer des letzten Knotens im Kontext-Knotensatz.
`current()`	Liefert den aktuellen Knoten zurück.
`name(`*`Knotensatz?`*`)`	Name des ersten Knotens in Dokumentenordnung im angegebenen Knotensatz; fehlt die Knotensatzangabe wird der Name des Kontextknotens zurückgeliefert.

Im folgenden geben wir z. B. den Namen des letzten Kunden aus:

```
<xsl:value-of
     select='/kunden/kunde[position() = last()]/name'/>
```

Die Funktion `current()` bedarf weiterer Erklärung. In den meisten Fällen sind der Kontextknoten und der aktuelle Knoten identisch und somit z. B. die folgenden beiden Elemente:

```
<xsl:value-of select='current()'/>
<xsl:value-of select='.'/>
```

Innerhalb eines Prädikats ist dies jedoch nicht der Fall, weil „.“ in der Auswertung eines Pfadausdrucks jeweils den gerade erreichten Knoten bezeichnet, sich in der Auswertung also schrittweise ändert. Beispiel:

```
<xsl:template match='kunde'>
  <!-- Alle Kunden, die denselben Titel haben -->
  <xsl:for-each
       select='//kunde[./titel = current()/titel and
                       ./@id  != current()/@id]'/>
    ...
  </xsl:for-each>
</xsl:template>
```

In `http://www.w3schools.com/xsl/func_current.asp` wird das folgende Beispiel gegeben:

```
<xsl:apply-templates select="//cd[@titel=current()/@ref]"/>
```

das alle `cd`-Elemente verarbeitet, deren `titel`-Attribute denselben Wert haben wie das `ref`-Attribut des gegenwärtigen Knotens.
Im Unterschied dazu verarbeitet

```
<xsl:apply-templates select="//cd[@titel=./@ref]"/>
```

alle `cd`-Elemente, bei denen `titel`-Attributwert und `ref`-Attributwert gleich sind.

- **Zahlen**
  Für Zahlen existieren die „gängigen“ Operatoren wie `+`, `-`, `*`, `div` und `mod`. Der folgende Ausdruck trifft z. B. auf alle Personen zu, deren Position in der Menge `kunden` gerade ist (jeden zweiten Kunden):
  `/kunden/kunde[(position() mod 2) = 0]`

- **Zeichenketten**
  Für Zeichenketten betrachten wir die folgenden Funktionen:

Funktion	Rückgabe
`concat(`*`String1`*`, `*`String2`*`, ...)`	Verkettung der Argumente.

Funktion	Rückgabe
`substring(`*`String`*`, `*`Zahl1`*`, `*`Zahl2?`*`)`	Teilstring von *`String`*. Dabei gibt *`Zahl1`* die Stelle vom Anfang des Strings, *`Zahl2`* die Länge ab *`Zahl1`* an; falls *`Zahl2`* fehlt, geht der Teilstring bis zum Ende von *`String`*.
`substring-after (`*`String`*`, `*`Suchstring`*`)`	Teilstring ab dem Ende *`Suchstring`* bis zum Ende von *`String`*. Liefert eine leere Zeichenkette, falls *`Such-string`* nicht in *`String`* vorkommt.
`substring-before (`*`String`*`, `*`Suchstring`*`)`	Teilstring vom Anfang von *`String`* bis zum Anfang von *`Suchstring`*.
`translate (`*`String`*`, `*`Suchstring`*`, `*`Ersatzstring`*`)`	Liefert *`String`* zurück, wobei alle in *`Suchstring`* vorkommenden Zeichen durch Zeichen an der entsprechenden Position in *`Ersatzstring`* ersetzt werden.

Wichtig ist darauf zu achten, dass die Position des **ersten** Zeichens in einem String 1 ist (nicht 0). Beispiele aus dem W3C-Standard [12]:
`substring("12345", 0, 3)` liefert `12`
`substring-before("1999/04/01","/")` liefert `1999`
`substring-after("1999/04/01","/")` liefert `04/01`
`substring-after("1999/04/01","19")` liefert `99/04/01`
`translate("--aaa--","abc-","ABC")` liefert `AAA`
`translate("bar","abc","ABC")` liefert `BAr`

Weiterhin existieren die folgenden Operatoren:

Funktion	Rückgabe
`contains(`*`String`*`, `*`Substring`*`)`	Wahr, wenn *`Substring`* innerhalb *`String`* enthalten ist.

Funktion	Rückgabe
`starts-with(`*`String`*`,` *`Substring`*`)`	Wahr, wenn *`String`* mit *`Substring`* beginnt.
`string-length(`*`String`*`)`	Anzahl der Zeichen in *`String`*.

Beispiele

Wir geben im folgenden weitere Beispiele, um den Umgang mit XPath zu verdeutlichen.

Regelangabe für den Wurzelknoten:

```
<xsl:template match="/"> ... </xsl:template>
```

Gebe Wert des Kontextknotens aus:

```
<xsl:value-of select="."/>
```

Gebe Wert vom Unterknoten `name` des Kontextknotens aus:

```
<xsl:value-of select="./name"/>
```

Alle `filename`-Elemente im Dokument kursiv ausgeben:

```
<xsl:template match="filename">
  <i><xsl:value-of select='.'/></i>
</xsl:template>
```

Ausgabe des Namens des Kontextknotens:

```
<xsl:value-of select='name()'/>
```

Für beliebige Attribute gebe Attributname und -wert aus:

```
<xsl:template match='@*'>
  <p>
    <xsl:text>Attributname: </xsl:text>
    <xsl:value-of select="name()"/><br/>
    <xsl:text>Attributwert: </xsl:text>
    <xsl:value-of select="."/>
  </p>
</xsl:template>
```

5.5.2 Auflösung von Regelkollisionen

Wir haben schon erwähnt, dass bei mehreren passenden Regeln, die passendste für einen Knoten genommen wird. Der Standard legt dazu fest, dass es die spezifischste Regel ist und gibt eine Rangfolge zur Berechnung vor, die sog. *Priorität*. Eine Regel mit einem spezifischen Namen hat eine höhere Priorität als eine ebenfalls passende Regel mit Wildcard `*`. Wenn zum Beispiel `kunde/wunschliste/cds` der Kontextknoten ist, dann hat `cds` Vorrang vor `*`. Beachte, dass `kunde/wunschliste/cds` und `kunde/wunschliste/*` jedoch gleichwertig sind.

Ebenfalls haben Regeln, die mehr als nur einen Knotennamen enthalten, eine höhere Priorität als solche, die nur aus einem Knotennamen bestehen. Zum Beispiel hat `kunde[adresse/ort='Kassel']` bei dem passenden Kunden eine höhere Priorität als die allgemeine Regel `kunde`. Weiterhin hat `kunden/kunde` Vorrang vor `kunde`. Die Regeln `kunde[adresse/ort='Kassel']` und `kunden/kunde` sind jedoch gleichwertig. Wenn mehrere Regeln die gleiche Priorität besitzen, kann der XSLT-Prozessor den Konflikt lösen, indem er die zuletzt definierte verwendet.

`<xsl:template>` bekommt weiterhin das Attribut `priority` mit dem Standardwert 0 und Wertebereich zwischen -1 und 1, mit dem die Priorität einer Regel explizit gesetzt werden kann (je größer die Zahl, desto höher die Priorität).

5.5.3 Standardregeln

Wenn für einen Knoten keine Regel aus dem Stylesheet passt, so wird eine sog. *Standardvorlage* verwendet. XSLT stellt für jeden Knotentyp eine Standardregel zur Verfügung:

- **Wurzel**

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

- **Element**

```
<xsl:template match='*'>
  <xsl:apply-templates/>
<xsl:template>
```

- **Attribut- und Textnodes**

```
<xsl:template match="text()|@*">
  <xsl:value-of select="."/>
</xsl:template>
```

- **Prozessoranweisungen und Kommentare**
  Standardregel: ignorieren!

```
<xsl:template
  match="processing-instruction()|comment()"/>
```

Man sollte aber darauf achten, dass diese Regeln erst zur Ausführung gebracht werden müssen.

Beispiel: Das einfachste Stylesheet, das alle Texte im Dokument ausgibt (`alleTexte.xsl`):

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<xsl:stylesheet version='1.0'
 xmlns:xsl='http://www.w3.org/1999/XSL/Transform'>
 <xsl:template match='/'>
  <html>
   <head>
    <title>Template mit Standardregeln</title>
   </head>
   <body>
    <xsl:apply-templates/>
   </body>
  </html>
 </xsl:template>
</xsl:stylesheet>
```

Die Anwendung dieses Stylesheets auf das zu Beginn von Abschnitt 5.5.1 angegebene Dokument `kunden.xml` liefert die folgende Ausgabe:

Abb. 5–4 *Anwendung von „alleTexte.xsl“ auf „kunden.xml“*

Frage: Werden die Attribute ebenfalls ausgegeben?

Hinweis: Mit `<xsl:value-of>` werden Elemente mit gemischtem Inhalt (Inhalt enthält Text und Unterelemente) so ausgegeben, dass die Textanteile dieses Elements und aller Unterelemente in Baumreihenfolge (genauer: In-Order-Reihenfolge) erscheinen. Gibt es Regeln für die Unterelemente, kommen diese nicht zur Anwendung, weil ein `<xsl:apply-templates/>` fehlt.

Gibt man nur `<xsl:apply-templates/>` an (wie unten gezeigt), erscheinen der umgebende Text und die Ausgabe der Unterelemente gemäß der Regeln für diese Unterelemente, hier *italic*. Wären Attribute im Element enthalten, würden diese nicht angezeigt, weil ein fehlendes `select`-Attribut in `<apply-templates/>` nur Text- und Elementknoten als Kinderknoten auswählt.

Hat das Element `<txt>` z. B. den Inhalt

```
<txt>Geben Sie die Lösung als Dateien
<file>A1IhrName.dtd</file> und <file>A2IhrName.xml</file>
ab.</txt>
```

so würde man im Stylesheet `<txt>` und `<file>` wie folgt behandeln:

```
<xsl:template match="file">
  <i><xsl:value-of select='.'/></i>
</xsl:template>
<xsl:template match="txt">
  <xsl:apply-templates/>
</xsl:template>
```

Womit dann die Regel für `<file>` ebenfalls angewandt wird. Die Ausgabe lautet:

```
<?xml version="1.0" encoding="UTF-8"?>
Geben Sie die Lösung als Dateien
<i>A1IhrName.dtd</i> und <i>A2IhrName.xml</i>
ab.
```

5.6 Weitere XSLT-Elemente

5.6.1 xsl:element

`<xsl:element` **`name='`*`element-name`*`'`**`> ... </xsl:element>`

Wird verwendet, um ein Element im Ausgabedokument zu erzeugen. Der Wert des Attributs `name` enthält den Namen des zu erzeugenden Elements. Der Wert des erzeugten Elements ist der Inhalt von `<xsl:element>`, also die Angaben zwischen `<xsl:element>` und `</xsl:element>` im Stylesheet. `<xsl:element>` ist z. B. nützlich, um Elementnamen dynamisch zu berechnen. Beispiel:

Eingabedokument:

```
<angabe art='beruf' wert='student'/>
<angabe art='alter' wert='24'/>
```

Regel im Stylesheet:

```
<xsl:template match='angabe'>
  <xsl:element name='{@art}'>
    <xsl:value-of select='@wert'/>
  </xsl:element>
</xsl:template>
```

Ausgabedokument:

```
<beruf>student</beruf>
<alter>24</alter>
```

5.6.2 xsl:attribute

```
<xsl:attribute name='attr-name'> ... </xsl:attribute>
```

Dient zur Erzeugung von dynamischen Attributwerten für die erzeugten Elemente im Ausgabedokument. Dies ist notwendig, wenn der Wert eines Attributs durch eine Berechnung aus Angaben des Eingabedokuments ermittelt wird.

Im folgenden Beispiel erzeugen wir aus der Eingabe

```
<email>
  <![CDATA[morad@db.informatik.uni-kassel.de]]>
</email>
```

im Eingabedokument die Ausgabe:

```
<a href='mailto:morad@db.informatik.uni-kassel.de'>
  <tt>morad@db.informatik.uni-kassel.de</tt>
</a>
```

Die Regel im Stylesheet lautet:

```
<xsl:template match='email'>
  <xsl:element name='a'>
    <xsl:attribute name='href'>
      <xsl:text>mailto:</xsl:text><xsl:value-of select='.'/>
    </xsl:attribute>
    <tt><xsl:apply-templates/></tt>
  </xsl:element>
</xsl:template>
```

5.6.3 xsl:text

```
<xsl:text disable-output-escaping="(yes | no)">
  ...
</xsl:text>
```

Normalerweise kann man Text im Stylesheet so angeben, wie er in der Ausgabe produziert werden soll. Will man aber Leerzeichen, Tabulatoren etc. beibehalten oder zusätzliche Leerzeichen produzieren, so verwendet

man <xsl:text>. Sehr nützlich ist in diesem Zusammenhang das optionale Attribut disable-output-escaping, wodurch man Sonderzeichen wie < und & produzieren kann. Beispiel:

```
<xsl:text disable-output-escaping='yes'>
  cout &lt;&lt; "Ausgabe von Strings in C++";
</xsl:text>
```

5.6.4 xsl:copy

```
<xsl:copy> ... </xsl:copy>
```

Kopiert den aktuellen Knoten (nicht rekursiv) in den Ergebnisbaum. Der Inhalt von <xsl:copy> besteht aus Anweisungen, die Attribute und ggf. eine Auswahl der Unterelemente dem erzeugten Element hinzufügen.

<xsl:copy> kann man verwenden, um Teile eines Dokuments zu kopieren. Wir erzeugen im folgenden Beispiel aus kunden.xml eine Liste mit leeren Kundenelementen und deren IDs:

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<xsl:stylesheet version='1.0'
  xmlns:xsl='http://www.w3.org/1999/XSL/Transform'>
  <xsl:template match="/">
    <kunden><xsl:apply-templates/></kunden>
  </xsl:template>
  <xsl:template match='/kunden/kunde'>
    <xsl:copy><xsl:apply-templates select="@*"/></xsl:copy>
  </xsl:template>
  <xsl:template match='@*'>
    <xsl:copy/>
  </xsl:template>
</xsl:stylesheet>
```

Angewandt auf kunden.xml bekommen wir die Ausgabe:

```
<?xml version="1.0" encoding="UTF-8"?>
<kunden>
  <kunde id="0001"/>
  <kunde id="0002"/>
  <kunde id="0003"/>
  <kunde id="1000"/>
</kunden>
```

5.6.5 xsl:copy-of

```
<xsl:copy-of select='expression'/>
```

Kopiert einen Knoten mit all seinen Unterknoten in den Zielbaum. Dies ist sehr nützlich, wenn bestimmte Teile aus dem Quelldokument in das Zieldokument unverändert übertragen werden sollen.

Hat man in einer Sprache für die Erstellung von Manuskripten z. B. das Element `<table>` für die Erstellung von Tabellen aus HTML übernommen, so braucht man in einem Stylesheet zur Umwandlung in HTML lediglich die Tabelle zu kopieren. Somit lautet die Regel für `<table>`:

```
<xsl:template match="table">
  <br/>
  <xsl:copy-of select="."/>
  <br/>
</xsl:template>
```

5.6.6 xsl:comment

Kommentare im Ausgabedokument können mit `<xsl:comment>` erzeugt werden. Beispiel:

```
<xsl:comment>
  Visualisierung in SVG, Datum 23.11.2012
</xsl:comment>
```

produziert den Kommentar im Ausgabedokument:

```
<!--Visualisierung in SVG, Datum 23.11.2012-->
```

5.6.7 xsl:processing-instruction

```
<xsl:processing-instruction name='target'>
  ...
</xsl:processing-instruction>
```

Steueranweisungen können mit XSLT ebenfalls erzeugt werden. Dabei können Attribute für die erzeugte Steueranweisung durch einfachen Text im Stylesheet produziert werden (nicht mit `<xsl:attribute>`). Beispiel:

Quelldokument:

```
<db product='MySQL'>
  ...
</db>
```

Im Stylesheet:

```
<xsl:processing-instruction name='db-binding'>
  dbms="<xsl:value-of select='/db/@product'/>"
</xsl:processing-instruction>
```

Dies produziert die PI:

```
<?db-binding dbms="MySQL"?>
```

5.7 XSLT-Kontrollstrukturen

Neben der rekursiven Ausführung von Regeln, existieren in XSLT mehrere Elemente für Kontrollstrukturen. Diese bekommen meistens einen oder mehrere XPath-Ausdrücke in einem Attributwert (z. B. `test` oder `select`) übergeben.

5.7.1 xsl:if

```
<xsl:if test='boolean-expression'> ... </xsl:if>
```

Führt den Inhalt zwischen Start- und Endtag aus, wenn der Testwert wahr ist. Im folgenden Beispiel wird hinter jedem Namen, außer dem letzten, ein Komma angehängt:

```
<xsl:template match="namelist/name">
  <xsl:apply-templates/>
  <xsl:if test="not(position()=last())">, </xsl:if>
</xsl:template>
```

Im folgenden wird in der erzeugten Tabelle jede zweite Zeile gelb gefärbt:

```
<xsl:template match="item">
  <tr>
    <xsl:if test="position() mod 2 = 0">
      <xsl:attribute name="bgcolor">yellow</xsl:attribute>
    </xsl:if>
    <xsl:apply-templates/>
  </tr>
</xsl:template>
```

5.7.2 xsl:for-each

```
<xsl:for-each select='node-set-expression'>
  ...
</xsl:for-each>
```

Diese Schleifenkonstruktion wendet die im Inhalt angegebenen Formatieranweisungen nacheinander auf alle durch `select` ausgewählten Knoten an. Obwohl die Schleife meist mit Templates ersetzbar wäre, ist es oft leichter, nach einer Auswahl des gesuchten Vaters mit `<xsl:template>` dessen Söhne gezielt mit `<xsl:for-each>` zu verarbeiten. `<xsl:for-each>` kann ein `<xsl:sort>`-Element enthalten, um die Knoten zu sortieren.

Im folgenden Beispiel erzeugen wir innerhalb der Regel für einen Kunden eine Liste aller Bücher auf seiner Wunschliste:

```
<xsl:template match='kunden/kunde'>
  <xsl:text>Bücherwünsche von: </xsl:text>
  <xsl:value-of select="name"/>
  <ol>
    <xsl:for-each select="wunschliste/buecher/buch">
      <li>
        <xsl:value-of select="titel"/>
      </li>
    </xsl:for-each>
  </ol>
</xsl:template>
```

5.7.3 xsl:choose

```
<xsl:choose>
  <xsl:when test='boolean-expression'> ... </xsl:when>
  ...
  <xsl:otherwise> ... </xsl:otherwise>
</xsl:choose>
```

Dieses Element wird in aller Regel mehrere Fallunterscheidungen anbieten, die jeweils mit `<xsl:when test='...'>` eingeleitet werden. Zuletzt gibt man ein **optionales** `<xsl:otherwise>`-Element an, um nichtgefangene Knoten zu verarbeiten.

Im folgenden Beispiel erzeugen wir in Abhängigkeit der Wunschliste eines Kunden eine „Beschreibung“ dieses Kunden:

```
<xsl:template match='kunden/kunde'>
  <p>
    <xsl:value-of select="name"/><xsl:text> ist </xsl:text>
    <xsl:choose>
      <xsl:when test="count(.//cd) > 0 and
                      count(.//buch) = 0">
        <xsl:text>musikalisch.</xsl:text>
      </xsl:when>
      <xsl:when test="count(.//cd) = 0 and
                      count(.//buch) > 0">
        <xsl:text>lesebegeistert.</xsl:text>
      </xsl:when>
      <xsl:when test="count(.//cd) > 0 and
                      count(.//buch) > 0">
        <xsl:text>vielseitig interessiert.</xsl:text>
      </xsl:when>
      <xsl:when test="count(.//cd) = 0 and
                      count(.//buch) = 0">
        <xsl:text>bescheiden.</xsl:text>
      </xsl:when>
    </xsl:choose>
  </p>
</xsl:template>
```

und bekommen die folgende Ausgabe in HTML:

```
<p>M&uuml;ller ist vielseitig interessiert.</p>
<p>Bubart ist lesebegeistert.</p>
<p>Wegner ist musikalisch.</p>
<p>Ahmad ist bescheiden.</p>
```

5.8 Benannte Templates, Variablen und Parameter

Für wiederkehrende Funktionen kann man ein sog. benanntes Template erstellen, um dieses an verschiedenen Stellen im Stylesheet zu verwenden. Diese werden wie andere Templates mit `<xsl:template>` erstellt, erhalten jedoch ein Attribut `name`, das den Namen des Templates angibt. Im folgenden Beispiel erzeugen wir das benannte Template `linkLeiste`:

```
<xsl:template name='linkLeiste'>
  ...
</xsl:template>
```

Dieses kann dann an anderen Stellen im Stylesheet wie folgt aufgerufen werden:

```
<xsl:call-template name='linkLeiste'>
  ...
</xsl:call-template>
```

Sehr wichtig zu erwähnen ist, dass der Kontext innerhalb einer benannten Vorlage von dem **aufrufenden Element** kommt.

Innerhalb einer benannten Vorlage können sog. Parameter definiert werden. Im Beispiel definieren wir den Parameter `titelP` mit Standardwert `Links`:

```
<xsl:template name='linkLeiste'>
  <xsl:param name="titelP">Links</xsl:param>
  ...
</xsl:template>
```

Der Name eines Parameters wird durch das Attribut `name` angegeben. Der Inhalt von `<xsl:param>` enthält den Standardwert, der verwendet wird, falls der Aufruf ohne Parameter erfolgt.

Die Übergabe eines Wertes geschieht innerhalb von `<xsl:call-template>`.Wir übergeben hier den Wert `Software-Links` an den Parameter `titelP` und überschreiben damit den Standardwert:

```
<xsl:call-template name='linkLeiste'>
  <xsl:with-param name="titelP"
                  select="'Software-Links'"/>
</xsl:call-template>
```

Alternativ könnte man schreiben:

```
<xsl:call-template name='linkLeiste'>
  <xsl:with-param name="titelP">
    Software-Links
  </xsl:with-param>
</xsl:call-template>
```

Variablen können überall im Stylesheet definiert werden. Der Name Variable ist jedoch nicht wörtlich zu nehmen, da es sich in Wirklichkeit um **Konstanten** handelt. Im folgenden Beispiel definieren wir die Variable `font` mit Wert `Times`:

```
<xsl:variable name="font">Times</xsl:variable>
```

oder

```
<xsl:variable name="font" select="'Times'"/>
```

Man achte darauf, dass der Wert eines Parameters oder einer Variable konstant bleibt und **nicht** neu gesetzt werden kann.

Um den Wert einer Variable oder eines Parameters zu bekommen, verwendet XSLT das `$`-Zeichen:

```
<xsl:value-of select='$font'/>
```

Als Attributwert kann eine Variable oder ein Parameter in geschweiften Klammern angegeben werden:

```
<text zeichensatz='{$font}'>Text-Element</text>
```

5.8.1 Beispiel

Angenommen, wir hätten das folgende Dokument mit XML-Daten über einen Projektablauf (`project.xml`):

```
<?xml version='1.0' encoding="ISO-8859-1"?>
<TASKS nf2type="sett">
  <TASK nf2type="ptuplet">
    <TASKID>START</TASKID>
    <DESCRIPTION>project kickoff</DESCRIPTION>
    <DUR>0</DUR>
    <ES>0</ES>
    <LS>0</LS>
    <EF>0</EF>
    <LF>0</LF>
    <ISOTIME>01-01-1998</ISOTIME>
    <REQUIRES nf2type="sett" state="empty"/>
  </TASK>
  ...
  <TASK nf2type="ptuplet">
```

```
   <TASKID>GDESIGN</TASKID>
   <DESCRIPTION>global design</DESCRIPTION>
   <DUR>15</DUR>
   <ES>21</ES>
   <LS>26</LS>
   <EF>36</EF>
   <LF>41</LF>
   <ISOTIME>10-09-1998</ISOTIME>
   <REQUIRES nf2type="sett">
     <TASK>GSPEC</TASK>
     <TASK>STAFF-PREP</TASK>
   </REQUIRES>
  </TASK>
  ...
</TASKS>
```

In einem Stylesheet, das daraus ein GANTT-Diagramm (Projektablaufplan) in SVG produziert, könnte man z. B. benannte Templates wie `drawTask` (einen einzelnen Task zeichnen) oder `drawCoords` (Koordinaten erstellen) schreiben. Wir schauen uns `drawCoords` an, das nur ein Koordinatensystem mit fester Größe erstellt (Vereinfachter Ausschnitt des Stylesheets `project-to-svg.xsl`):

```
<?xml version='1.0' encoding="ISO-8859-1"?>
<xsl:stylesheet version='1.0'
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns="http://www.w3.org/2000/svg">
<xsl:output method="xml" indent="yes"
            media-type="image/svg"/>

...

<xsl:template name="drawCoords">
  <xsl:param name="xend">0</xsl:param>
  <xsl:param name="yend">0</xsl:param>
  <xsl:variable name="CXBegin" select="5"/>
  <xsl:variable name="CYBegin" select="5"/>
  <line x1="{$CXBegin}" y1="{$CYBegin}"
        x2="{$xend}"  y2="{$CYBegin}"
        style="stroke-width:3;fill:none; stroke:red"/>
  <line x1="{$CXBegin}" y1="{$CYBegin}"
        x2="{$CXBegin}" y2="{$yend}"
        style="stroke-width:3;fill:none; stroke:red"/>
  <line x1="{$xend}" y1="{$CYBegin}"
```

```
        x2="{$xend}" y2="{$yend}"
        style="stroke-width:3;fill:none; stroke:red"/>
  <line x1="{$CXBegin}"  y1="{$yend}"
        x2="{$xend}" y2="{$yend}"
        style="stroke-width:3;fill:none; stroke:red"/>
  ...
</xsl:template>

...

<xsl:template match='/'>
  <svg>
    ...
    <xsl:call-template name="drawCoords">
      <xsl:with-param name="xend" select="600"/>
      <xsl:with-param name="yend" select="400"/>
    </xsl:call-template>
    ...
    <xsl:for-each select="TASKS/TASK">
      ... <!-- Hier wird drawTask aufgerufen. -->
    </xsl:for-each>
  </svg>
</xsl:template>
</xsl:stylesheet>
```

Das produzierte SVG-Dokument sieht wie folgt aus:

```
<?xml version="1.0" encoding="UTF-8"?>
<svg xmlns="http://www.w3.org/2000/svg">
  <line style="stroke-width:3;fill:none; stroke:red"
        y2="5" x2="600" y1="5" x1="5"/>
  <line style="stroke-width:3;fill:none; stroke:red"
        y2="400" x2="5" y1="5" x1="5"/>
  <line style="stroke-width:3;fill:none; stroke:red"
        y2="400" x2="600" y1="5" x1="600"/>
  <line style="stroke-width:3;fill:none; stroke:red"
        y2="400" x2="600" y1="400" x1="5"/>
  ...
</svg>
```

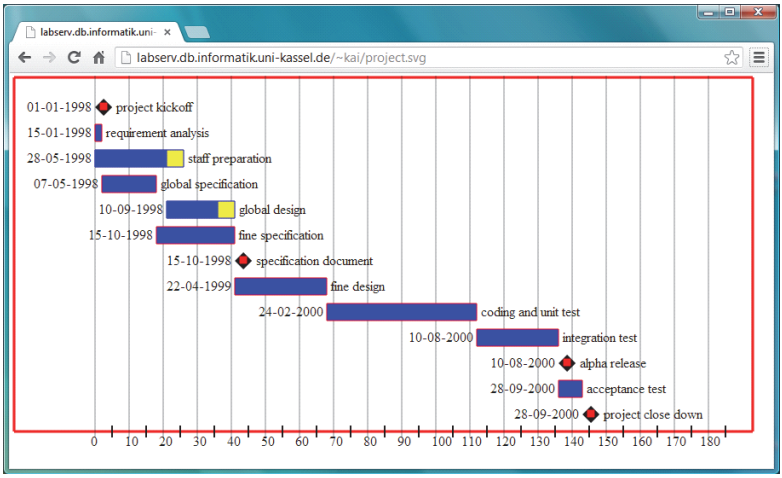


Abb. 5–5 *Das SVG-Dokument in Chrome*

5.9 Ausgabemethode

Mit dem Element `<xsl:output>` kann die *Ausgabemethode* festgelegt werden. Dieses hat die folgende Form (verkürzt):

```
<xsl:output
  method = "xml" | "html" | "text"
  encoding = string
  omit-xml-declaration = "yes" | "no"
  standalone = "yes" | "no"
  doctype-public = string
  doctype-system = string
  indent = "yes" | "no"
  media-type = string />
```

Wichtig sind die folgenden Attribute:

- `method` legt die Ausgabemethode fest.
- `encoding` erstellt ein `encoding`-Attribut im Zieldokument.
- `indent` gibt an, ob der erzeugte Text im Zieldokument eingerückt werden soll.
- `media-type` gibt einen MIME-Typ für das Zieldokument an.

Falls dieses Element fehlt, so versucht der Stylesheet-Prozessor zu entscheiden, welche Ausgabemethode zu verwenden ist. Dabei wird `html` verwendet, falls das Rootelement des Zieldokuments `<html>` ist, sonst gilt `xml` als Standardmethode.

Ein Stylesheet-Prozessor verhält sich unterschiedlich je nach Ausgabemethode. Nur bei der Ausgabemethode `xml` wird Wohlgeformtheit gefordert. Es muss sich nicht um ein wohlgeformtes vollständiges XML-Dokument handeln, sondern nur um eine wohlgeformte externe allgemeine Entität mit gemischtem Inhalt. Das bedeutet, dass man spätestens dann, wenn man die Ausgabe als Inhalt in ein XML-Wurzelelement setzt, ein wohlgeformtes vollständiges XML-Dokument erhält.

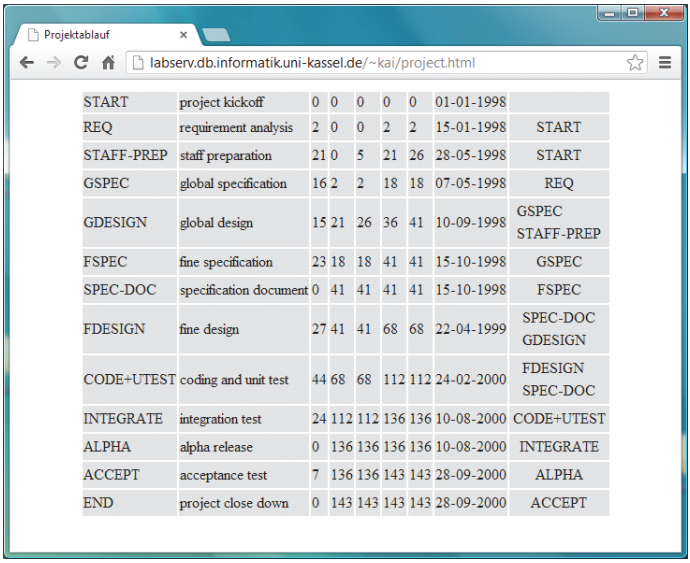
Die Details für jede Ausgabemethode können dem Standard entnommen werden.

5.10 Ein vollständiges Beispiel

Wir betrachten nochmal das Dokument `project.xml`. Mit dem Attribut `nf2type` beschreiben wir die Struktur eines Elements. So besagt `nf2-type='sett'`, dass das Element eine Menge ist. Wir entwickeln ein Stylesheet, das anhand der Angaben in diesem Attribut eine Darstellung des Dokuments in Form einer geschachtelten HTML-Tabelle erzeugt (`project-to-nf2.xsl`):

```
<?xml version="1.0" encoding='ISO-8859-1'?>
<xsl:stylesheet version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

 <xsl:template match="/">
  <html>
   <head><title>Projektablauf</title></head>
   <body>
    <table align="center" border="0">
     <xsl:apply-templates/>
    </table>
   </body>
  </html>
 </xsl:template>
```

```
  <!-- Mengen, Listen und Multimengen -->
  <xsl:template match="*[@nf2type='sett' or
                  @nf2type='listt' or @nf2type='msett']">
    <td align="left" valign="top">
      <table align="center" border="0">
        <xsl:apply-templates/>
      </table>
    </td>
  </xsl:template>

  <!-- Implizite Tupel -->
  <xsl:template match="*[@nf2type='ptuplet']">
    <tr bgcolor="#DDDDDD">
      <xsl:apply-templates/>
    </tr>
  </xsl:template>

  <!-- Explizite Tupel -->
  <xsl:template match="*[@nf2type='gtuplet']">
    <xsl:apply-templates/>
  </xsl:template>

  <!-- Atomare Werte -->
  <xsl:template match="*[not(@nf2type)]">
    <td><xsl:value-of select="."/></td>
  </xsl:template>

  <!-- Atomare Werte als direkte Kinder einer Menge -->
  <xsl:template
          match="*[@nf2type='sett']/*[not(@nf2type)]">
    <tr bgcolor="#DDDDDD">
      <td><xsl:value-of select="."/></td>
    </tr>
  </xsl:template>

</xsl:stylesheet>
```



START	project kickoff	0	0	0	0	01-01-1998	
REQ	requirement analysis	2	0	0	2	15-01-1998	START
STAFF-PREP	staff preparation	21	0	5	21	26	28-05-1998	START
GSPEC	global specification	16	2	2	18	18	07-05-1998	REQ
GDESIGN	global design	15	21	26	36	41	10-09-1998	GSPEC STAFF-PREP
FSPEC	fine specification	23	18	18	41	41	15-10-1998	GSPEC
SPEC-DOC	specification document	0	41	41	41	41	15-10-1998	FSPEC
FDESIGN	fine design	27	41	41	68	68	22-04-1999	SPEC-DOC GDESIGN
CODE+UTEST	coding and unit test	44	68	68	112	112	24-02-2000	FDESIGN SPEC-DOC
INTEGRATE	integration test	24	112	112	136	136	10-08-2000	CODE+UTEST
ALPHA	alpha release	0	136	136	136	136	10-08-2000	INTEGRATE
ACCEPT	acceptance test	7	136	136	143	143	28-09-2000	ALPHA
END	project close down	0	143	143	143	143	28-09-2000	ACCEPT

Abb. 5–6 *Darstellung mit Chrome*

6 SVG - Scalable Vector Graphics

Wir beziehen uns in dieser Übersicht auf das Tutorial von Kevin Lindsey, das im Web unter `http://www.kevlindev.com/tutorials/index.htm` verfügbar ist. Ergänzend lag das Buch

> Marcel Salathé: SVG Scalable Vector Graphics für professionelle Einsteiger, Markt+Technik Verlag, München, 2001

vor.

Lindsey beginnt mit einem SVG-Grundgerüst:

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20010904//EN"
     "http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink">
  <!-- SVG content goes here -->
</svg>
```

Darin besagt die Deklaration aus Zeile 1, dass es sich um ein XML-Dokument nach XML 1.0 handelt. Der Aufbau des Dokuments wird durch die W3C DTD für SVG 1.0 – ein Riesenschema – bestimmt (Zeilen 2 und 3).

Zeile 4 ist das Wurzelelement, das laut DTD ein `svg`-Element sein muss. Es definiert auch Namensräume, in Zeile 5 für XLink, um auf externe Elemente zugreifen zu können, z. B. Bilder.

Zeile 6 ist Platzhalter für den eigentlichen SVG-Code. Zeile 7 schließt das `svg`-Element ab.

6.1 Elementare Graphikelemente

Hier sind einige einfache Formen aus Lindsey.

	`<circle cx="25" cy="25" r="20" fill="red"/>`
	`<rect x="5" y="5" width="40" height="40" fill="red"/>`
	`<rect x="5" y="5" rx="5" ry="5" width="40" height="40" fill="red"/>`
	`<ellipse cx="25" cy="25" rx="20" ry="10" fill="red"/>`
	`<line x1="5" y1="5" x2="45" y2="45" stroke="red"/>`
	`<polyline points="5,5 45,45 5,45 45,5" stroke="red" fill="none"/>`

```
<polygon points="5,5 45,45 5,45 45,5"
         stroke="red" fill="none"/>
```

```
<path d="M5,5 C5,45 45,45 45,5"
      stroke="red" fill="none"/>
```

```
<defs>
  <rect id="rect" width="15" height="15"
        fill="red"/>
</defs>
<use x="5"  y="5"  xlink:href="#rect"/>
<use x="30" y="30" xlink:href="#rect"/>
```

Einige Kommentare zu den Beispielen, wobei es weniger darauf ankommt, alle Features und Formen zu besprechen, als ein Gefühl für die Syntax von SVG zu erlangen.

```
<circle cx="25" cy="25" r="20" fill="red"/>
```

Kreise werden durch die Koordinaten des Mittelpunkts und einen Radius definiert. Fehlen die Koordinaten, wird (0, 0) genommen (linke obere Ecke des Viewports). Ohne Angaben zu Einheiten wird in *Pixel* gerechnet. Lindsey schreibt das Füllattribut direkt hinein.

Salathé definiert einen „Stil“ für sein Kreisbeispiel:

```
<g style="fill:lightgray;stroke:black;stroke-width:5">
  <circle cx="120" cy="90" r="80"/>
  <circle r="25"/>
</g>
```

Dies produziert auch eine schwarze Umrandung.

Das **Rechteck** wird durch eine Höhe, Breite und die Position der linken oberen Ecke definiert:

```
<rect x="5" y="5" width="40" height="40" fill="red"/>
<rect x="5" y="5" rx="5" ry="5" width="40" height="40"
      fill="red"/>
```

Die zweite Variante produziert ein Rechteck mit abgerundeten Ecken, deren Rundungsradien mit `rx` und `ry` (analog zur Ellipse) angegeben werden. Wird nur ein Wert genannt, gilt er für `rx` und `ry`.

```
<ellipse cx="25" cy="25" rx="20" ry="10" fill="red"/>
```

Ellipsen haben einen Mittelpunkt (center) und zwei Radien `rx` und `ry`.

Zunächst zeichnen wir eine **gerade Linie** durch Angabe der Anfangs- und Endkoordinaten. Eine Linienbreite wird hier nicht definiert (Default 1 Pixel), ggf. legt man wieder einen Stil `style="stroke:red;stroke-width:2"` fest.

```
<line x1="5" y1="5" x2="45" y2="45" stroke="red"/>

<polyline points="5,5 45,45 5,45 45,5" stroke="red"
          fill="none"/>
```

Ein **Polygonzug** (Salathé: Polylinie) ist eine (meist offene) Folge von Punkten, die zusammenhängend durch Linien verbunden sind. Jeder Punkt besteht aus einer x- und y-Koordinate. Daher muss die Liste der Punkte eine gerade Anzahl von Werten enthalten. Die x- und y-Koordinaten eines Punkts können zwecks besserer Lesbarkeit durch Komma getrennt werden. Ohne weitere Angabe wird der Zug mit der Farbe Schwarz gefüllt, das Attribut `fill="none"` unterdrückt dies.

```
<polygon points="5,5 45,45 5,45 45,5" stroke="red"
         fill="none"/>
```

Analog ist das **Polygon** die geschlossene Variante des Zugs mit der impliziten Verbindung des letzten angegebenen Punkts mit dem Startpunkt. Ohne eigenes Fill-Attribut wird das Polygon schwarz gefüllt.

Formen kann man als Spezialfall eines Pfades betrachten, der im Stil eines Plotters einen Stift zu einem angegebenen Punkt bewegt (*moveto*) und ihn dort absenkt. Mit weiteren Befehlen (lineto, curveto, horizontal lineto, vertical loneto) zieht der Stift eine Linie oder Kurve.

```
<path d="M5,5 C5,45 45,45 45,5" stroke="red fill="none"/>
```

Im Beispiel oben wird eine kubische Bézierkurve gezeichnet. Die Koordinate nach `M` ist Startpunkt der Linie (*moveto*), danach kommen die beiden Kontrollpunkte (bei quadratischen Bézierkurven nur ein Kontrollpunkt), dann der Endpunkt. Der Buchstabe `C` ist Abkürzung für *curveto*, `Q` wäre Abkürzung für die quadratische Variante (*quadratic curveto*).

Die Punktangaben hier waren *absolute* Werte (bezogen auf den Ursprung (0, 0). Verwendet man Kleinbuchstaben `m`, `c`, `q`, `l`, ... dann sind die folgenden Koordinatenangaben *relativ* zum vorherigen Punkt.

Erwähnen sollte man auch Bogenkurven als Teile geneigter Ellipsen.

Zuletzt das Beispiel mit den zwei kleinen (identischen) Rechtecken.

```
<defs>
  <rect id="rect" width="15" height="15" fill="red"/>
</defs>
<use x="5"  y="5"  xlink:href="#rect"/>
<use x="30" y="30" xlink:href="#rect"/>
```

Der entscheidende Aspekt hier ist die Trennung in Definition des Graphikteils im `defs`-Element und der Gebrauch mit `use`-Elementen, die das zu benutzende Element referenzieren und die aktuellen Koordinaten für die Plazierung bestimmen.

Genauso können im `use`-Element Stilangaben `fill="..."` usw. gemacht werden.

Abschließend zu diesem Kurzausblick zu „Formen“ die Erzeugung eines grünen Kreises (wie erste Form oben) mittels JavaScript und DOM-Schnittstelle:

```
var svgns = "http://www.w3.org/2000/svg";
function makeShape(evt) {
  if (window.svgDocument == null)
    svgDocument = evt.target.ownerDocument;
  var shape = svgDocument.createElementNS(svgns,"circle");
  shape.setAttributeNS(null,"cx",25);
  shape.setAttributeNS(null,"cy",25);
  shape.setAttributeNS(null,"r",20);
  shape.setAttributeNS(null,"fill","green");
```

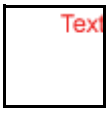
```
  svgDocument.documentElement.appendChild(shape);
}
```

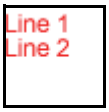
Lindsey stellt auch eine JavaScript Klasse `NodeBuilder` zur Verfügung, mit der sich SVG-Objekte erzeugen lassen. Wir gehen hier nicht darauf ein.

6.2 Elementare Textelemente

Analog zu dem kurzen Abriss oben seien hier einige der Textgestaltungsmöglichkeiten beschrieben.

Vorsicht: Die x-Koordinaten unten sind korrekt, Lindsey gibt auf der Web-Seite fälschlich `x="0"` an, hat im SVG-Quelltext aber die richtigen Werte (wie ich nach einer halben Stunde herausfand – grummel).

	`<text x="0" y="13" fill="red"` `text-anchor="start">Text</text>`
	`<text x="25" y="13" fill="red"` `text-anchor="middle">Text</text>`
	`<text x="50" y="13" fill="red"` `text-anchor="end">Text</text>`




```
<text x="0" y="13" fill="red">
  <tspan>Line 1</tspan>
  <tspan x="0" dy="1em">Line 2</tspan>
</text>
```


```
<defs>
  <path id="textPath"
        d="M10,50 C10,0 90,0 90,50"/>
</defs>
<use xlink:href="#textPath" stroke="blue"
     fill="none"/>
<text fill="red">
  <textPath xlink:href="#textPath">
    Text on a Path
  </textPath>
</text>
```

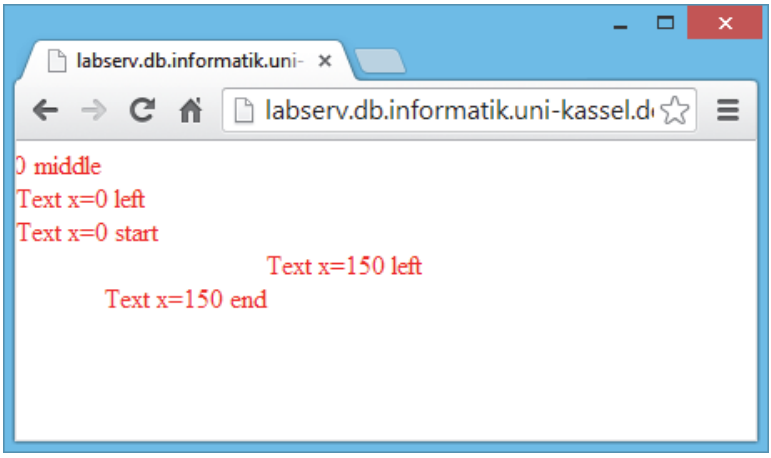
Grundsätzlich werden Texte als `text`-Elemente beschrieben. Das hat große Vorteile für die spätere Trennung von Inhalt und Form. SVG akzeptiert grundsätzlich Unicode-Codierungen.

Jeder Text hat eine *Startkoordinate*. In der Regel ist dies für die x-Achse die linke Kante des ersten Buchstabens, für die y-Achse die *Grundlinie der Schrift*. Die Einschränkung „in der Regel“ betrifft Texte mit Rotation oder umgekehrter Schriftrichtung. Die Beispiele oben sind im übrigen korrigiert gegenüber den originalen Seiten im Web (dort immer `x=0`).

Text kann eine Bündigkeit haben. Üblich sind „links“, „zentriert“ und „rechts“. Für SVG wird mit dem Attribut `text-anchor` und den Werten *start*, *middle*, *end* bestimmt, wie sich der Text zu einem durch (x, y)-Koordinaten gegebenen Punkt orientiert: bei *start* (oft kann man auch alternativ *left* verwenden) bestimmt der Punkt den Start des Textes, bei *middle* die Mittelausrichtung um diesen Punkt, bei *end* den Endpunkt des Textes, d. h. der Text schließt rechtsbündig an die gedachte x-Linie an.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<svg xmlns="http://www.w3.org/2000/svg">

  <text x="0" y="20" fill="red" text-anchor="middle">
    Text x=0 middle
  </text>
  <text x="0" y="40" fill="red" text-anchor="left">
    Text x=0 left
  </text>
  <text x="0" y="60" fill="red" text-anchor="start">
    Text x=0 start
  </text>
  <text x="150" y="80" fill="red" text-anchor="left">
    Text x=150 left
  </text>
  <text x="150" y="100" fill="red" text-anchor="end">
    Text x=150 end
  </text>

</svg>
```

Man beachte insbesondere, dass ein y-Wert von 0 den Text verschwinden lässt. Weitere Beispiele sind der Text oben mit dem Bild unten.



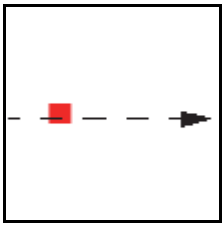
Das vierte Beispiel von Lindsey zeigt die Verwendung von `tspan`-Elementen, die nur innerhalb eines `text`-Elements auftreten dürfen. Man beachte hier auch die relative Positionierung der zweiten Zeile (`dy="..."`). `tspan`-Elemente erben die Eigenschaften der umgebenden

`text`-Elemente (außer Koordinaten), nicht aber die des vorherigen `tspan`-Elements.

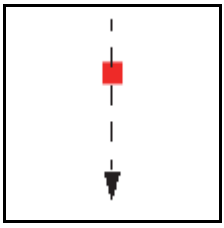
Zu beachten ist auch, dass SVG-Text anders als bei HTML nicht umgebrochen wird. Dies ist Aufgabe einer höheren Anwendungsebene, etwa eines DTP-Programms, das SVG erzeugt (macht Sinn, ist aber lästig).

Analog lässt sich ein Text, der mit `defs`-Elementen definiert wurde, per `tref`-Elemente wieder anzeigen, ggf. mit eigenen Stilattributen. Weitere Attribute sind setzbar, etwa die Schriftstärke (Salathé: Schriftdicke - horribile dictu): Zahlenwerte 100 bis 900 (nur die ganzen Hunderter) und die vordefinierten Werte `normal`, `bold`, `bolder` und `lighter` für das Attribut `font-weight`.

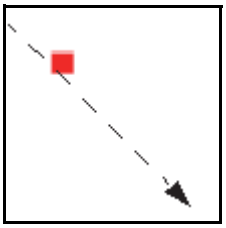
6.3 Animation



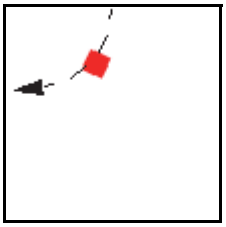
```
<rect y="45" width="10" height="10"
      fill="red">
  <animate attributeName="x" from="0"
           to="90" dur="10s"
           repeatCount="indefinite"/>
</rect>
```



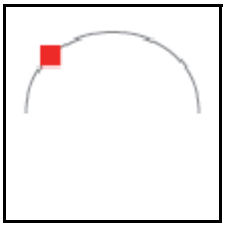
```
<rect x="45" width="10" height="10"
      fill="red">
  <animate attributeName="y" from="0"
           to="90" dur="10s"
           repeatCount="indefinite"/>
</rect>
```



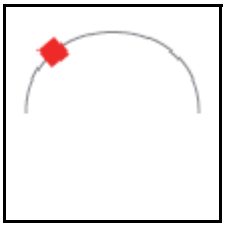
```
<rect width="10" height="10" fill="red">
  <animate attributeName="x" from="0"
           to="90" dur="10s"
           repeatCount="indefinite"/>
  <animate attributeName="y" from="0"
           to="90" dur="10s"
           repeatCount="indefinite"/>
</rect>
```



```
<rect x="45" width="10" height="10"
      fill="red">
  <animateTransform
     attributeName="transform"
     type="rotate" from="0" to="90"
     dur="10s"
     repeatCount="indefinite"/>
</rect>
```



```
<rect x="-5" y="-5" width="10"
      height="10" fill="red">
  <animateMotion
     path="M10,50 C10,0 90,0 90,50"
     dur="10s" repeatCount="indefinite"/>
</rect>
```



```
<rect x="-5" y="-5" width="10"
      height="10" fill="red">
  <animateMotion
     path="M10,50 C10,0 90,0 90,50"
     dur="10s" rotate="auto"
     repeatCount="indefinite"/>
</rect>
```

Unter Animation versteht man (in der Informatik ;-)) die zeitliche Veränderung von Bildern. Die Animationselemente von SVG beruhen auf den-

jenigen von SMIL (Synchronized Multimedia Integration Language) Animation.

Laut Salathé wurden die folgenden Elemente von SMIL übernommen:

- `<animate>`
- `<set>`
- `<animateMotion>`
- `<animateColor>`

Die SVG-spezifischen und SMIL-kompatiblen Elemente und Attribute sind:

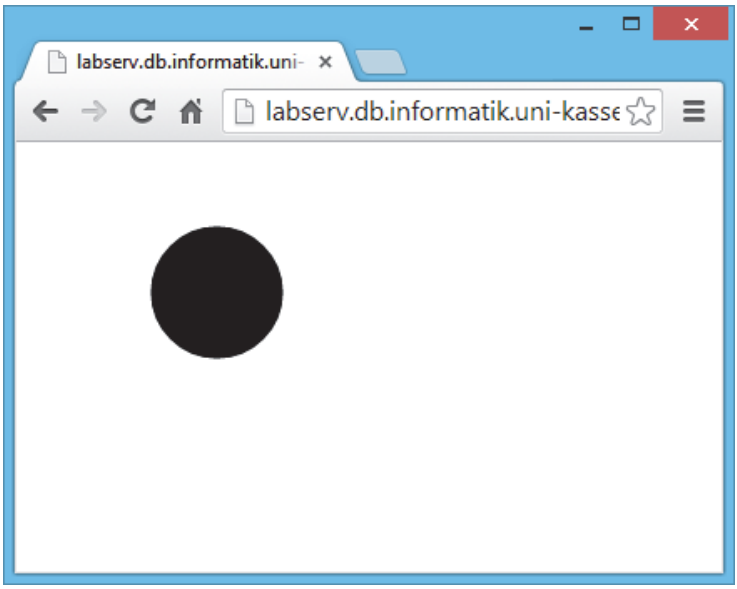
- `<animateTransform>`
- `<mpath>`
- `path` (Attribut des Elements `<animateMotion>`)
- `rotate` (Attribut des Elements `<animateMotion>`)

Man sieht an den Beispielen oben schön, wie Attribute von „normalen“ Elementen animiert werden. Im ersten Beispiel etwa die x-Koordinate, im zweiten Beispiel y, im dritten Beispiel x und y, im vierten Beispiel eine Rotation um 90 Grad, im fünften und sechsten Beispiel eine Animation entlang eines Pfades, einmal ohne Rotation des Würfels und einmal mit. Auf den Web-Seiten von Lindsey kann man dies schön verfolgen, die Wiederholung ist unbegrenzt (`repeatCount="indefinite"`).

Salathé gibt Beispiele an, bei denen sich etwa der Radius eines Kreises ändert, etwa innerhalb von 6s von 0 auf 80 wächst.

Damit der Kreis anschließend nicht verschwindet, fügt er ein `fill="freeze"` in das `<animate>`-Element ein. Dieses `fill` hat nichts mit dem Füllen von Figuren zu tun. Der Defaultwert für `fill` ist `"remove"` und das Animationsattribut wird auf den Ausgangswert zurückgesetzt, hier `r=0`, wodurch der Kreis verschwindet.

```
<?xml version="1.0"?>
<svg xmlns="http://www.w3.org/2000/svg"
     width="300" height="200">
  <circle cx="120" cy="90" r="0">
    <animate attributeName="r"
             begin="0s" dur="6s" fill="freeze"
             from="0" to="80"/>
  </circle>
</svg>
```



Der abgebildete Schnappschuss wurde ca. 2 bis 3 Sekunden nach dem Start aufgenommen.

Je Attribut oder Eigenschaft, die zu animieren ist, muss ein `<animate>`-Element angegeben werden. Dieses kann innerhalb der Objekts, hier `<circle>`, definiert sein, oder außerhalb mit `xlink:href="#Name"`.

```
<circle id="meinKreis" cx="120" cy="90" r="0"/>
<animate xlink:href="#meinKreis" attributeName="r"
         begin="0s" dur="6s" fill="freeze"
         from="0" to="80"/>
```

Das `animate`-Element hat eine große Anzahl von Attributen. Wir präsentieren hier nur die Liste der Attribute (ggf. mögliche Werte in Klammern).

- `xlink:href`
- `attributeName`
- `attributeType` (CSS, XML, auto)
- `begin` (Zeitangaben mit Minuten:Sekunden.Millisek.)
- `dur` (... oder Stunden:Minuten:Sekunden.Millisek.)
- `end` (`begin` und `end` können auch mit Events besetzt werden)
- `repeatCount` (Zahl größer 0 oder *indefinite*)
- `repeatDur` (Zeiteinheit oder *indefinite*)
- `fill`[1]
- `from`
- `to`
- `by`
- `values`
- `calcMode` (discrete, linear, paced, spline)
- `keyTimes`
- `keySplines`
- `additive` (replace, sum)
- `accumulate` (none, sum)

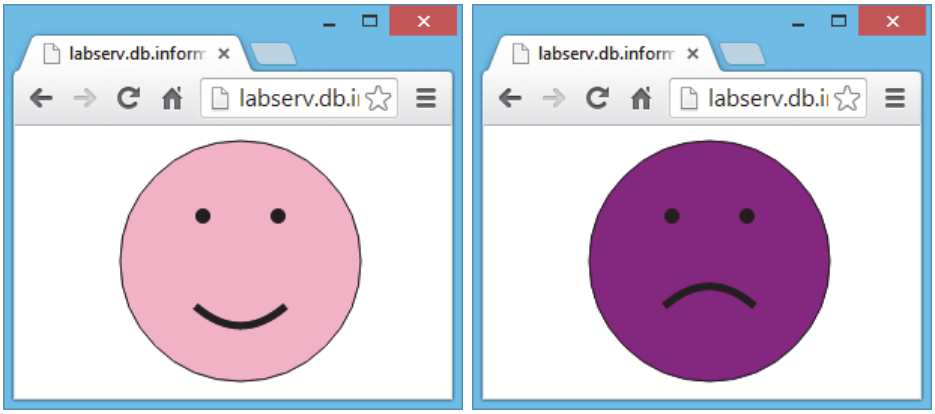
Es folgt eine weitere Reihe von Elementen:

- `<set>` setzen eines Attributs oder einer Eigenschaft während einer gewissen Zeit auf einen gewissen Wert; Attribute wie bei `<animate>`.
- `<animateMotion>` Animation eines Objekts entlang eines Pfades; kann mit Attribute `rotate` versehen werden, üblicher Wert dann *auto*.
- `<animateColor>`

Die folgende Animation („Choleriker à la Wegner“) zeigt eine Kombination der oben genannten Animationseffekte. Die Produktion (inkl. Idee) benötigte ca. eine Stunde (trial & error).

Sowohl für die Farbe als auch die Variation der quadratischen Bézierkurve wurde mit `values="..."` gearbeitet, wobei drei Werte eingegeben wurden, davon Anfangs- und Endwert gleich. Andere Realisierungen sind sicher möglich.

1. Unglückliche Wortwahl, warum nicht z. B. `keep="..."`.



```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20010904//EN"
     "http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink">
  <circle r="80" cx="150" cy="90"
          stroke="black" fill="pink">
    <animateColor attributeName="fill"
                  values="pink;purple;pink"
                  dur="6s" repeatCount="indefinite"/>
  </circle>
  <defs>
    <circle id="auge" r="5" fill="black"/>
  </defs>
  <use x="125" y="60" xlink:href="#auge"/>
  <use x="175" y="60" xlink:href="#auge"/>
  <path d="M120,120 q30,30 60,0"
        stroke="black" stroke-width="5" fill="none">
    <animate attributeName="d"
             values="M120,120 q30,30  60,0;
                     M120,120 q30,-30 60,0;
                     M120,120 q30,30  60,0"
             dur="6s" repeatCount="indefinite"/>
  </path>
</svg>
```

6.4 Events

6.4.1 Mausereignisse

Lindsey hat eine schöne Liste von Beispielen, die man aber nur in der Aktion anschauen kann. Es wird jeweils ein Rechteck von rot auf blau umgefärbt bei Eintritt des Ereignisses.

```
<rect x="5" y="5" width="40" height="40" fill="red">
 <set attributeName="fill" to="blue" begin="click"/>
</rect>

<rect x="5" y="5" width="40" height="40" fill="red">
 <set attributeName="fill" to="blue" begin="mousedown"/>
</rect>

<rect x="5" y="5" width="40" height="40" fill="red">
 <set attributeName="fill" to="blue" begin="mouseup"/>
</rect>

<rect x="5" y="5" width="40" height="40" fill="red">
 <set attributeName="fill" to="blue" begin="mouseover"/>
</rect>

<rect x="5" y="5" width="40" height="40" fill="red">
 <set attributeName="fill" to="blue" begin="mouseout"/>
</rect>

<rect x="5" y="5" width="40" height="40" fill="red">
 <set attributeName="fill" to="blue" begin="mousemove"/>
</rect>
```

Das Attribut `begin` kann sich auch auf das Ereignis eines anderen Objekts beziehen. Möchte man zum Beispiel erreichen, dass sich das Rechteck nach dem Klick auf einen Knopf mit der ID `button` umfärbt, so würde man `begin="button.click"` angeben.

6.4.2 Tastaturereignisse

Lindsey gibt Beispiele für Tastaturereignisse, wobei mit jedem Tastendruck z. B. ein Zähler hochgezählt wird. Lindsey verwendet dazu das Attribut `onkeydown` für Tastaturereignisse, das zwar vom Adobe SVG-

Plug-In unterstützt wird, jedoch kein Bestandteil des SVG-Standards ist. Unten wird eine alternative Lösung gezeigt, die mit der Methode `addEventListener()` arbeitet.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20010904//EN"
     "http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">

<svg version="1.0" xmlns="http://www.w3.org/2000/svg"
     onload="init(evt)">

  <script type="application/ecmascript">
    <![CDATA[
      var count = 0;
      function init(evt) {
        svgDoc = evt.target.ownerDocument;
        svgDoc.addEventListener("keypress", press, false);
      }
      function press(evt) {
        svgDoc = evt.target.ownerDocument;
        var text = svgDoc.getElementById("text").firstChild;
        text.data = ++count;
      }
    ]]>
  </script>

  <rect width="100%" height="100%" fill="blue"/>
  <text id="text" x="25" y="27" fill="yellow">0</text>

</svg>
```

6.4.3 Dokumentereignisse

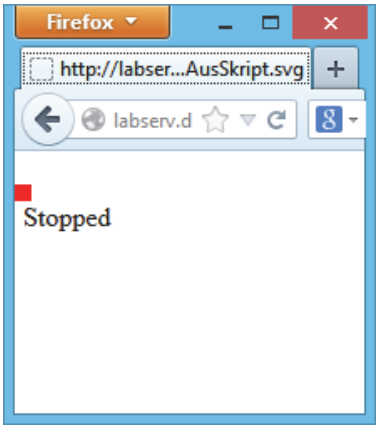
Lindsey gibt Beispiele an, wobei ich nur in zwei Fällen Ereignisse auslösen konnte (z. B. bei Resize), wobei dann ein Warnbutton kommt und anschließend sich das Fenster in der Größe ändert.

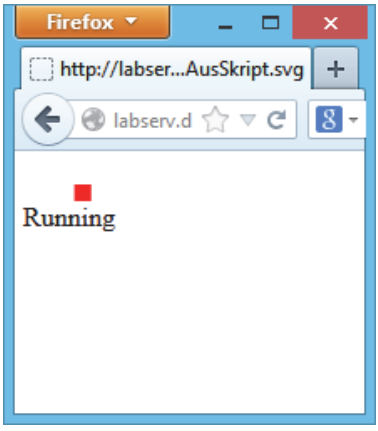
- `<svg onabort="notify(evt)" ...>`
- `<svg onerror="notify(evt)" ...>`
- `<svg onunload="notify(evt)" ...>`
- `<svg onscroll="notify(evt)" ...>`
- `<svg onzoom="notify(evt)" ...>`
- `<svg onresize="notify(evt)" ...>`

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20010904//EN"
     "http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">
<svg version="1.0" xmlns="http://www.w3.org/2000/svg"
      onresize="notify(evt)">
  <script type="application/ecmascript">
    <![CDATA[
     function notify(evt) {
      alert("Resize!");
     }
    ]]>
  </script>
  <rect width="100%" height="100%" fill="blue"/>
</svg>
```

6.4.4 Animationsereignisse

Hier wird im Beispiel bei Lindsey das rote Rechteck in Bewegung gesetzt und es erscheint der Text „Running“. Anschließend hält das Rechteck an und es erscheint der Text „Stopped“. Durch Klicken in das Rechteck wird die Animation erneut gestartet.





```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20010904//EN"
     "http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">
<svg version="1.0" xmlns="http://www.w3.org/2000/svg">
  <script type="application/ecmascript">
    <![CDATA[
     function advance(evt, message) {
       svgDocument = evt.target.ownerDocument;
       var text =
           svgDocument.getElementById("text").firstChild;
       text.data = message;
     }
    ]]>
  </script>
  <rect y="20" width="10" height="10" fill="red">
    <animate attributeName="x" values="0;40"
             begin="click" dur="2s"
             onbegin="advance(evt, 'Running')"
             onend="advance(evt, 'Stopped')"/>
  </rect>
  <text id="text" x="5" y="45"
        style="text-anchor: start">
    Stopped
  </text>
</svg>
```

6.5 Einbindung von Audiodateien

Lindsey enthält noch viele andere nützliche Beispiele, etwa zum Verfolgen der Mauskoordinaten und für Transformationen. Hier nur kurz ein SVG-Beispiel für das Adobe Plug-In. Bei Anklicken des roten Kreises ertönt der „Gong“.

```
<svg xmlns:a="http://www.adobe.com/svg10-extensions"
     a:timeline="independent">
  <a:audio xlink:href="ding.wav" begin="play.click"/>
  <circle id="play" cx="25" cy="25" r="10" fill="red"/>
</svg>
```

Der SVG-Standard unterstützt im Gegensatz zum Adobe Plug-In keine Audio- und Videodateien. Die für mobile Geräte vorgesehene Variante SVG Tiny 1.2 kennt `<audio>`- und `<video>`-Elemente. Als Alternative

lassen sich die HTML5-Funktionalitäten zum Abspielen von Audio und Video nutzen, indem man den SVG-Code in ein HTML-Dokument einbettet.

6.6 Schlussbemerkung

Es gab auch Bestrebungen, SVG als Grundlage für einen Bildschirmmanager zu nehmen („Ximian“). Damit zeichnen sich auch Parallelen zu anderen graphischen Sprachen und daraus abgeleiteten Widget-Managern ab, speziell erinnert mich die Entwicklung stark an Tcl/Tk. Im Fall von SVG ist die Syntax durch XML offener und mit JavaScript hat man eine weitverbreitete Sprache für die aktiven Elemente.

7 XQuery

XQuery ist eine Anfragesprache an XML-Datenbestände, die seit Januar 2007 in der Version 1.0 als W3C-Norm vorliegt [25, 47]. Damit verbunden war eine Überarbeitung von XPath: XPath 2.0 ist jetzt eine syntaktische Untermenge von XQuery 1.0. Warum braucht man eine solche Sprache?

XML-Dokumente sind lesbare Formen der Datendarstellung und haben große Ähnlichkeit mit hierarchisch strukturierten Tabellen eines objekt-relationalen DBMS. Dominiert die dokumenten-zentrische Sicht, dann sind laut [31] die Daten (Dokumente) selten völlig gleich strukturiert, die Reihenfolge der Einträge ist wichtig, es gibt sinngebende Daten auf allen Ebenen (mixed content), eine Volltextsuche ist unabdingbar, aber nicht ausreichend. Beispiele sind:

- Zeitschriftenbeiträge, Bücher
- Gebrauchsanweisungen, Handbücher
- E-Mail
- Präsentationen
- Verträge

Dominiert die daten-zentrische Sicht, dann sind XML-Dokumente Daten im herkömmlichen (Datenbank-) Sinn, die Reihenfolge ist oft nicht relevant (Schlüssel!), die Daten sind einheitlich und meist einfach strukturiert, haben Datentypen, sinntragende Daten sind in Blattelementen oder Attributen, gemischter Inhalt ist selten oder Dekoration. Beispiele sind:

- Telefonbücher

- wissenschaftliche Daten
- Fahrpläne, Flugpläne
- Bestellungen

XML unterstützt gut beide Sichten, man denke an Krankenakten mit einfach strukturierten Daten (Geburtsdatum, Adresse, Behandlungstage und Abrechnungsziffern), mit binären Daten (Röntgenbilder) und wenig strukturierten Dokumenten (Diagnose, Anamnese, ...). XML bietet sich daher sowohl für den Datenaustausch als auch für die Datenspeicherung an.

In letzterem Fall könnte man die Speicherung in einem modifizierten Datenbanksystem vorsehen, dass (mehr oder minder stark gewandelte) Dokumente oder Dokumentdaten aufnimmt. Hierzu wird derzeit mit SQL/XML ein Standard vorbereitet (vgl. den Artikel von Eisenberg und Melton [32]), der allerdings auch die Abfrage von XML-Dokumenten mittels der Abfragesprache XQuery, die in SQL eingebettet wird, zulässt. In diesem Szenario lassen sich Stärken eines DBMS, etwa die Transaktionsverarbeitung und Datensicherungsmechanismen, ausnutzen.

Alternativ kann man sich eine reine XML-Speicherung vorstellen. Dann arbeitet XQuery direkt auf den Dokumenten, sollte aber wie SQL deskriptiv formulierte Anfragen ermöglichen. Dies ist ganz ähnlich zur Arbeitsweise der *Templates* in XSLT, wenn man die prozeduralen Komponenten (for-each, if-then-else-fi) ignoriert. Wesentlich wird wieder die Verwendung von XPath sein, um Baumknoten zu adressieren. Das Gegenstück zum Select-from-where (SFW) Paradigma in SQL ist der FLWOR-Ausdruck[1] (for, let, where, order-by, return), der auch die Neustrukturierung des Ausgaberesultats regelt. Dies behandeln wir weiter unten.

Für diese Form der direkten Speicherung sind in den letzten drei Jahren sog. native XML-Datenbanken entwickelt worden (Natix, Tamino, Xendice), die für manche Anwendungszwecke eine durchaus sinnvolle Alternative darstellen.

---

1. Ausgesprochen wie das englische „*flower*“.

Die Behandlung von XQuery folgt dem Artikel [29] und dem Buch [30], beide von Lehner und Schöning, sowie dem übersichtlichen Primer unter `http://www.softwareag.com/xml/tools/xquery_primer.pdf` und der frei verfügbaren Einführung unter `http://www.datadirect.com/techzone/xml/xquery/docs/Katz_C01.pdf`, bzw. *XQuery: A Guided Tour* von Jonathan Robie, verfügbar unter `http://www.datadirect.com/developer/xquery/xquerybook/index.ssp`. Eine sehr ausführliche, aber auch etwas langatmige Übersicht bietet das 2006 erschienene Buch von Melton und Buxton [48].

Zur Übung stehen verschiedene XQuery-Implementierungen zur Verfügung, wie z. B. Galax[1] oder IPSI-XQ[2] der Fraunhofer Gesellschaft.

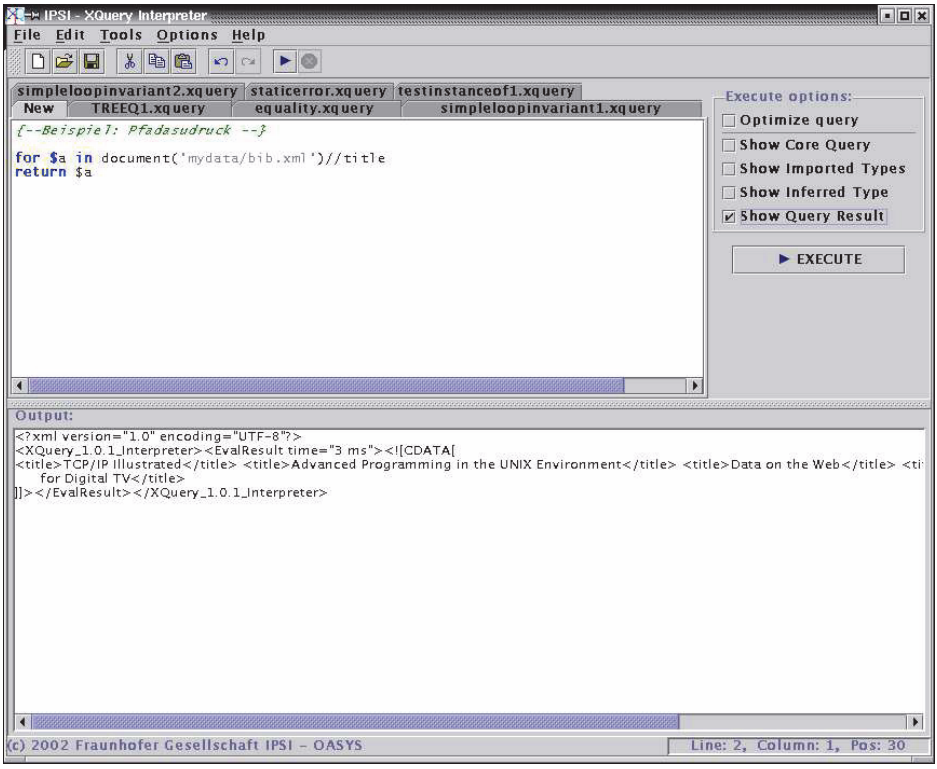


Abb. 7–1 *Ausführung einer XQuery-Abfrage mit IPSI-XQ*

1. http://www.galaxquery.org/
2. http://www.ipsi.fraunhofer.de/oasys/projects/ipsi-xq/index_d.html

7.1 Ausdrücke

7.1.1 Elementare Ausdrücke

Wie in Programmiersprachen unterscheidet man einfache und komplexe Ausdrücke. Zu letzteren gehören die FLWOR-Ausdrücke, Pfadausdrücke, arithmetische, konditionale, quantifizierende und Vergleichsausdrücke.

Zu den elementaren Ausdrücken gehören die Variablenreferenzen, wobei `$` einem gültigen XQuery-Bezeichner vorangestellt wird.

```
$name := "Morad Ahmad"
```

Rechts steht ein Zeichenkettenliteral, analog lassen sich Zahlen darstellen, die ggf. noch mittels Konstruktoren in Werte eines numerischen Typs gewandelt werden:

```
xs:float(1.34e4)
```

Weitere elementare Ausdrücke sind Funktionsaufrufe, etwa eine Funktion `compare()` aus dem Namensraum `fn` (`http://www.w3.org/2005/xpath-functions`).

```
fn:compare("Patt-Idiotismus", "Patriotismus")
```

7.1.2 Kommentare

XQuery verwendet die Zeichen `(:` und `:)` für potentiell geschachtelte Kommentare, die ebenfalls elementare Ausdrücke sind.

```
(: Dies ist ein geschachtelter (: XQuery :) Kommentar :)
```

XQuery-Kommentare erzeugen keine XML-Kommentare und sind von diesen wohlunterschieden. Dafür existieren in XQuery *Kommentar-Konstruktoren*.

7.1.3 Das XQuery-Datenmodell

Das **Datenmodell** von XQuery (interne Darstellung eines XML-Dokuments) ist identisch mit dem von XPath (siehe Kapitel 5).

Zentraler Konstrukt ist die Sequenz mit einer beliebigen Anzahl von Einträgen (atomare Werte oder Knoten), die eine Position haben. Sequenzen sind demnach geordnet, möglich ist die leere Sequenz `()`. Einelementige Sequenzen und ein einzelnes Element sind gleichwertig. Zur Analyse und Modifikation gibt es eine Reihe von Funktionen und Operatoren, z. B. [29]:

- Kommaoperator
  `(1, <X/>), (3)` liefert
  `(1, <X/>, 3)`
- `to`-Operator
  `2 to 5` liefert `(2, 3, 4, 5)`
- Eliminierung von Duplikaten
  `fn:distinct-values ((2,5,3,7,3,5,5))` liefert
  `(2,5,3,7)` wobei die Reihenfolge implementationsabhängig ist.

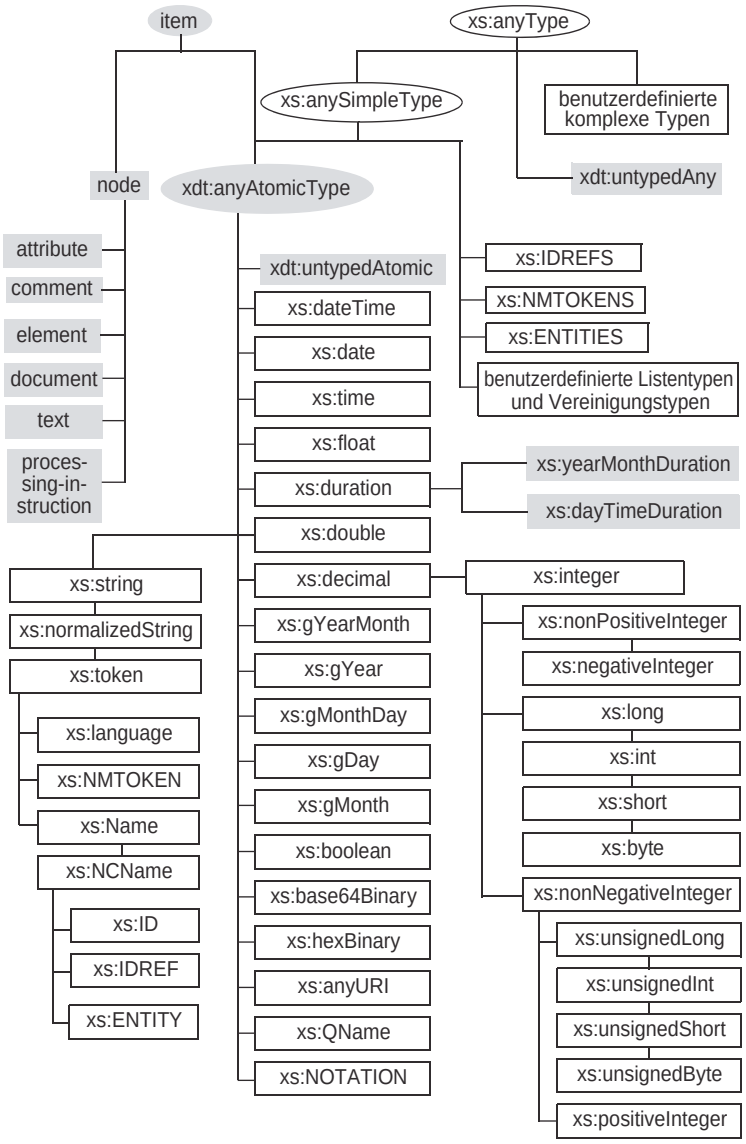
XQuery stützt sich auf die in XML Schema vordefinierten Datentypen (Namensraumpräfix meist `xs` oder `xsd`), erweitert diese aber um einige unspezifische Datentypen. Die Abbildung unten (aus [30]) zeigt die Typhierarchie von XQuery.

Die allgemeinste Form eines Eintrags in einer XQuery-Sequenz ist vom Typ `item()`, die gesamte Sequenz dann vom Typ `item()*`. Die Überführung eines XML-Dokuments in eine Instanz des XML-Infoset und von dort in eine Instanz des XQuery-Modells erfolgt durch die Eingabefunktionen `fn:doc()` oder `fn:collection()` und bleibt verborgen.

7.1.4 Einfache Datentypen

XQuery unterstützt drei numerische Datentypen: `integer`, `decimal` und `double`, welche den Spezifikationen in XML Schema entsprechen. Für Zahlen gelten die folgenden Regeln:

- Jede Zahl kann ein Vorzeichen haben.
- Eine Zahl nur aus Ziffern ist vom Typ `integer`.
- Eine Zahl nur aus Ziffern und einem einzelnen Punkt ist vom Typ `decimal`.



- Jede gültige Zahl, die eine `e`- oder `E`-Angabe enthält, ist vom Typ `double`.

Beispiele:

```
1          (: An integer :)
-2         (: An integer :)
```

```
+2          (: An integer :)
1.23        (: A decimal :)
-1.23       (: A decimal :)
1.2e5       (: A double :)
-1.2E5      (: A double :)
```

Strings werden wie in XML in einfachen oder doppelten Anführungszeichen angegeben. Beispiel:

```
"a string"
'a string'
"This is a string, isn't it?"
'This is a "string" '
```

7.1.5 Lokalisierungspfade

Zur Adressierung von Knoten dienen Lokalisierungspfade. Diese sind die gleichen wie in XPath. **XPath 2.0** [26] verwendet sogar die gleichen Pfadausdrücke wie XQuery (Erweiterungen von XQuery eingebaut).

Lehner und Schöning schreiben [29]: „Ein XPath-Ausdruck besteht aus einer, durch das /-Zeichen getrennten Folge von Lokalisierungsschritten, die von links nach rechts ausgewertet werden. In jedem Schritt wird ausgehend von den Ergebnisknoten des vorangehenden Lokalisationsschrittes eine neue Menge von Knoten ermittelt. Initial erfolgt die Auswertung ausgehend von dem so genannten Kontextknoten, der im dynamischen Kontext geführt wird."

Der initiale Kontext wird bei XQuery durch die oben bereits erwähnten Eingabefunktionen `fn:doc()` und `fn:collection()` hergestellt. So liefert

```
fn:doc("patienten.xml")//Adresse/Wohnort
```

eine Sequenz von allen Wohnortknoten (einschließlich Duplikate) aller registrierten Patienten.

Ein Lokalisierungsschritt besteht wiederum aus drei Komponenten (Navigationsachse, Knotentest und optional einer Liste von Prädikaten), die eine gezielte Identifikation von Knoten in einem Dokument ermöglichen, d. h.:

```
Achse::Knotentest[Prädikat1]...[PrädikatN]
```

7.1.6 Navigationsachsen

XQuery unterstützt mit Ausnahme der Namensraumachse alle aus XPath bekannten Navigationsachsen. Eine Navigationsachse gibt dabei ausgehend von dem aktuellen Knoten den Teil des XML-Dokuments an, in dem nach Knoten im Rahmen der Auswertung des aktuellen Lokalisierungsschritts gesucht werden soll. Die Abbildung 7–2 unten zeigt visuell die Suchräume für die unterschiedlichen Achsen.

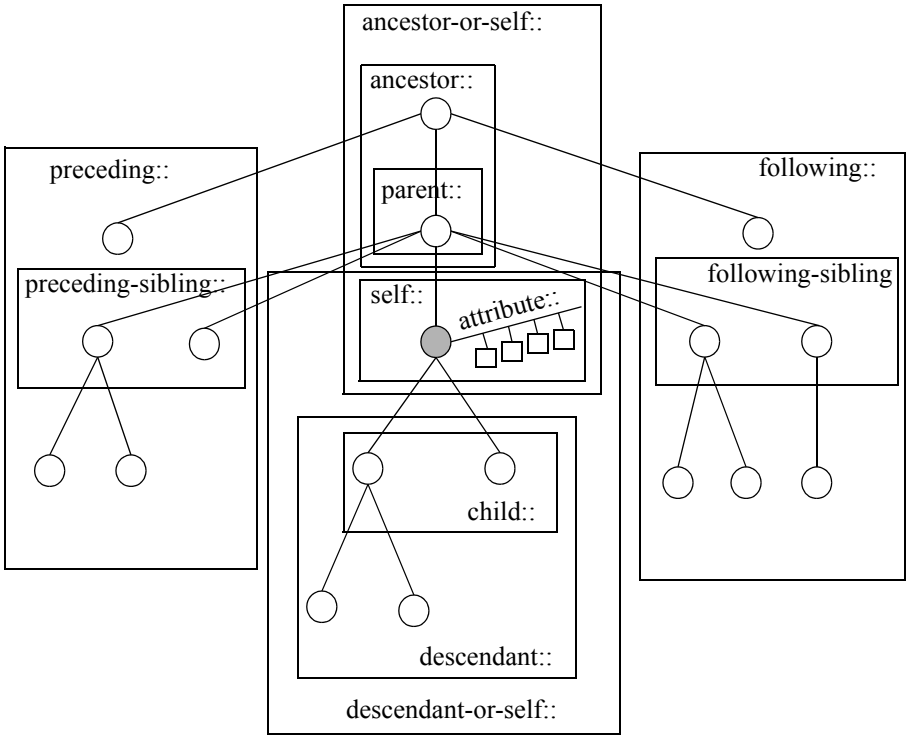


Abb. 7–2 *Navigationsachsen in XQuery (nach [29])*

Explizit zu erwähnen ist an dieser Stelle die Achse `child::`, die alle direkten Kinder des Kontextknotens liefert und als Standardachse gilt, so dass die Angabe optional ist. Folgende Ausdrücke sind somit äquivalent:

```
/child::Patient/child::Name
/Patient/Name
```

Darüber hinaus sind drei häufig benutzte Abkürzungen zu erwähnen: Die `attribute::`-Achse (alle Attribute des Kontextknotens) kann durch das `@`-Symbol, die `self::`-Achse durch einen einfachen und die `parent::`-Achse durch einen zweifachen Punkt (`..`) ersetzt werden. Somit sind folgende Ausdrücke wieder äquivalent:

```
/child::Patient/self::*/child::Name/
                    parent::*/attribute::PatientNr
/Patient/./Name/../@PatientNr
```

Eine Trennung von zwei Lokalisierungsschritten durch `//`-Zeichen wird ebenfalls als Abkürzung geführt und durch einen Ausdruck mit Bezug auf die `descendant-or-self::`-Achse während der Auswertung ersetzt. Die beiden Ausdrücke

```
Patient//Operation
Patient/descendant-or-self::node()/Operation
```

Ein `/`-Zeichen zu Beginn eines XPath-Ausdrucks impliziert, dass in der Wurzel des Dokuments, in dem sich der aktuelle Kontextknoten befindet, mit der Auswertung des ersten Lokalisierungsschritts begonnen werden soll; implizit erfolgt dabei wiederum eine Expansion zu folgendem Ausdruck:

```
fn:root(self::node())
```

Ein doppelter Schrägstrich zu Beginn eines Dokuments erweitert den absoluten Pfadausdruck um die Suche im gesamten Dokument und wird ersetzt durch folgenden Ausdruck:

```
fn:root(self::node())/descendant-or-self::node()
```

7.1.7 Knotentest und Knotentypen

Der zweite Bestandteil eines XPath-Ausdrucks ermöglicht, die Suche auf Knoten eines bestimmten Typs einzuschränken. Ein Knotentest kann sich dabei entweder auf den Namen eines Knotens oder auf den Knotentyp beziehen. Eine Wildcard (`*`) wird benutzt, falls kein Knotentest erforderlich ist. Die folgende Tabelle (nach [29]) zeigt einige Lokalisierungsschritte mit Knotentest und den entsprechenden Erklärungen.

`child::Name`	Kinderknoten mit Bezeichnung `Name`
`child::*` `*`	alle Kinderelemente
`attribute::*` `@*`	alle Attribute des aktuellen Knotens
`child::element(Name)`	Kinderelemente mit Bezeichnung `Name`
`child::element(*,` `  Angestellte_T)`	Kindknoten vom Typ `Angestellte_T`
`attribute::attribute(*,` `  xs:integer)`	Attribute mit ganzzahligen Werten

`attribute::attribute(Alter)`
`attribute::Alter`
`@Alter`
	Attribut mit Namen `Alter`

Ein Test bezüglich des Typs eines Knotens bezieht sich wiederum auf Konstrukte aus dem XQuery-Datenmodell, die in Abb. 7–3 dargestellt sind. Eine Sequenz ist entweder leer (`empty-sequence()`) oder vom Typ `item()*`. Ein einzelner Eintrag (`item()`) hat entweder einen atomaren oder einen Knotentyp (`node()`). Auf bestimmte Knotenarten kann geprüft werden. Detaillierter kann dabei auf eine Kombination aus Knotenbe-

zeichnung und dem dahinter liegenden Typ getestet werden. Der Ausdruck

```
element(Vorname, xs:string)
```

liefert alle Elementknoten mit Bezeichnung `Vorname` mit Inhalt des Knotens vom Datentyp `xs:string`. Ein Test bezüglich der Knoteneigenschaft (`node()`) und damit Ausschluss atomarer Datentypen wird beispielsweise auch im vorangegangenen Abschnitt bei der Auflösung der Zeichen `/` bzw. `//` verwendet.

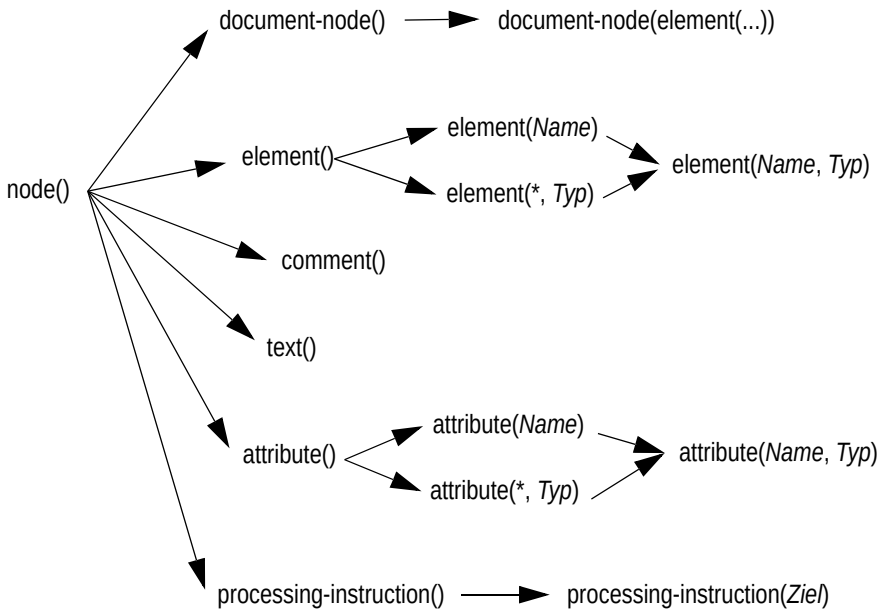


Abb. 7–3 *Testmöglichkeiten auf Knotentypen (nach [29])*

7.1.8 Prädikate

Eingebettet in ein `[]`-Paar kann jeweils ein Prädikat zur weiteren Filterung von Knoten in einem Lokalisierungsschritt auftreten. So liefert folgender Lokalisierungsschritt nur Chirurgen:

```
descendant::Arzt[@rolle="Chirurg"]
```

Das Prädikat kann entweder ein beliebiger XQuery-Ausdruck sein, dessen Wahrheitswert durch die Funktion `fn:boolean()` zur Laufzeit bestimmt wird, oder die Angabe der gewünschten Kontextposition. Die dritte Operation wird wie folgt bestimmt

```
child::Operation[3]
```

Allgemeiner kann ein Positionstest durch Rückgriff auf die Funktion `fn:position()` erfolgen, zum Beispiel:

```
child::Operation[fn:position() = (3 to 5)]
```

Eine Liste von Prädikaten in einem Lokalisierungsschritt wird schrittweise abgearbeitet. Die beiden folgenden Ausdrücke sind somit *nicht* äquivalent:

```
child::Operation[Transplantation][fn:position() = 3]
child::Operation[fn:position() = 3][Transplantation]
```

Die Kombination aus Achsen und Tests über Knotentypen bzw. allgemeine Prädikate bildet ein sehr mächtiges Mittel, um gezielt Elemente in einem XML-Datenbestand lokalisieren zu können.

7.2 FLWOR-Ausdrücke

Neben den oben beschriebenen Pfadausdrücken bilden die FLWOR-Ausdrücke das Zentralstück von XQuery zur Formulierung von Anfragen. Der Ausdruck bindet Variablen an Ergebnisse von Ausdrücken, wertet darauf in einer `where`-Klausel ein Prädikat aus und konstruiert ein neues Dokument als Resultat.

In dieser Einführung verwenden wir das folgende XML-Dokument `bib.xml`:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE bib SYSTEM "bib.dtd">
<bib>
  <book year="1994">
    <title>TCP/IP Illustrated</title>
    <author><last>Stevens</last><first>W.</first></author>
    <publisher>Addison-Wesley</publisher>
    <price>65.95</price>
  </book>
```

```
  <book year="1992">
    <title>
      Advanced Programming in the UNIX Environment
    </title>
    <author><last>Stevens</last><first>W.</first></author>
    <publisher>Addison-Wesley</publisher>
    <price>65.95</price>
  </book>
  <book year="2000">
    <title>Data on the Web</title>
    <author><last>Abiteboul</last><first>Serge</first></author>
    <author><last>Buneman</last><first>Peter</first></author>
    <author><last>Suciu</last><first>Dan</first></author>
    <publisher>Morgan Kaufmann Publishers</publisher>
    <price>55.95</price>
  </book>
  <book year="1999">
    <title>
      The Economics of Technology and Content for Digital TV
    </title>
    <editor>
      <last>Gerbarg</last><first>Darcy</first>
      <affiliation>CITI</affiliation>
    </editor>
    <publisher>Kluwer Academic Publishers</publisher>
    <price>129.95</price>
  </book>
</bib>
```

Die zugehörige DTD `bib.dtd`:

```
<!ELEMENT bib (book*)>
<!ELEMENT book (title, (author+ | editor+),
                        publisher, price)>
<!ATTLIST book year CDATA #REQUIRED>
<!ELEMENT author (last, first)>
<!ELEMENT editor (last, first, affiliation)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT last (#PCDATA)>
<!ELEMENT first (#PCDATA)>
<!ELEMENT affiliation (#PCDATA)>
<!ELEMENT publisher (#PCDATA)>
<!ELEMENT price (#PCDATA)>
```

Die folgende Abfrage nach dem Buchtitel der im Jahr 2000 publizierten Bücher

```
for $b in fn:doc("bib.xml")//book
```

```
where $b/@year = "2000"
return $b/title
```

liefert dann das Ergebnis:

```
<title>Data on the Web</title>
```

Die Bezeichnung FLWOR (gesprochen „flower“) ist ein Akronym für die ersten Buchstaben der Klauseln, aus denen ein FLWOR-Ausdruck besteht. Die Analogie zum *select-from-where* in SQL ist offensichtlich, allerdings ist SQL eine Realisierung einer Relationenalgebra, während XQuery eine Art tupel-relationales Kalkül implementiert. Ein eher trivialer Unterschied liegt darin, dass XQuery auf Groß- und Kleinschreibung achtet.

- **`for`** definiert eine Variable, die eine Liste von Elementen durchläuft (eine Variable, die **einzeln** an jedes Element gebunden wird).
- **`let`** definiert eine Variable, deren Wert **eine ganze Liste** von Elementen ist (eine Variable, die an die **ganze Sequenz** gebunden wird).
- **`where`** filtert ein Ergebnis, so dass nur diejenigen Elemente verbleiben, die eine Bedingung erfüllen.
- **`order by`** sortiert das Ergebnis.
- **`return`** liefert eine Sequenz von Einträgen als Ergebnis des FLWOR-Ausdrucks zurück, die gemäß der angegebenen Elementkonstruktion erzeugt werden.

Dabei wird ein FLWOR-Ausdruck nur selten genau nach dieser Reihenfolge aufgebaut. Er beginnt mit mindestens einer `let`- oder `for`-Klausel, ggf. mehrere gemischt in beliebiger Reihenfolge, gefolgt von einer optionalen `where`-Klausel, gefolgt von einer optionalen `order by`-Klausel, gefolgt von einer notwendigen `return`-Klausel.

Die Grundvorstellung ist nun, dass FLWOR-Ausdrücke einen Strom von Tupeln erzeugen. Jedes Tupel besteht aus einem oder mehreren (*Variable*, *Wert*)-Paaren: Man sagt, ein Wert wird an die Variable gebunden. Dieser Strom durchläuft dann den `where`-Filter, wird ggf. neu geordnet und erzeugt mittels `return` eine Ausgabesequenz, die in der Regel nicht wieder ein vollständiges XML-Dokument ist.

7.2.1 for- und let-Klausel

Der Unterschied zwischen `for` und `let` ist, dass der Wert einer Variable bei `let` eine gesamte Liste ist, während bei `for` die Listenelemente durchlaufen werden. Zunächst ein ganz einfaches Beispiel mit einer `for`-Klausel.

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

Ergebnis:

```
<tuple><i>1</i></tuple>,
<tuple><i>2</i></tuple>,
<tuple><i>3</i></tuple>
```

In der Regel wird die Variable mittels Pfadausdruck gebunden, so wie im Buch-Beispiel oben. Gibt es eine Ordnung für die selektierten Werte, an die die Variable gebunden wird, dann ist auch die Ausgabe so geordnet[1].

Durch zwei geschachtelte `for`-Klauseln kann eine Art Kreuzprodukt gebildet werden, wie man es vom *join* in SQL her kennt.

```
for $i in (1, 2, 3),
    $j in (4, 5, 6)
return
  <tuple><i>{$i}</i><j>{$j}</j></tuple>
```

Das Resultat lautet

```
<tuple><i>1</i><j>4</j></tuple>,
<tuple><i>1</i><j>5</j></tuple>,
<tuple><i>1</i><j>6</j></tuple>,
<tuple><i>2</i><j>4</j></tuple>,
<tuple><i>2</i><j>5</j></tuple>,
<tuple><i>2</i><j>6</j></tuple>,
<tuple><i>3</i><j>4</j></tuple>,
<tuple><i>3</i><j>5</j></tuple>,
<tuple><i>3</i><j>6</j></tuple>
```

Hinweis: Auch die Schreibweise

1. Die einzig vorstellbare Ausnahme wären Angaben der Art .../@*, also Attributwerte, da Attribute in einem XML-Dokument keine Ordnung haben.

```
for $i in (1, 2, 3)
for $j in (4, 5, 6)
return
   ...
```

wäre erlaubt. Ebenso einfach eine Abfrage mit einer `let`-Klausel:

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

Hier wird *i* an die Sequenz als ganzes gebunden und liefert damit

```
<tuple><i>1 2 3</i></tuple>
```

Interessanter ist die Kombination von `let`-Klausel mit `for`.

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

Ergebnis:

```
<tuple><i>1</i><j>1 2 3</j></tuple>,
<tuple><i>2</i><j>1 2 3</j></tuple>,
<tuple><i>3</i><j>1 2 3</j></tuple>
```

Der Wert einer „`let`-Variablen“ wird bei jedem Schleifendurchlauf neu berechnet, auch wenn man es hier nicht sieht. Das folgende Beispiel

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

zeigt es deutlicher. Das Ergebnis der XQuery ist

```
<tuple><i>1</i><j>1 2</j></tuple>,
<tuple><i>2</i><j>2 3</j></tuple>,
<tuple><i>3</i><j>3 4</j></tuple>
```

Ein etwas praktischeres Beispiel, das den Buchtitel und die Anzahl der Autoren ausgibt, wäre

```
for $b in fn:doc("bib.xml")//book
let $c := $b/author
return
  <book>{ $b/title, <count>{ count($c) }</count>}</book>
```

Das Ergebnis lautet:

```
<book>
  <title>TCP/IP Illustrated</title>
  <count>1</count>
</book>,
<book>
  <title>Advanced Programming in the UNIX
         Environment</title>
  <count>1</count>
</book>,
<book>
  <title>Data on the Web</title>
  <count>3</count>
</book>,
<book>
  <title>The Economics of Technology and Content for
         Digital TV</title>
  <count>0</count>
</book>
```

Formuliert man die Query übrigens wie folgt

```
for $b in fn:doc("bib.xml")//book
let $c := fn:doc("bib.xml")//author
return
  <book>{ $b/title, <count>{fn:count($c)}</count> }</book>
```

hat jedes der vier Bücher genau fünf Autoren!

Etwas Vorsicht ist geboten mit dem Begriff „Variable“. Wer in der folgenden Abfrage erwartet, dass *j* von 1 bis 3 hochgezählt wird, sieht sich getäuscht.

```
let $j := 0
for $i in ("pi", "pa", "po")
let $j := $j + 1
return
  <tuple><i>{$i}</i><j>{$j}</j></tuple>
```

Das Ergebnis zeigt, dass *j* in jedem Durchlauf den Wert 1 hat:

```
<tuple><i>pi</i><j>1</j></tuple>,
<tuple><i>pa</i><j>1</j></tuple>,
<tuple><i>po</i><j>1</j></tuple>
```

Das hängt mit den Gültigkeitsbereichen der Variablen zusammen. Der Standard bestimmt, dass eine in einer `for`- oder `let`-Klausel gebundene Variable in allen nachfolgenden Klauseln sichtbar ist. Somit ist oben die zum Wert 0 gebundene Variable *j* sichtbar in der folgenden `for`-Klausel und in jeder Iteration mit dem Wert 0 in der rechten Seite der zweiten `let`-Klausel. Dieses *j* mit Wert 0 wäre auch noch im `return` gültig, hätte nicht die zweite `let`-Klauses für die zu bindende Variable wieder den Bezeichner *j* gewählt. Diese neue Variable *j* überdeckt das alte *j* und wird im Beispiel in allen drei Iterationsschritten an den Wert 1 gebunden und liefert somit im `return` dreimal diesen Wert.

Genauso wenig liefert dann

```
let $sum := 0
for $p in fn:doc("bib.xml")//price
let $sum := $sum + $p
return  <gesamt>{$sum}</gesamt>
```

die Gesamtsumme der Preise im Buchdokument. Vielmehr werden die vier erzeugten Tupel ausgegeben.

```
<gesamt>65.95</gesamt>,
<gesamt>65.95</gesamt>,
<gesamt>55.95</gesamt>,
<gesamt>129.95</gesamt>
```

Das gewünschte Ergebnis lässt sich aber über die `sum`-Funktion produzieren:

```
let $p := fn:doc("bib.xml")//price
return  <gesamt>{fn:sum($p)}</gesamt>
```

7.2.2 where-Klausel

Wie bei SQL dient die `where`-Klausel mit einem Prädikat der Selektion. Die `return`-Klausel wird nur für die Tupel evaluiert, die diese Selektion überlebt haben. In Anlehnung an OQL gibt es die Möglichkeit der Filterung mit Bezug auf Gruppen, ähnlich der `HAVING`-Klausel in SQL.

Zunächst aber ein einfaches Beispiel, etwa die Titel der Bücher, die weniger als $60 kosten.

```
for $b in fn:doc("bib.xml")//book
where $b/price < 60.00
return $b/title
```

Nur ein Buch erfüllt dieses Kriterium[1].

```
<title>Data on the Web</title>
```

Geringfügig aufwendiger eine Abfrage nach Büchern mit mindestens zwei Autoren.

```
for $b in fn:doc("bib.xml")//book
let $c := $b/author
where fn:count($c) > 1
return $b/title
```

Die Ausgabe ist der selbe Buchtitel wie oben.

In `where`-Klauseln kann auch eine existentielle („es existiert ein“) oder universelle („für alle“) Quantifizierung mittels einer Variable nach `some` und `every` vorgenommern werden. Der gesamte Ausdruck wird dann zu wahr ausgewertet, falls eine bzw. alle Belegungen der Variablen wahr ergeben.

```
for $b in fn:doc("bib.xml")//book
where some $a in $b/author satisfies
        ($a/last="Stevens" and $a/first="W.")
return $b/title
```

Hier geht es um Buchtitel mit mindestens einem Autor, der W. Stevens heißt.

```
<title>TCP/IP Illustrated</title>,
<title>Advanced Programming in the UNIX Environment</title>
```

Suchen wir nach Titeln von Büchern, deren Autoren alle W. Stevens heißen, geht das mit

1. wobei wir gegenüber dem Tutorial von Robie nachhelfen mussten. Dort waren alle Preise über $60, die Abfrage forderte < 50.00, trotzdem erschien der Titel!

```
for $b in fn:doc("bib.xml")//book
where every $a in $b/author satisfies
        ($a/last="Stevens" and $a/first="W.")
return $b/title
```

und liefert überraschenderweise einen dritten Titel.

```
<title>TCP/IP Illustrated</title>,
<title>Advanced Programming in the UNIX Environment</title>,
<title>The Economics of Technology and Content for Digital TV</title>
```

Da dieses Buch keine Autoren hat, erfolgt die Auswertung der `where`-Klausel mit dem universellen Quantifizierer auf der leeren Folge. Das ergibt aber immer wahr.

Zuletzt eine Umkehrung der Hierarchie: Titel, die ein Autor geschrieben hat.

```
<author-list>
{
   let $a := fn:doc("bib.xml")//author
   for $lname in fn:distinct-values($a/last),
       $fname in fn:distinct-values($a[last=$lname]/first)
   order by $lname, $fname
   return
      <author>
         <name>{$fname, $lname}</name>
         {
            for $b in fn:doc("bib.xml")//book
            where some $ba in $b/author satisfies
                  ($ba/last=$lname and $ba/first=$fname)
            order by $b/title
            return $b/title
         }
      </author>
}
</author-list>
```

Die Ausgabe lautet

```
<author-list>
  <author>
    <name>Serge Abiteboul</name>
    <title>Data on the Web</title>
  </author>
  <author>
    <name>Peter Buneman</name>
```

```
    <title>Data on the Web</title>
  </author>
  <author>
    <name>W. Stevens</name>
    <title>Advanced Programming in the UNIX Environment</title>
    <title>TCP/IP Illustrated</title>
  </author>
  <author>
    <name>Dan Suciu</name>
    <title>Data on the Web</title>
  </author>
</author-list>
```

7.2.3 order by-Klausel

Durch die order by-Klausel lässt sich die Anordnung der Ergebnisse im `return`-Teil beeinflussen. Zunächst einfach die Buchtitel aufsteigend sortiert.

```
for $t in fn:doc("bib.xml")//title
order by $t
return $t
```

Die Formatierung des Ergebnisses unten hängt im übrigen von der Formatierung des `bib.xml`-Dokuments ab, was ggf. nur unschön ist. Gefährlich ist dagegen ein Leerzeichen nach `<title>`. Es ist Teil des Titels und ändert die Sortierfolge, jedenfalls bei `galax`, auf dem der Test lief!

```
<title>Advanced Programming in the UNIX Environment</title>,
<title>Data on the Web</title>,
<title>TCP/IP Illustrated</title>,
<title>The Economics of Technology and Content for Digital TV</title>
```

Durch sog. *order modifiers* kann die Sortierung beeinflusst werden, z. B. kann aufsteigende oder absteigende Sortierordnung verlangt werden. Mit `empty greatest` bzw. `empty least` können Einträge, die eine leere Sequenz bilden, gesteuert werden, im ersten Fall etwa ganz an den Schluss bei Sortierfolge `ascending` (aufsteigend). Ferner wird bei stabiler Multimengensortierung (Angabe `stable`) die aus dem Quelldokument stammende relative Ordnung gleicher Elemente beibehalten (vgl. auch [49]).

Die folgende Abfrage gibt die Buchtitel sortiert nach dem Nachnamen des 1. Autors aus. Bei gleichem Nachnamen bestimmt der Vorname die Reihenfolge. Ist auch dieser gleich (wie in unserem `bib.xml` Beispiel) soll die relative Reihenfolge der Bücher erhalten bleiben. Das Buch ohne Autoren kommt wegen `empty least` an den Anfang.

```
for $b in fn:doc("bib.xml")//book
let $a1 := $b/author[1]
stable order by $a1/last empty least, $a1/first empty least
return $b/title
```

Das Ergebnis lautet:

```
<title>The Economics of Technology and Content for Digital TV</title>,
<title>Data on the Web</title>,
<title>TCP/IP Illustrated</title>,
<title>Advanced Programming in the UNIX Environment</title>
```

7.2.4 return-Klausel und Element-Konstruktoren

In der `return`-Klausel wird das Ergebnis konstruiert, in der Regel als Element-Sequenz in XML-Schreibweise. Das folgende Beispiel ist also eine gültige XQuery, die ein Dokument mit dem selben Inhalt produziert:

```
<versuch>Der erste Versuch schlägt nicht immer fehl</versuch>
```

Darüberhinaus können sog. *Konstruktoren* verwendet werden. Dabei existiert für jeden der sieben Knotentypen ein Konstruktor, der eine Instanz von diesem Knotentyp erzeugt. Das obige Element `<versuch>` kann man also auch wie folgt erzeugen:

```
element versuch {
  "Der erste Versuch schlägt nicht immer fehl"
}
```

Üblicherweise spielen aber nur Elemente und Attribute eine Rolle für die Ausgabeerzeugung. In geschweiften Klammern können beliebige XQuery-Anweisungen vorkommen. Ebenfalls dürfen diese beliebig ineinander geschachtelt werden. Auf diese Weise können Knoten (Elemente, Attribute etc.) dynamisch erzeugt werden. Das folgende Beispiel

```
<beispiel>
  <p>Der Konstruktor:</p>
  <p>fn:doc("bib.xml")//book[1]/title</p>
  <p>erzeugt:</p>
  <p>{fn:doc("bib.xml")//book[1]/title}</p>
</beispiel>
```

erzeugt die Ausgabe:

```
<beispiel>
  <p>Der Konstruktor:</p>
  <p>fn:doc("bib.xml")//book[1]/title</p>
  <p>erzeugt:</p>
  <p><title>TCP/IP Illustrated</title></p>
</beispiel>
```

7.3 Die Positionsvariable at

Die `for`-Klausel in Verbindung mit `at` erlaubt den Zugriff auf eine Positionsvariable. Im folgenden Beispiel werden die Buchtitel durchnumeriert.

```
for $t at $i in fn:doc("bib.xml")//title
return <title pos="{$i}">{fn:string($t)}</title>
```

Ausgabe:

```
<title pos="1">TCP/IP Illustrated</title>,
<title pos="2">Advanced Programming in the UNIX Environment</title>,
<title pos="3">Data on the Web</title>,
<title pos="4">The Economics of Technology and Content for Digital
TV</title>
```

Im Tutorial von Robie [50] wird gezeigt, welche Möglichkeiten sich bei Tabellen ergeben, die als XHTML-Dokumente vorliegen und auf die man damit per XQuery zugreifen kann.

7.4 Weitere Möglichkeiten von XQuery

XQuery ist nicht nur eine Abfragesprache, sondern eine vollständige funktionale Sprache. Mit XQuery können also verschiedene Funktionen zur Verarbeitung von XML-Dokumenten implementiert werden. Wir erwähnen hier kurz einige dieser Eigenschaften und verweisen für weitere Details auf den W3C-Standard.

7.4.1 XQuery Conditional Expressions (If-then-else-Konstrukt)

Für Bücher mit einem Herausgeber erzeuge das Element `<herausgeber>`, sonst ein `<autor>`-Element.

```
for $b in fn:doc("bib.xml")//book
return
<buch>
  {$b/title}
  {if (fn:exists($b/editor)) then
    <herausgeber>{$b/editor[1]/last}</herausgeber>
  else
    <autor>{$b/author[1]/last}</autor>
  }
</buch>
```

Ausgabe:

```
<buch>
  <title>TCP/IP Illustrated</title>
  <autor><last>Stevens</last></autor>
</buch>,
<buch>
  <title>Advanced Programming in the UNIX Environment</title>
  <autor><last>Stevens</last></autor>
</buch>,
<buch>
  <title>Data on the Web</title>
  <autor><last>Abiteboul</last></autor>
</buch>,
<buch>
  <title>The Economics of Technology and Content for Digital TV</title>
  <herausgeber><last>Gerbarg</last></herausgeber>
</buch>
```

7.4.2 Eingebaute Funktionen

XQuery hat eine Menge vordefinierter Funktionen und Operatoren, die zum Teil von anderen Sprachen bekannt sind. Lehner und Schöning [29] erwähnen `fn:count()`, `fn:exists()`, `fn:one-or-more()` zur Analyse von Sequenzen und von Knoten mit `fn:node-name()`, `fn:string()`, `fn:number()`. Das Verfolgen von Referenzen mit `fn:id()` und `fn:idref()` ist zunächst nur innerhalb eines Dokuments möglich.

Weiterhin existieren aus SQL bekannte Funktionen wie `fn:min()`, `fn:max()`, `fn:sum()`, das oben bereits erwähnte `fn:count()` und

`fn:avg()`. Die folgende Abfrage liefert z. B. Bücher, die einen höheren Preis als der Durchschnitt haben:

```
let $b := fn:doc("bib.xml")//book
let $average := fn:avg($b/price)
return $b[price > $average]
```

Das Ergebnis lautet:

```
<book year="1999">
  <title>
    The Economics of Technology and Content for Digital TV
  </title>
  <editor>
    <last>Gerbarg</last>
    <first>Darcy</first>
    <affiliation>CITI</affiliation>
  </editor>
  <publisher>Kluwer Academic Publishers</publisher>
  <price>129.95</price>
</book>
```

Dazu kommen String-Verarbeitungsfunktionen `fn:substring()`, `fn:matches()` und kalendarische Extraktions- und Konvertierungsfunktionen `fn:get-year-from-yearMonthDuration()`, `fn:adjust-dateTime-to-timezone()`.

Eine ausführliche Beschreibung dieser Funktionen ist auf `http://www.w3.org/TR/xquery-operators` zu finden [27].

7.4.3 Weitere XQuery-Besonderheiten

Wir gehen hier nicht auf weitere Besonderheiten von XQuery ein. Dazu gehören

- das Modulkonzept zur Einbindung weiterer Bibliotheksmodule,
- das Verarbeitungskonzept mit statischem und dynamischem Kontext,
- die Überführung der Ergebnissequenz in ein XML-Dokument,
- die Vergleichsausdrücke für (komplexe) Knoten und Gruppierungen mit und ohne Duplikatseliminierung (vgl. hierzu auch die Arbeiten von Wegner et al. [33, 34]).

7.5 Einschub XQueryX

XQueryX ist eine XML-Sprache für XQuery. Die Spezifikation findet man auf der Seite `http://www.w3.org/TR/xqueryx`.

Was kann man sich unter einer XML-Syntax für XQuery vorstellen? Eine Abfrage, z. B. die folgende (aus XQueryX 1.0: Titel aller Bücher von Addison-Wesley, die nach 1991 erschienen sind)

```
<bib>
{
  for $b in doc("bib.xml")/bib/book
  where $b/publisher = "Addison-Wesley" and $b/@year > 1991
  return
    <book year="{ $b/@year }">
      { $b/title }
    </book>
}
</bib>
```

lässt sich als ein XML-Dokument hinschreiben, z. B. wie folgt:

```
<?xml version="1.0"?>
<xqx:module xmlns:xqx="http://www.w3.org/2005/XQueryX"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="http://www.w3.org/2005/XQueryX
                                http://www.w3.org/2005/XQueryX/xqueryx.xsd">
  <xqx:mainModule>
    <xqx:queryBody>
      <xqx:elementConstructor>
        <xqx:tagName>bib</xqx:tagName>
        <xqx:elementContent>
          <xqx:flworExpr>
            <xqx:forClause>
              <xqx:forClauseItem>
                <xqx:typedVariableBinding>
                  <xqx:varName>b</xqx:varName>
                </xqx:typedVariableBinding>
                <xqx:forExpr>
                  <xqx:pathExpr>
                    <xqx:stepExpr>
                      <xqx:filterExpr>
                        <xqx:functionCallExpr>
                          <xqx:functionName>doc</xqx:functionName>
                          <xqx:arguments>
                            <xqx:stringConstantExpr>
                              <xqx:value>bib.xml</xqx:value>
                            </xqx:stringConstantExpr>
                          </xqx:arguments>
                        </xqx:functionCallExpr>
                      </xqx:filterExpr>
                    </xqx:stepExpr>
                    <xqx:stepExpr>
                      <xqx:xpathAxis>child</xqx:xpathAxis>
                      <xqx:nameTest>bib</xqx:nameTest>
                    </xqx:stepExpr>
```

```
            <xqx:stepExpr>
              <xqx:xpathAxis>child</xqx:xpathAxis>
              <xqx:nameTest>book</xqx:nameTest>
            </xqx:stepExpr>
          </xqx:pathExpr>
        </xqx:forExpr>
      </xqx:forClauseItem>
    </xqx:forClause>
    <xqx:whereClause>
      <xqx:andOp>
        <xqx:firstOperand>
          <xqx:equalOp>
            <xqx:firstOperand>
              <xqx:pathExpr>
                <xqx:stepExpr>
                  <xqx:filterExpr>
                    <xqx:varRef>
                      <xqx:name>b</xqx:name>
                    </xqx:varRef>
                  </xqx:filterExpr>
                </xqx:stepExpr>
                <xqx:stepExpr>
                  <xqx:xpathAxis>child</xqx:xpathAxis>
                  <xqx:nameTest>publisher</xqx:nameTest>
                </xqx:stepExpr>
              </xqx:pathExpr>
            </xqx:firstOperand>
            <xqx:secondOperand>
              <xqx:stringConstantExpr>
                <xqx:value>Addison-Wesley</xqx:value>
              </xqx:stringConstantExpr>
            </xqx:secondOperand>
          </xqx:equalOp>
        </xqx:firstOperand>
        <xqx:secondOperand>
          <xqx:greaterThanOp>
            <xqx:firstOperand>
              <xqx:pathExpr>
                <xqx:stepExpr>
                  <xqx:filterExpr>
                    <xqx:varRef>
                      <xqx:name>b</xqx:name>
                    </xqx:varRef>
                  </xqx:filterExpr>
                </xqx:stepExpr>
                <xqx:stepExpr>
                  <xqx:xpathAxis>attribute</xqx:xpathAxis>
                  <xqx:nameTest>year</xqx:nameTest>
                </xqx:stepExpr>
              </xqx:pathExpr>
            </xqx:firstOperand>
            <xqx:secondOperand>
              <xqx:integerConstantExpr>
                <xqx:value>1991</xqx:value>
              </xqx:integerConstantExpr>
            </xqx:secondOperand>
          </xqx:greaterThanOp>
        </xqx:secondOperand>
      </xqx:andOp>
    </xqx:whereClause>
    <xqx:returnClause>
      <xqx:elementConstructor>
        <xqx:tagName>book</xqx:tagName>
        <xqx:attributeList>
          <xqx:attributeConstructor>
            <xqx:attributeName>year</xqx:attributeName>
            <xqx:attributeValueExpr>
              <xqx:pathExpr>
```

```
                    <xqx:stepExpr>
                      <xqx:filterExpr>
                        <xqx:varRef>
                          <xqx:name>b</xqx:name>
                        </xqx:varRef>
                      </xqx:filterExpr>
                    </xqx:stepExpr>
                    <xqx:stepExpr>
                      <xqx:xpathAxis>attribute</xqx:xpathAxis>
                      <xqx:nameTest>year</xqx:nameTest>
                    </xqx:stepExpr>
                  </xqx:pathExpr>
                </xqx:attributeValueExpr>
              </xqx:attributeConstructor>
            </xqx:attributeList>
            <xqx:elementContent>
              <xqx:pathExpr>
                <xqx:stepExpr>
                  <xqx:filterExpr>
                    <xqx:varRef>
                      <xqx:name>b</xqx:name>
                    </xqx:varRef>
                  </xqx:filterExpr>
                </xqx:stepExpr>
                <xqx:stepExpr>
                  <xqx:xpathAxis>child</xqx:xpathAxis>
                  <xqx:nameTest>title</xqx:nameTest>
                </xqx:stepExpr>
              </xqx:pathExpr>
            </xqx:elementContent>
          </xqx:elementConstructor>
        </xqx:returnClause>
      </xqx:flworExpr>
    </xqx:elementContent>
  </xqx:elementConstructor>
 </xqx:queryBody>
</xqx:mainModule>
</xqx:module>
```

Diese Form einer Query als XML-Dokument ist recht unlesbar und hat keinen so großen praktischen Nutzen. Man kann aber so zeigen, dass das Modell abgeschlossen ist, man kann Queries auf Queries starten, gewisse Syntaxanalysen mit dem Parser erledigen und man kann Queries einheitlich abspeichern. Dafür gibt es ein XML Schema, das alle formulierbaren Queries abdeckt. Das XML Schema zu XQueryX findet sich unter `http://www.w3.org/TR/xqueryx#Schema`.

8 SQL/XML

8.1 XML und Datenbanken

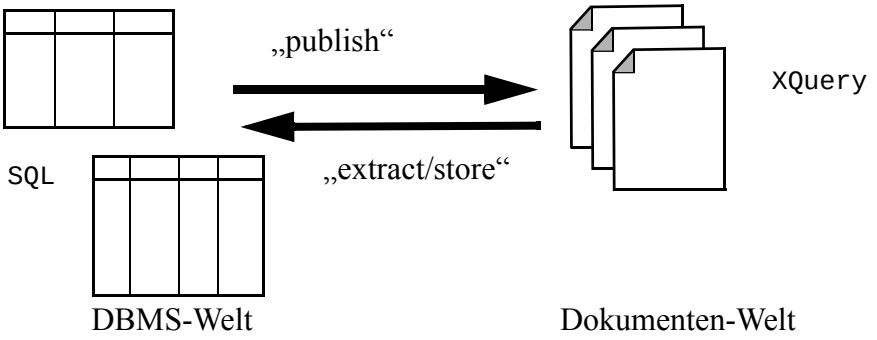
8.1.1 Was ist SQL/XML?

Die zunehmende Verbreitung von XML als Datenrepräsentations- und Datenaustauschsprache bringt das offensichtliche Problem mit sich, wie die dokumentenorientierte Sicht von XML mit der tabellenorientierten Sicht der Datenbankwelt in Einklang gebracht werden kann. Errungenschaften der heute ausgereiften relationalen DBMS-Technik sind:

- Transaktionsverarbeitung
- Hohes Datenvolumen
- Integrität, Konsistenz durch Redundanzfreiheit (Normalisierung)
- Recovery, Backup
- Indexunterstützung, Aggregationen, Data Mining
- Einheitliche Schnittstelle durch SQL (Structured Query Language)

Mit *SQL/XML* gibt es nun einen Ansatz, beide Welten miteinander zu verbinden. SQL/XML ist eine ISO-Norm [37], die als Anhang 14 zu SQL:2003 zunächst als *First Edition* veröffentlicht wurde und die Ende 2005 in einer zweiten Auflage (*Second Edition*) zur Abstimmung gestellt wurde (*Final Committee Draft*, FCD). In 2008 und 2011 gab es Überarbeitungen der Norm. Wir zitieren hier im Wesentlichen die Arbeiten von Eisenberg und Melton [32], die Übersicht von Funderburk, Malaika und Reinwald [35], die noch auf SQL/XML:2003 beruht, sowie das Buch von Can Türker[36]. Eine weitere gute Quelle sind die Seiten der `sqlx.org`.

Die Aufgabe von SQL/XML besteht aus einem Austausch in zwei Richtungen. Die eine Richtung, hier als „publish“ bezeichnet, baut aus Tabel-



lendaten XML-Dokumente auf. Hierfür sind Publikationsfunktionen nötig, die Aufgabe ist sehr ähnlich der Erzeugung von dynamischen Web-Seiten aus Datenbanken, etwa mit MySQL und PHP.

Zugleich kann man die Umformung von SQL-Daten in XML-Werte (values) als Sichtgenerierung im relationalen Modell betrachten, wobei eine Speicherung in XML-Dokumenten und spätere Weiterverarbeitung eine sog. *materialisierte Sicht* (materialized view) wäre, was *nicht* die Regel in SQL ist. In SQL wird eine Abfrage auf einer Sicht (die ja wiederum auch auf einer Abfrage beruht) zu einer einzigen Abfrage auf den Basistabellen zusammengefasst und dann ausgewertet, so dass nie Resultate aus potentiell veralteten Daten geliefert werden. Die Publish-Funktion als Sichtgenerierung macht es ferner möglich, Strukturen zu erzeugen, die eigentlich im relationalen Modell nicht vorgesehen sind, z.B. geordnete Listen, Baumstrukturen, usw.

Für die Gegenrichtung gibt es zwei Ansätze. Zum Einen kann man einen Datentyp XML-Type für „XML-wertige“ Spalten definieren. Eine solche Spalte nimmt komplette Dokumente auf. In der IBM DB2 Implementierung (DB2 XML Extender) heißt diese Speicherform **XML Column** und ist gedacht für „intakte“ Dokumente. Damit wird zum Ausdruck gebracht, dass es aus rechtlichen Gründen oft notwendig ist, XML-Daten (Bestellungen, E-Mails, Zusagen, Abwicklungsdaten, usw.) unverändert als Ganzes abzuspeichern.

Trotzdem wird für diese Form eine Indexunterstützung angeboten (sog. *side tables*), die sich als *Pfadausdruck* („mit welchen Werten soll der Index arbeiten, wonach wird indexiert“) in einer sogenanten *DAD* angeben lassen. *DAD* steht für *Document Access Definition*, selbst auch ein XML-Dokument, das zunächst angibt, wie man von relationalen Tabellen zu XML kommt (also eher die publish-Seite), aber eben auch für Indexfestlegungen und die Gegenrichtung (relationale Datengewinnung aus XML-Dokumenten) gebraucht wird. Die *DAD-Pfadausdrücke* sind defacto eingeschränkte XPath-Ausdrücke. Die Norm selbst kennt allerdings keine DADs.

Zugleich lässt sich aus SQL heraus über eine *stored procedure* Schnittstelle auf Inhalte, etwa einen einzelnen Elementtext, solcher Dokumente zugreifen, die als Ganzes in einer Spalte abgespeichert sind. Die Schnittstelle ist praktisch der XQuery-Prozessor. In DB2 erfolgt die physische Speicherung entweder als XMLVarchar für bis zu 3K Größe in der Datenbank selbst, als XMLCLOB (XML character large object) für bis zu 32K auch in der Datenbank, und als XMLFILE im lokalen Dateisystem. Gespeicherte Dokumente lassen sich beim Einfügen in der Spalte und auch nachträglich validieren. Hierzu führt die Norm für den Spaltentyp XML Variationen ein, speziell auch für den Fall, dass der gespeicherte Wert kein vollständiges, valides Dokument ist sondern z.B. nur eine Sequenz von unterschiedlichen Werten.

Die zweite, etwas spannendere Speicherform ist *XML Collection Storage* (DB2 Sprechweise), in SQL/XML als *XMLTable* Pseudofunktion bekannt. Hier geht es um das „Zerkleinern“ (shredding) der XML-Strukturen, um die Datenelemente in den „flachen“ (1. Normalform!) Tabellen des DBMS abzulegen.

Der Prozess ist analog zur Normalisierung, d.h. eine hierarchische 1:*n*-Beziehung (z.B. BESTELLUNGEN:BESTELLPOSTEN - Bestellungen bestehen aus mindestens einem, meist mehreren Bestellposten, ein Bestellposten gehört zu genau einer Bestellung), wird in zwei Tabellen abgelegt und über Fremdschlüsselattribute verknüpft (jeder Bestellposten hat eine Spalte Bestell-Id, der dort eingetragene Wert ist der „fremde“ Schlüssel der Bestellung aus der BESTELLUNGEN-Tabelle).

Die Schwierigkeit ist allerdings, dass XML-Dokumente nicht so strikt aufgebaut sind. So werden Beziehungen mit Schlüsseleigenschaft meist über Attribute vom Typ ID und Verweise darauf mit IDREF abgebildet, ein Element könnte aber auch mit seinem Textinhalt als Schlüssel dienen, was dann in der Schemadefinition als solches zu markieren wäre und bei UPDATE- oder INSERT-Operationen auf dem Dokument geprüft werden müsste. Die Arbeiten in XML hierzu stehen erst noch am Anfang.

Ein „geschreddertes“ Dokument kann als virtuelle Tabelle in eine bereits existierende SQL-Basistabelle eingefügt werden oder an einem regulären SQL-Ausdruck, etwa auch einem Join, teilnehmen. Details folgen weiter unten.

8.2 XML-Publikationsfunktionen

8.2.1 Anwendungsbeispiel

Als Anwendungsbeispiel verwenden wir vier Tabellen aus [35] mit Bestellungen, Bestellposten, Produkten und Kunden:

ORDER_ID	CUSTOMER_ID	SHIP_METHOD	date	STATUS
100	777	UPS	23.10.99	shipped
101	777	USPS	25.01.02	accepted
102	888	UPS	05.02.02	shipped

Tab. 8–1 *Tabelle „Order“*

ORDER_ID	ITEM_NUMBER	PRODUCT_ID	QUANTITY	PRICE
100	1	1001	1	108,25
100	2	1002	2	17,42
101	1	1002	5	17,42
102	1	1003	1	104,10

Tab. 8–2 *Tabelle „Order_Items“*

PRODUCT_ID	DESCRIPTION	STOCK	PRICE
1001	Sound Blaster Audigy MP3+	1	108,25
1002	Travel Alarm With Radio	1	17,42
1003	5.1 Surround Sound Speaker System	1	104,10

Tab. 8–3 *Tabelle „Product“*

CUSTOMER_ID	NAME	ADDRESS
777	William P Barnes	104 West Avery Lane, CA
888	Shirley Jackson	1344 Pennsylvania Ave, OH

Tab. 8–4 *Tabelle „Customer“*

8.2.2 Default View

Die zunächst recht elementare Einsicht ist, dass jede relationale Tabelle (damit auch jedes SQL-Abfrageergebnis) wie folgt in ein XML-Dokument gewandelt werden kann.

```
<table-name>
  <row>
    <column1-name>data</column1-name>
    <column2-name>data</column2-name>
    <column3-name>data</column3-name>
    ...
  </row>
  <row>
    <column1-name>data</column1-name>
    <column2-name>data</column2-name>
    <column3-name>data</column3-name>
    ...
  </row>
  ...
</table-name>
```

Diese Sicht wird deshalb *default view* genannt. Sie könnte mit XSLT umgeformt werden. Für die Tabelle **Customer** oben liefert der folgende Oracle-spezifische Aufruf

```
SELECT DBMS_XMLGEN.getXML('SELECT * FROM "Customer"')
FROM DUAL;
```

das folgende Dokument:

```
<ROWSET>
 <ROW>
  <CUSTOMER_ID>777</CUSTOMER_ID>
  <NAME>William P Barnes</NAME>
  <ADDRESS>104 West Avery Lane, CA</ADDRESS>
 </ROW>
 <ROW>
  <CUSTOMER_ID>888</CUSTOMER_ID>
  <NAME>Shirley Jackson</NAME>
  <ADDRESS>1344 Pennsylvania Ave, OH</ADDRESS>
 </ROW>
</ROWSET>
```

Eine etwas gefälligere Umformung ließe sich mit der folgenden in eine SQL-Abfrage eingebetteten XQuery über der *default view* bewerkstelligen (ebenfalls für Oracle):

```
SELECT XMLQUERY('
  <customerList>{
    for $c in ora:view("UEBUSER104","""Customer""")/ROW
    return <customer id="{ $c/CUSTOMER_ID }">
             <name>{ fn:string($c/NAME) }</name>
             <address>{ fn:string($c/ADDRESS) }</address>
           </customer>
  }</customerList>' RETURNING CONTENT NULL ON EMPTY)
FROM DUAL;
```

Das Ergebnis lautet:

```
<customerList>
  <customer id="777">
    <name>William P Barnes</name>
    <address>104 West Avery Lane, CA</address>
  </customer>
  <customer id="888">
    <name>Shirley Jackson</name>
```

```
    <address>1344 Pennsylvania Ave, OH</address>
  </customer>
</customerList>
```

8.2.3 Funktionen zur XML-Erzeugung

Hier geht es nun um spezielle SQL-Erweiterungen, die sog. *publishing functions*, auch als SQLX-Funktionen bekannt:

- XMLElement und XMLAttributes konstruieren XML-Elemente mit Attributen
- XMLForest konstruiert eine Sequenz von XML-Elementen
- XMLConcat verkettet XML-Elemente
- XMLAgg aggregiert XML-Elemente
- XMLComment zur Erzeugung von Kommentaren
- XMLPI zur Erzeugung von Prozessorinstruktionen

Die folgenden Abfragen wurden unter Oracle 11 getestet, wobei die Beispielabfragen aus [35] z.T. einige Fehler enthielten.

Abfrage 1:

```
SELECT XMLELEMENT(
         NAME "order",
         XMLATTRIBUTES(o.ORDER_ID),
         XMLELEMENT(NAME "signdate", o."date"),
         XMLELEMENT(
           NAME "order_value",
           (SELECT SUM(QUANTITY * PRICE)
           FROM "Order_Items" oi
           WHERE oi.ORDER_ID = o.ORDER_ID)
         )
       )
FROM "Order" o
WHERE STATUS = 'shipped';
```

Das Ergebnis sind zwei Zeilen, von denen jede ein XML-Element `<order>` enthält:

```
<order ORDER_ID="100">
  <signdate>23.10.99</signdate>
```

```
  <order_value>143,09</order_value>
</order>
<order ORDER_ID="102">
  <signdate>05.02.02</signdate>
  <order_value>104,10</order_value>
</order>
```

Hinweis: Der Standard erlaubt in XMLAttributes die Form

xml-attribute-value [AS *xml-attribute-name*]

d.h., man hätte oben `XMLATTRIBUTES(o.ORDER_ID AS "id")` schreiben können und dann den üblichen Attributnamen `id` eingetragen. Die Anführungszeichen um `id` müssen gesetzt werden, damit der Attributname nicht komplett in Großbuchstaben ausgegeben wird. In Abfrage 4 unten wird diese Form gewählt.

Abfrage 2 verwendet *XMLForest*. Die Funktion ist eine Abkürzungsmethode zur Erzeugung einer Folge von Elementen aus einer variablen Liste von SQL-Werten, die sich ggf. auch zur Laufzeit aus Ausdrücken über Spaltenwerten errechnen. In diesem Fall muss ein Alias-Name für das zu generierte Element genannt werden (`AS ...`), sonst kann darauf verzichtet werden und XMLForest nimmt dann den Spaltennamen wie im default view:

```
SELECT XMLELEMENT(
         NAME "order",
         XMLFOREST(
           ORDER_ID AS "ID",
           CUSTOMER_ID AS "customerID",
           SHIP_METHOD AS "shipMethod"
         )
       )
FROM "Order"
WHERE STATUS = 'shipped';
```

Ergebnis 2:

```
<order>
  <ID>100</ID>
  <customerID>777</customerID>
  <shipMethod>UPS</shipMethod>
</order>
```

```
<order>
  <ID>102</ID>
  <customerID>888</customerID>
  <shipMethod>UPS</shipMethod>
</order>
```

Abfrage 3 verwendet *XMLConcat*. Die Funktion nimmt eine variable Anzahl von XML-Werten als Ausdrücke und erzeugt einen einzelnen XML-Wert als Folge von XML-Werten:

```
SELECT
  XMLELEMENT(
    NAME "order",
    XMLCONCAT(
     XMLELEMENT(NAME "ID", ORDER_ID),
     XMLELEMENT(NAME "customerID", CUSTOMER_ID),
     XMLELEMENT(NAME "shipMethod", SHIP_METHOD)
    )
  )
FROM "Order"
WHERE STATUS = 'shipped';
```

Das Ergebnis 3 ist gleich dem vorherigen Ergebnis bei Abfrage 2.

Abfrage 4 verwendet *XMLAgg*, die Aggregationsfunktion. Mit ihr lassen sich 1:*n*-Beziehungen in XML abbilden. Man beachte deshalb den Join unten, mit dem Bestellungen (Order) und Bestellposten (Order_Items) verknüpft werden:

```
SELECT XMLELEMENT(
  NAME "order",
  XMLATTRIBUTES(o.ORDER_ID AS "ID"),
  XMLAGG(
    XMLELEMENT(
      NAME "item",
      XMLATTRIBUTES(oi.ITEM_NUMBER AS "ItemNumber"),
      XMLFOREST(oi.PRODUCT_ID AS "ProductID",
                oi.QUANTITY AS "Quantity")
    )
    ORDER BY oi.ITEM_NUMBER
  )
)
FROM "Order" o, "Order_Items" oi
WHERE o.ORDER_ID = oi.ORDER_ID
GROUP BY o.ORDER_ID;
```

Ergebnis 4:

```
<order ID="100">
  <item ItemNumber="1">
    <ProductID>1001</ProductID>
    <Quantity>1</Quantity>
  </item>
  <item ItemNumber="2">
    <ProductID>1002</ProductID>
    <Quantity>2</Quantity>
  </item>
</order>
<order ID="101">
  <item ItemNumber="1">
    <ProductID>1002</ProductID>
    <Quantity>5</Quantity>
  </item>
</order>
<order ID="102">
  <item ItemNumber="1">
    <ProductID>1003</ProductID>
    <Quantity>1</Quantity>
  </item>
</order>
```

Die letzte Abfrage 5 ist aufgebaut wie Abfrage 4, nur erweitert um Kundenname und Produktbeschreibung, damit auch diese Tabellen mal verwendet werden. Abfrage 5 ist nicht in [35] enthalten.

```
SELECT XMLELEMENT(NAME "order",
  XMLATTRIBUTES(o.ORDER_ID AS "ID",
                c.NAME AS "CustomerName"),
  XMLAGG(
    XMLELEMENT(
      NAME "item",
      XMLATTRIBUTES(oi.ITEM_NUMBER AS "ItemNumber"),
      XMLFOREST(
        oi.PRODUCT_ID AS "ProductID",
        p.DESCRIPTION AS "ProductDescription",
        oi.QUANTITY AS "Quantity"
      )
    )
    ORDER BY oi.ITEM_NUMBER
  )
)
```

```
FROM "Order" o, "Order_Items" oi, "Customer" c, "Product" p
WHERE o.ORDER_ID = oi.ORDER_ID AND
      c.CUSTOMER_ID = o.CUSTOMER_ID AND
      p.PRODUCT_ID = oi.PRODUCT_ID
GROUP BY o.ORDER_ID, c.NAME;
```

Ergebnis 5:

```
<order ID="100" CustomerName="William P Barnes">
  <item ItemNumber="1">
    <ProductID>1001</ProductID>
    <ProductDescription>
      Sound Blaster Audigy MP3+
    </ProductDescription>
    <Quantity>1</Quantity>
  </item>
  <item ItemNumber="2">
    <ProductID>1002</ProductID>
    <ProductDescription>
      Travel Alarm With Radio
    </ProductDescription>
    <Quantity>2</Quantity>
  </item>
</order>
<order ID="101" CustomerName="William P Barnes">
  <item ItemNumber="1">
    <ProductID>1002</ProductID>
    <ProductDescription>
      Travel Alarm With Radio
    </ProductDescription>
    <Quantity>5</Quantity>
  </item>
</order>
<order ID="102" CustomerName="Shirley Jackson">
  <item ItemNumber="1">
    <ProductID>1003</ProductID>
    <ProductDescription>
      5.1 Surround Sound Speaker System
    </ProductDescription>
    <Quantity>1</Quantity>
  </item>
</order>
```

8.2.4 Document Access Definitions in DB2 Extender

In [35] folgt dann ein Abschnitt über das Publizieren von XML-Dokumenten aus relationalen Tabellen mittels einer sog. Document Access Definition (DAD) als Teil von IBMs DB2 XML Extender. DADs sind selbst wieder XML-Dokumente. DADs bestehen aus einem SQL SELECT, das die Datenanlieferung regelt und einer an XML Schema erinnernden Strukturangabe, die regelt, welche Werte zu Attributen werden und welche zu Elementen, sowie Angaben, wie die Elemente zu schachteln sind.

Die ISO-Norm enthält keine DADs, deswegen übergehen wir diesen Teil. Klar ist, dass die automatische Erzeugung von XML-Dokumenten nach einer XML-Schema-Angabe eine sehr attraktive Möglichkeit ist.

8.3 Die Funktionen XMLQuery und XMLCast

Mit der Funktion *XMLQuery* [32] kann das Ergebnis einer XQuery-Abfrage an SQL weitergereicht werden. Eine Einsatzmöglichkeit ist die Suche mit Werten aus den Tabellen nach XML-Werten in einem XML-Dokument, zum Beispiel für einen Join oder um daraus wieder ein XML-Dokument zu erzeugen. Die allgemeine Syntax lautet:

```
XMLQUERY ( xquery-expression
  [ PASSING { BY REF | BY VALUE } argument-list ]
  RETURNING { CONTENT | SEQUENCE } [ BY REF | BY VALUE ]
            { NULL ON EMPTY | EMPTY ON EMPTY } )
```

Die *xquery-expression* ist der eigentliche XQuery-Ausdruck, der als Literal angegeben werden muss (also mit einfachen Anführungszeichen). Die PASSING-Klausel enthält eine durch Kommas getrennte Liste von Argumenten. Jeder Eintrag der Liste hat die Syntax

```
value-expression [ AS identifier ] [ BY REF | BY VALUE ]
```

und bindet den Wert eines SQL-Ausdrucks an eine globale XQuery-Variable in dem XQuery-Ausdruck. Eines der Argumente darf ohne den Zusatz `AS` *identifier* stehen und definiert den Kontextknoten für die XQuery-Abfrage. Für die gesamte Liste oder für jedes einzelne Argument

kann bestimmt werden, ob die Übergabe als kopierter Wert oder per Referenz erfolgt.

Im Beispiel aus [32] wird das Stringliteral `'A.Eisenberg'` an `var1` gebunden. Das zweite Argument `buying_agents` wird an den aktuellen Kontext des XQuery-Ausdrucks gebunden, weil keine Variable angegeben ist (`AS` ... fehlt).

```
SELECT top_price,
  XMLQUERY(
     'for $cost in /buyer/contract/item/amount
     where /buyer/name = $var1
     return $cost'
  PASSING BY VALUE
    'A.Eisenberg' AS var1, buying_agents
  RETURNING SEQUENCE BY VALUE EMPTY ON EMPTY)
FROM buyers;
```

Das zurückgelieferte Resultat kann per Referenz (`BY REF`) oder als kopierter Wert (`BY VALUE`) in SQL eingehen. Mit `RETURNING CONTENT` oder `RETURNING SEQUENCE` wird festgelegt, ob der Rückgabetyp `XML(CONTENT)` oder `XML(SEQUENCE)` sein soll (siehe Abschnitt 8.4). Im Fall eines leeren Ergebnisses wird mit dem Zusatz `NULL ON EMPTY` ein NULL-Wert zurückgeliefert, bei `EMPTY ON EMPTY` eine leere Sequenz.

Mit der Funktion *XMLCast* können Typumwandlungen analog zu SQL CAST vorgenommen werden. XMLCast hat die Syntax

```
XMLCAST ( value-expression AS type )
```

wobei einer der beiden Angaben sich auf einen XML-Typ beziehen muss. Intern wird ein SQL CAST vorgenommen. Generell kann man an dieser Stelle anmerken, dass sich das (überreichliche) Angebot an SQL-Typen nicht immer gut mit den XML-Typen verträgt, wobei das XQuery-Datenmodell ursprünglich auf dem Infoset aufbaute und nur drei atomare Datentypen (Boolean, double, string) kannte. Mit XQuery 1.0 und XPath 2.0 wurden deren Datenmodell an XML Schema angepasst und damit wesentlich erweitert. Es bildet jetzt die Basis für SQL/XML.

8.4 XML-Datentyp

Wie in der Einleitung beschrieben, kann die Abspeicherung von XML-Werten in einer Tabellenspalte vom Typ XML erfolgen. Die zweite Auflage der SQL/XML-Norm von 2003 parametrisiert diesen Typ nun, wobei die Typvariationen eine Hierarchie bilden.

XML(SEQUENCE) ist der allgemeinste Typ. Jeder XML-Wert in SQL/XML ist entweder der SQL-Nullwert oder eine XQuery-Sequenz und damit eine Ausprägung dieses Typs. Teile einer solchen Sequenz könnten Elemente sein, Prozessorinstruktionen, Texte (Literale), numerische Werte, auch nicht-wohlgeformte Werte, alle durch Kommata getrennt.

XML(CONTENT(ANY)) ist die zweite Variation unter XML(SEQUENCE). Jeder XML-Wert, der entweder einen Nullwert oder einen XQuery-*Dokumentknoten* bildet (einschließlich aller Kinder des Knotens), ist von diesem Typ. Ausprägungen können auch nicht-wohlgeformte Dokumente sein, etwa ein Dokument, das mehr als ein Wurzelelement besitzt. Das kann als Zwischenresultat einer Abfrage entstehen, das später zu einem wohlgeformten Dokumentknoten zurechtgestutzt wird.

XML(CONTENT(UNTYPED)) ist wiederum eine Untervariation von XML(CONTENT(ANY)) und bezieht sich auf noch nicht durchgeführte Schema-Validierungen.

XML(DOCUMENT(ANY)) ist im Gegensatz zu XML(CONTENT(ANY)) ein Dokumentknoten mit genau einem Wurzelelement (und ggf. Prozessorinstruktionen und Kommentaren).

XML(DOCUMENT(UNTYPED) ist eine Ableitung sowohl des Supertyps XML(DOCUMENT(ANY)) als auch XML(CONTENT(UNTYPED)) und teilt deren Eigenschaften.

XML(CONTENT(XMLSCHEMA)) und **XML(DOCUMENT(XMLSCHEMA))** sind Variationen der Typen XML(CONTENT(ANY)) bzw. XML(DOCUMENT(ANY)). Der enthaltene XML-Wert muss Typanmerkungen eines XML-Schemas besitzen.

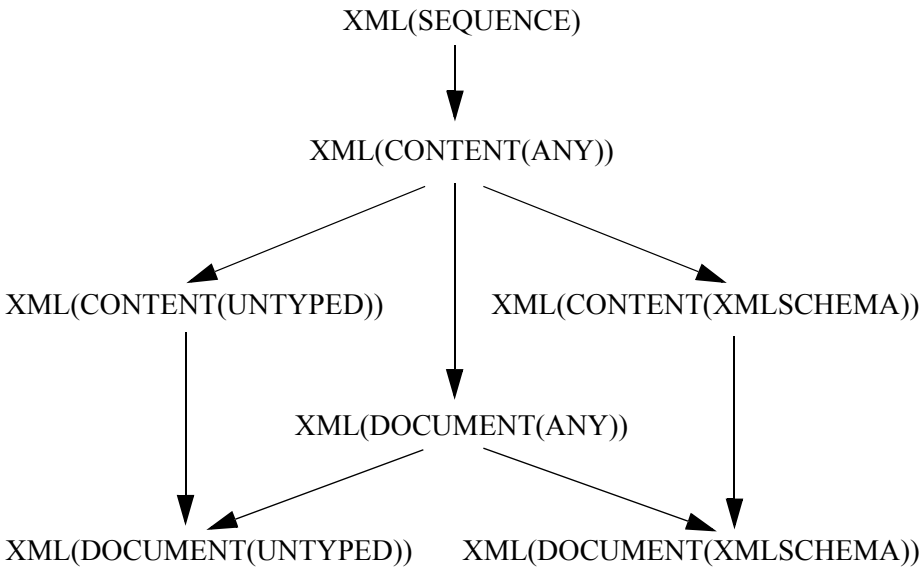


Abb. 8–1 *Hierarchie der XML-Typen von SQL/XML*

8.5 Prädikate

Der Standard enthält ein VALID-Prädikat, mit dem ein gegebener XML-Wert auf Schemakonformität geprüft werden kann, wobei die Schemata im Namensraum des Werts registriert sein müssen. Dies soll verhindern, dass Fremde mittels einer Reihe von „Test-Schemata“ in einer Art von Reverse Engineering den Metadatenaufbau eines Unternehmens herausbekommen können.

Die Syntax zum Testen allgemeiner XML-Werte lautet

```
xml-value IS [ NOT ] VALID
  [ identity-constraint-option ]
  [ validity-target ]
```

Neben dem VALID-Prädikat gibt es noch die folgenden weiteren:

- DOCUMENT-Prädikat (ob der Wert ein wohlgeformtes XML-Dokument ist)

- CONTENT-Prädikat für die XML(CONTENT)-Typen
- XMLEXISTS-Prädikat (in Verbindung mit einer XQuery)

Wir gehen hierauf nicht weiter ein.

8.6 XMLTable

Wie auch in der Einleitung besprochen, ist die zweite Speicherform für XML-Dokumente die „geschredderte" Speicherung, d.h. hierarchische Beziehungen werden aufgebrochen und in Tabellen in SQL abgespeichert. Diese Speicherform heißt bei *IBM DB2 XML Extender* dann *XML Collection Storage*. Die SQL/XML-Norm definiert eine Pseudofunktion *XMLTable*, die virtuelle SQL-Tabellen aus XML-Werten produziert. XMLTable ist für Oracle verfügbar. Die Syntax lautet:

```
XMLTABLE ( [ namespace-declaration , ] xquery-expression
  [ PASSING { BY REF | BY VALUE } argument-list ]
  COLUMNS xmltbl-column-definitions )
```

xquery-expression ist ein Ausdruck genau wie bei XMLQuery oben, d.h. als Literal zu spezifizieren Die verwendete *Argumentliste* übergibt die aktuellen Parameter per Referenz oder als kopierte Werte.[1]

Der COLUMNS-Teil ist eine gewöhnliche SQL-Spaltendefinition mit Spaltenbezeichner, Typ und Pfadangabe für die Auswahl der Werte aus dem Ergebnis der XQuery-Abfrage.

Zusätzlich ist es möglich, zur Erhaltung der Ordnung im Dokument (Relationen sind Mengen, demnach sind die Tupel in Tabellen eigentlich ungeordnet) eine Spalte für Ordinalzahlen einzuführen, in die der Positionswert des XML-Werts (was XQuery liefert) im Dokument automatisch eingetragen wird. Die Syntax lautet:

```
column-name FOR ORDINALITY
```

Die Spalten der virtuellen Tabelle werden wie folgt generiert:

---

1. Laut [32] erfolgt die Übergabe nur per Referenz, das zugehörige Beispiel in [32] hat aber den Zusatz `PASSING :hv1 AS $dept BY VALUE`.

```
column-name data-type [ BY REF | BY VALUE ]
  [ default-clause ]
  [ PATH xquery-expression ]
```

Das folgende Beispiel stammt aus [32]:

```
INSERT INTO EMPS (ID, NAME, SAL)
 SELECT EMPNO, NAME, SALARY
 FROM XMLTABLE(
        'fn:doc(".../emps.xml")//emp'
        COLUMNS
          EMPNO INTEGER PATH 'badge'
          NAME VARCHAR(50) PATH 'name'
          SALARY DECIMAL(8,2) PATH 'comp/salary'
      )
```

Damit beenden wir die Besprechung von SQL/XML.

9 XPointer und XLink

Die XML Linking Language (XLink) erlaubt die Beschreibung von einfachen unidirektionalen Verknüpfungsstrukturen, analog zu den einfachen Hyperlinks in HTML heute, als auch komplexe Links. Die XML Pointer Language (XPointer) kann interne Strukturen eines Dokuments als Teil einer URI-Referenz adressieren. XPointer und XLink werden in den W3C Dokumenten [13, 14] beschrieben. Der Text hier folgt dem Tutorial von Melonfire [38].

Der große Erfolg des Web liegt in der Fähigkeit, Seiten mittels Verweisen (Links) verknüpfen zu können. Das Konzept aus den Sechziger Jahren geht auf *Ted Nelson*, den Erfinder der Begriffe *Hypertext* und *Hypermedia*, zurück. In HTML ist das Prinzip durch den Anker-Tag realisiert:

```
<a href="...">Dies ist ein Verweis</a>
```

Der HTML-Link hat allerdings eine Reihe von Nachteilen:

- jeder Link verweist von einer Ausgangsstelle zu genau einer Zielstelle; Mehrfachziele sind nicht möglich.
- der Verweis geschieht immer mit dem vordefinierten A-Tag.
- die Link-Definition (wohin der Link zeigen soll) steht im Quelldokument, für das man Schreibberechtigung braucht.

Die Anforderungen an XLink verlangen nun, dass

- Verweise wohlgeformte XML-Konstrukte sind
- lesbaren Klartext darstellen

- mit Informationen über Art, Titel, Ziel(e) und ggf. einem Verhalten bei Verfolgen des Links versehen sind
- Mehrfachziele unterstützen
- ohne Schreibrechte – weder auf Quell- noch Zieldokument – definierbar sind
- beliebige Traversierungsrichtungen erlauben
- kompatibel mit HTML sind.

9.1 Das XLink-Konzept

XLink bricht die Angabe von Verweisen in drei Teile auf:

- die „Link Definition“ zur Angabe der Verbindungsart
- die „teilnehmenden Resourcen“ (items) die XLink verbindet; diese können lokal (im selben Dokument) oder in einem fremden Dokument sein.
- „Traversierungsregeln“ oder „Pfeile (arcs)“, die angeben, in welche Richtung traversiert wird: „outbound“ (lokal nach entfernt), „inbound“ (entfernt nach lokal) oder „third-party“ (entfernte Resource nach entfernter Resource).

Ein Beispiel:

```
<?xml version="1.0"?>
<performers xmlns:xlink="http://www.w3.org/1999/xlink">
 <item xlink:type="extended">

   <!-- link definition (local) -->
   <link xlink:type="resource" xlink:label="overview"
         xlink:title="Information on Sinatra">
     Frank Sinatra
   </link>

   <!-- link definitions (remote) - Sinatra's biography,
        songs and articles -->
   <link xlink:type="locator" xlink:href="bio.xml"
         xlink:label="bio" xlink:title="Biography"/>
   <link xlink:type="locator" xlink:href="songs.xml"
         xlink:label="songs" xlink:title="Songs"/>
```

```
   <link xlink:type="locator" xlink:href="press.xml"
         xlink:label="press" xlink:title="Press articles"/>

   <!-- local to remote arc - from name to biography -->
   <arc xlink:type="arc" xlink:from="overview"
        xlink:to="bio" xlink:show="replace"
        xlink:actuate="onRequest"/>

   <!-- remote to remote arc - from biography to song list
        -->
   <arc xlink:type="arc" xlink:from="bio" xlink:to="songs"
        xlink:show="replace" xlink:actuate="onRequest"/>

   <!-- remote to remote arc - from biography to press
        archive -->
   <arc xlink:type="arc" xlink:from="bio" xlink:to="press"
        xlink:show="replace" xlink:actuate="onRequest"/>

  </item>
</performers>
```

Das Beispiel zeigt konzeptuelle Verweise zwischen einem Künstler (Frank Sinatra), seiner Biographie, seinen Liedern und einem Pressearchiv. Die Pfeile (arcs) geben die Traversierungsrichtungen an: vom Künstler zu seiner Biographie, von der Biographie zur Liste der Lieder und von der Biographie zum Archiv der Presseausschnitte.

Man beachte, dass Verweise nicht als Elemente, sondern als *Attribute* (aus dem XLink-Namensraum) in beliebigen Elementen angegeben werden. Das wichtigste Attribut hat den Namen `type` mit der offensichtlichen Bedeutung. Im Beispiel oben treten vier Typen auf: *extended links*, *resources*, *locators* und *arcs*. Sie erscheinen in den Elementen `item`, `link` und `arc`.

Je nach Wert des XLink-Attributs `type` werden für Zusatzangaben eines oder mehrere Attribute hinzugefügt. Im Beispiel hat der Verweis vom Typ `locator` das zusätzliche Attribut `href`, wohingegen XLinks für den Typ `arc` die Attribute `from` und `to` besitzen können.

9.2 Einfache und erweiterte Links

XLink kennt zwei Grundarten von Verweisen: einfache und erweiterte (simple and extended). Die einfachen sind an die gewohnten Anker-Tags angelehnt mit klarer Verweisrichtung.

Beispiel:

```
<?xml version="1.0"?>
<performers xmlns:xlink="http://www.w3.org/1999/xlink">
  <item xlink:type="simple" xlink:href="bio.xml">
      Sinatra's biography</item>
  <item xlink:type="simple" xlink:href="songs.xml">
      Song list</item>
  <item xlink:type="simple" xlink:href="press.xml">
      Press clippings</item>
</performers>
```

Die gleichen Verweise in einer HTML-Seite sähen wie folgt aus:

```
<html>
  <head>
  </head>
  <body>
    <a href="bio.xml">Sinatra's biography</a>
    <a href="songs.xml">Song list</a>
    <a href="press.xml">Press clippings</a>
  </body>
</html>
```

Erweiterte Links sind dagegen etwas ganz anderes und werden oft außerhalb der Dokumente gespeichert, die sie verbinden.

Im folgenden Beispiel werden Beziehungen zwischen Werdegang, Gehalt und einer Beurteilung eines Angestellten modelliert.

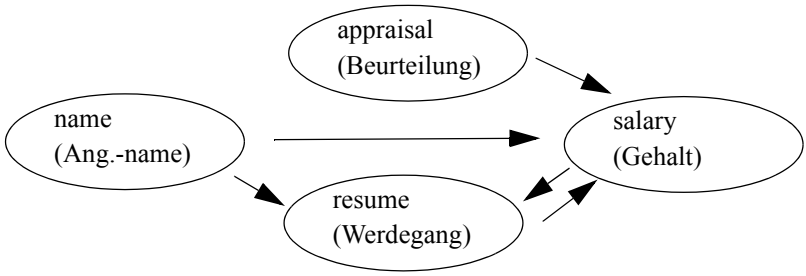


Abb. 9–1 *Beziehungen im Beispiel (nach [38])*

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

<ext xmlns:xlink="http://www.w3.org/1999/xlink"
     xlink:type="extended">

  <!-- starting point - employee's name (local) -->
  <local xlink:type="resource" xlink:label="name"
         xlink:title="John Doe"/>

  <!-- employee's salary (remote) -->
  <remote xlink:type="locator" xlink:href="salary.xml"
          xlink:label="salary"
          xlink:title="John Doe's salary information"/>

  <!-- employee's last appraisal (remote) -->
  <remote xlink:type="locator" xlink:href="appraisal.xml"
      xlink:label="appraisal"
      xlink:title="John Doe's last performance appraisal"/>

  <!-- employee's resume (remote) -->
  <remote xlink:type="locator" xlink:href="resume.xml"
          xlink:label="resume"
          xlink:title="John Doe's resume"/>

  <!-- link name to resume  -->
  <arc xlink:type="arc" xlink:from="name" xlink:to="resume"
       xlink:show="replace" xlink:actuate="onRequest"
       xlink:title="Link from name to resume"/>
```

```
 <!-- link name to current salary -->
 <arc xlink:type="arc" xlink:from="name" xlink:to="salary"
      xlink:show="replace" xlink:actuate="onRequest"
      xlink:title="Link from name to salary"/>

 <!-- link last appraisal to current salary-->
 <arc xlink:type="arc" xlink:from="appraisal"
      xlink:to="salary" xlink:show="replace"
      xlink:actuate="onRequest"
      xlink:title="Link from appraisal to salary"/>

 <!-- these next two arcs set up a bidirectional link -->
 <!-- link qualifications to salary -->
 <arc xlink:type="arc" xlink:from="resume"
      xlink:to="salary" xlink:show="replace"
      xlink:actuate="onRequest"
      xlink:title="Link from resume to salary"/>

 <!-- link salary to qualifications -->
 <arc xlink:type="arc" xlink:from="salary"
      xlink:to="resume" xlink:show="replace"
      xlink:actuate="onRequest"
      xlink:title="Link from salary to resume"/>

</ext>
```

Ein erweiterter Verweis wird demnach als XML-Element realisiert, das als Inhalt weitere XLink-Definitionen über teilnehmende lokale und entfernte Ressourcen und Pfeile enthält.

```
<ext xmlns:xlink="http://www.w3.org/1999/xlink"
     xlink:type="extended">
  <!-- participating resources -->
</ext>
```

Die geschachtelten Elemente können vier Linktypen darstellen:

- *Resource*. Dieser Linktyp stellt eine lokale Resource dar, die an dem erweiterten Link teilnimmt. Üblicherweise enthält sie Inhalt, der als Startpunkt für die Traversierung dient.

  ```
  <local xlink:type="resource" xlink:label="name"
         xlink:title="John Doe"/>
  ```

- *Locator*. Dieser Linktyp stellt eine entfernte Resource dar (mit zusätzlichem "`href`" Attribut), die an dem erweiterten Link teilnimmt.

```
<remote xlink:type="locator" xlink:href="salary.xml"
        xlink:label="salary"
        xlink:title="John Doe's salary information"/>
```

- *Arc*. Ein Pfeil gibt die Navigationsregeln zwischen „locators“ und „resources“ an (mittels zusätzlichen "`from`" und "`to`" Attributen); „arc“ ist der Hauptkonstrukt zur Richtungs- und Verhaltensangabe der Verweistraversierung.

```
<arc xlink:type="arc"
     xlink:from="name" xlink:to="salary"
     xlink:show="replace" xlink:actuate="onRequest"
     xlink:title="Link from name to salary"/>
```

  „Arcs“ sind ganz praktisch, wenn es gilt, einen bi-direktionalen Link einzurichten, z. B. $A \rightarrow B$ und $B \rightarrow A$.

```
<arc xlink:type="arc"
     xlink:from="resume" xlink:to="salary"
     xlink:show="replace" xlink:actuate="onRequest"
     xlink:title="Link from resume to salary"/>

<arc xlink:type="arc"
     xlink:from="salary" xlink:to="resume"
     xlink:show="replace" xlink:actuate="onRequest"
     xlink:title="Link from salary to resume"/>
```

  Allerdings sagt die XLink-Spezifikation ganz klar, dass Pfeile nicht dupliziert werden können, d. h. eine „Von-nach-Beziehung“ kann nur von genau einem Pfeil repräsentiert werden.

- *Title*. Dieser Linktyp liefert zusätzliche, lesbare Informationen über teilnehmende Ressourcen und kann als Alternative zum XLink-Attribut „title“ bei Elementen der Typen „extended“, „locator“ und „arc“ verwendet werden. Dies kann z. B. im Rahmen der Internationalisierung von Dokumenten genutzt werden.

```
<ext xmlns:xlink="http://www.w3.org/1999/xlink"
     xlink:type="extended">
  <greeting xlink:type="title"
            xml:lang="en">Hello!</greeting>
  <greeting xlink:type="title"
            xml:lang="fr">Bon jour!</greeting>
  <greeting xlink:type="title"
            xml:lang="it">Ciao!</greeting>
  ...
</ext>
```

9.3 Regeln für den Gebrauch der Attribute in XLink

Jedes Element, das einen XLink-Verweis definiert, *muss* ein `type`-Attribut enthalten.[1] Gültige Werte dafür sind: `simple`, `extended`, `locator`, `arc`, `resource`, `title` oder `none`.

Daneben sind neun zusätzliche Attribute möglich:

- Das `href`-Attribut zur Angabe einer URL einer entfernten Resource; Pflicht für `locator`-Verweise. Zusätzlich zur URL der Resource kann ein „fragment identifier“ angegeben werden, der auf eine spezielle Stelle (z. B. angegeben durch ein `id`-Attribut) im Zieldokument hinzeigt (siehe Besprechung zu XPointer).
- Das `show`-Attribut gibt an, wie das Ziel des Links präsentiert wird. Mögliche Werte sind `new` (Anzeige im neuen Fenster); `replace` (zeigt referenzierte Ressource im aktuellen Fenster, das gelöscht wurde); `embed` (Anzeige in einem Teilbereich des gegenwärtigen Fensters); `other` (Anzeige muss durch anwendungsspezifische Angabe erfolgen) oder `none` (keine Anzeigeangabe).
- Das `actuate`-Attribut[2] wird benutzt um anzugeben, wann ein Link traversiert werden soll: `onLoad` (Anzeige der Resource, auf die der Verweis zeigt, sobald vollständig geladen wurde); `onRequest` (Anzeige nur wenn ausdrücklich verlangt, z. B. durch Mausklick); `other` und `none`.

1. Ab XLink 1.1 ist das `type`-Attribut bei einfachen Links optional.
2. *to actuate*: in Gang bringen, auslösen

- Das `label`-Attribut wird benutzt, damit man den Link später bei einem Pfeil identifizieren kann.
- Die `from`- und `to`-Attribute geben den Start- und Endpunkt eines Pfeils an. Sie nutzen die `label`-Attribute in den Links zur Identifizierung dieser Punkte.
- Die `role`- und `arcrole`-Attribute referenzieren ein Dokument (URL), das Informationen über die Rolle und den Zweck des Verweises enthält (vgl. Rollen von Beziehungen im ER-Modell). Eine besondere Bedeutung hat das `arcrole`-Attribut in Verbindung mit einer `linkbase` (siehe unten).
- Das `title`-Attribut, nicht zu verwechseln mit dem Attributwert `title` für das `type`-Attribut, dient einer Beschreibung im Klartext.

Nicht alle Attribute (außer `type` bei XLink 1.0) werden überall benötigt. Die Regeln lauten:

1. Ein einfacher Link kann die Attribute `href` und optional `show` und `actuate` enthalten.
2. Einfache und erweiterte Links können optional `role`- und `title`-Attribute enthalten, um den Zweck zu erläutern.
3. Ein `locator`-Link muss ein `href`-Attribut enthalten zur Angabe der URL der entfernten Resource. Der Verweis enthält meist auch das optionale `label`-Attribut als Identifikation für Pfeile (arcs).
4. Aus dem gleichen Grund enthält ein `resource`-Link meist ein `label`-Attribut.
5. Ein Pfeil (arc) enthält typischerweise die optionalen `from`- und `to`-Attribute, ferner optional `show` und `actuate`, sowie optional das `arcrole`-Attribute, wenn der Pfeil mit einer `linkbase` auftritt.

9.4 Beispiel

„Warum einfach, wenn es auch kompliziert geht“, lautet ein alter Spruch.

```
<item xmlns:xlink="http://www.w3.org/1999/xlink"
      xlink:type="simple" xlink:href="bio.xml">
    Sinatra's biography</item>
```

Als erweiterten Link mit gleicher Funktionalität erhält man:

```
<item xmlns:xlink="http://www.w3.org/1999/xlink"
      xlink:type="extended">

  <!-- name -->
  <loc xlink:type="resource" xlink:label="local"
       xlink:title="Frank Sinatra">Frank Sinatra</loc>

  <!-- bio -->
  <rem xlink:type="locator" xlink:href="bio.xml"
       xlink:label="remote" xlink:title="Biography"/>

  <!-- local to remote arc - from name to biography -->
  <arc xlink:type="arc" xlink:from="local"
       xlink:to="remote"/>

</item>
```

Man beachte also, dass ein einfacher Link sowohl das entfernte Ziel benennt, als auch die unidirektionale Traversierung. Ein Lokator (locator) gibt dagegen nur das Ziel (die entfernte Resource) an und überlässt die Traversierung dem Pfeil (arc).

9.5 Linkbase

Der Linkbase-Konstrukt ist eine Datenbasis von Links, d. h. es geht um die Möglichkeit, mehrere Verweisziele in einem separaten XML-Dokument zusammenzufassen und vom Quelldokument einen Verweis auf diese Datei zu setzen.

Das folgende Beispiel verknüpft ein Land (USA) mit seinen Bundesstaaten.

```
<ext xmlns:xlink="http://www.w3.org/1999/xlink"
     xlink:type="extended">

  <!-- start from here -->
  <link xlink:type="resource" xlink:label="country"
        xlink:title="Country">United States</link>

  <!-- linkbase containing links to states -->
  <directory xlink:type="locator" xlink:href="states.xml"
```

```
            xlink:label="states"
            xlink:title="States within country"/>

 <!-- special linkbase arc with arcrole attribute -->
 <arc xlink:type="arc" xlink:from="country"
      xlink:to="states"
      xlink:arcrole="http://www.w3.org/1999/
                     xlink/properties/linkbase"
      xlink:show="replace" xlink:actuate="onRequest"/>

</ext>
```

Ein XLink-Prozessor würde bei einem solchen Verweis die Linkbase aufsuchen (hier `states.xml`) und dann nacheinander die dort enthaltenen Verweise gemäß der dort abgelegten Traversierungsregeln verfolgen. Der Standard sieht auch die Verkettung von Linkbases vor. Die Anwendung sollte zirkuläre Verweisketten erkennen und entsprechend beachten.

9.6 XPointer

XPointer-Verweise [13, 40] auf Fragmente in entfernten Dokumenten sind wieder ähnlich den bekannten HTML-Ankern. Grundsätzlich gibt es zwei Formen: Kurzhand- und Schemaform.

- `document#id` („shorthand form“)
- `document#scheme(...)` („scheme-based form“)

Die Besprechung hier folgt der Beispielsammlung von `zvon.org` in [39].

Die *Kurzhand-Form* beruht auf dem ID-Mechanismus in XML. Das Ziel wird über den Wert eines ID-Attributs oder ID-Elements bestimmt.

XPointer: `b1`

```
<!DOCTYPE AAA
[
  <!ELEMENT AAA (BBB+)>
  <!ELEMENT BBB EMPTY>
  <!ATTLIST BBB id ID #REQUIRED>
]
>

<AAA>
   <BBB id="b1"/>
   <BBB id="b2"/>
</AAA>
```

Schemabasierte Formen kennen verschiedene Mechanismen zur Identifikation passender Dokumentteile. Das Beispiel unten verwendet das `element()`-Schema (vgl. W3C Recommendation [40]).

XPointer: `element(/1/1)`

```
<AAA>
  <BBB id="b1"/>
  <BBB id="b2"/>
</AAA>
```

9.6.1 Kurzform mit ID-Attribut

Das Element BBB hat ein `id`-Attribut, das als ID-Typ in einem XML-Schema definiert ist.

XPointer:

```
b1

Matches element with ID "b1".
```

```
<AAA
  xsi:noNamespaceSchemaLocation="schema.xsd"
  xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance">
  <BBB id="b1"/>
  <BBB id="b2"/>
</AAA>
```

XML Schema (schema.xsd):

```
<xsd:schema
  xmlns:xsd=
    "http://www.w3.org/2001/XMLSchema"
  version="1.0">

<xsd:element name="AAA">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="BBB"
                   maxOccurs="unbounded">
        <xsd:complexType>
           <xsd:attribute name="id"
                          type="xsd:ID"/>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

</xsd:schema>
```

9.6.2 Kurzform mit schemabestimmtem ID-Element

The Element `BBB` hat ein `id`-Element das vom ID-Typ ist, wie durch das XML Schema definiert.

XPointer:

```
b1

Matches element with ID "b1".
```

```
<AAA
  xsi:noNamespaceSchemaLocation="schema.xsd"
  xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance">
  <BBB><id>b1</id></BBB>
  <BBB><id>b2</id></BBB>
</AAA>
```

XML Schema (schema.xsd):

```
<xsd:schema
  xmlns:xsd=
    "http://www.w3.org/2001/XMLSchema"
  version="1.0">

<xsd:element name="AAA">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="BBB"
                   maxOccurs="unbounded">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="id"
                         type="xsd:ID"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

</xsd:schema>
```

9.6.3 Kurzform mit ID-Attribut und DTD-Vereinbarung

XPointer: `b2`

Matches element with ID "b2".

```
<!DOCTYPE AAA
[
  <!ELEMENT AAA (BBB+)>
  <!ELEMENT BBB EMPTY>
  <!ATTLIST BBB id ID #REQUIRED>
]
>

<AAA>
  <BBB id="b1"/>
  <BBB id="b2"/>
</AAA>
```

9.6.4 Schemabasierte Pointer: Child Sequence

Eine „child sequence“ ist ein absoluter Pfad zu dem ausgewählten Element. Der Pfad ist eine Folge von Schrägstrichen und Zahlen, beginnend mit „/“. Jede Zahl *n* bestimmt das *n*-te Kind des vorher bestimmten Elements.

XPointer: `element(/1)`

Corresponding XPath: `/*[1]`

Matches root element

```
<AAA>
  <BBB>
    <CCC/>
    <CCC/>
  </BBB>
  <BBB>
    <CCC/>
    <CCC/>
  </BBB>
</AAA>
```

XPointer:	`element(/1/2)`
Corresponding XPath:	`/*[1]/*[2]`

Matches the second child of the root element.

```
<AAA>
  <BBB>
    <CCC/>
    <CCC/>
  </BBB>
  <BBB>
    <CCC/>
    <CCC/>
  </BBB>
</AAA>
```

XPointer:	`element(/1/2/1)`
Corresponding XPath:	`/*[1]/*[2]/*[1]`

Matches the first child of the second child of the root element.

```
<AAA>
  <BBB>
    <CCC/>
    <CCC/>
  </BBB>
  <BBB>
    <CCC/>
    <CCC/>
  </BBB>
</AAA>
```

9.6.5 Elementauswahl über ID

Das Element `BBB` hat ein `id`-Attribut, das vom Typ ID ist, z. B. nach DTD-Angabe.

XPointer:
```
element(b1)
```

Corresponding XPath:
```
id('b1')

Matches element with ID "b1".
```

```
<!DOCTYPE AAA
[
  <!ELEMENT AAA (BBB+)>
  <!ELEMENT BBB EMPTY>
  <!ATTLIST BBB id ID #REQUIRED>
]
>

<AAA>
  <BBB id="b1"/>
  <BBB id="b2"/>
</AAA>
```

9.6.6 ID kombiniert mit „Child Sequence“

Die „child sequence“ beginnt bei dem Element, dessen ID-Wert angegeben wurde, nicht an der Wurzel.

XPointer:
```
element(b1/1)
```

Corresponding XPath:
```
id('b1')/*[1]

Matches the first child of the element with
ID "b1".
```

```
<!DOCTYPE AAA
[
  <!ELEMENT AAA (BBB+)>
  <!ELEMENT BBB (CCC+)>
  <!ATTLIST BBB id ID #REQUIRED>
  <!ELEMENT CCC EMPTY>
]
>

<AAA>
  <BBB id="b0">
    <CCC/>
    <CCC/>
  </BBB>
  <BBB id="b1">
    <CCC/>
    <CCC/>
  </BBB>
</AAA>
```

10 Document Object Model

Das *Document Object Model (DOM)* ist eine plattform- und sprachunabhängige, baumartige interne Darstellung eines XML-Dokuments. Der Zugriff auf ein Dokument wird durch die *DOM-Schnittstelle* festgelegt. DOM ist in sog. Levels unterteilt. DOM-Level 1 [15] behandelt die Darstellung von XML- und HTML-Dokumenten im Allgemeinen. DOM-Level 2 [16] enthält weitere Teile für Namensräume, CSS-Stylesheets und Ereignisunterstützung. Level 3 [17] bietet zusätzlich Laden, Speichern, Parsen, Validieren und Serialisieren von Dokumenten und den Einsatz von XPath.

Die DOM-Schnittstelle wird durch die IDL (*Interface Definition Language*) definiert. In ihr werden die Methoden beschrieben, die jedes *Interface* vorweisen muss, nicht aber die **Klassen**, die für eine bestimmte Programmiersprache implementiert werden sollen. Deshalb wird für jede Programmiersprache eine sog. *Sprachanbindung* festgelegt. Wir verwenden hier die Java-Sprachanbindung, die ebenfalls im Standard unter `http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/java-language-binding.html` zur Verfügung steht. Das Paket, das diese Schnittstelle definiert, ist `org.w3c.dom`.

10.1 Erzeugen eines DOM-Baums mit dem DOM-Parser

Bis einschließlich Level 2 legt die Schnittstelle nicht fest, wie man aus einem XML-Dokument einen DOM-Baum erzeugt. In der Regel ist dies die Aufgabe desjenigen, der die Schnittstelle für eine bestimmte Programmiersprache implementiert. Wir verwenden hier Xerces [19], der einen

sog. *DOM-Parser* zur Verfügung stellt. Im nächsten Beispiel lesen wir ein XML-Dokument aus einer Datei ein und erzeugen daraus ein `Document`-Objekt, welches das XML-Dokument darstellt:

```
import org.apache.xerces.parsers.DOMParser;
import org.w3c.dom.Document;

public class domTest {

  public void performDemo (String uri) {
    DOMParser parser = new DOMParser();
    try {
      parser.parse(uri);
      Document document = parser.getDocument();
    } catch (Exception e) {
      System.out.println("Error during parsing");
    }
  }

  public static void main(String[] args) {
    if (args.length != 1) {
      System.out.println("usage domTest uri");
      System.exit(0);
    }
    domTest domtest = new domTest();
    domtest.performDemo(args[0]);
    }

}
```

Den erzeugten DOM-Baum (gegeben durch das Objekt `Document`) können wir mithilfe der Methoden der DOM-Schnittstelle dynamisch verändern und schließlich wieder als (neues) XML-Dokument *serialisieren*.

10.2 Darstellung des Dokuments

DOM stellt ein Dokument durch eine Baumstruktur dar. Dabei wird jeder Knoten durch ein `Node`-Objekt repräsentiert, woraus speziellere Knotentypen abgeleitet werden können. Einige Knotenarten können weitere Unterknoten haben, andere sind Blattknoten. Wir geben hier alle Knotentypen mit ihren möglichen Unterknoten an:

- **`Document`**
  `Element` (Anzahl maximal eins), `ProcessingInstruction`, `Comment`, `DocumentType` (Anzahl maximal eins)
- **`DocumentFragment`**
  `Element`, `ProcessingInstruction`, `Comment`, `Text`, `CDATASection`, `EntityReference`
- **`DocumentType`**
  keine Sohnknoten
- **`EntityReference`**
  `Element`, `ProcessingInstruction`, `Comment`, `Text`, `CDATASection`, `EntityReference`
- **`Element`**
  `Element`, `Text`, `Comment`, `ProcessingInstruction`, `CDATASection`, `EntityReference`
- **`Attr`**
  `Text`, `EntityReference`
- **`ProcessingInstruction`**
  keine Sohnknoten
- **`Comment`**
  keine Sohnknoten
- **`Text`**
  keine Sohnknoten
- **`CDATASection`**
  keine Sohnknoten
- **`Entity`**
  `Element`, `ProcessingInstruction`, `Comment`, `Text`, `CDATASection`, `EntityReference`
- **`Notation`**
  keine Sohnknoten

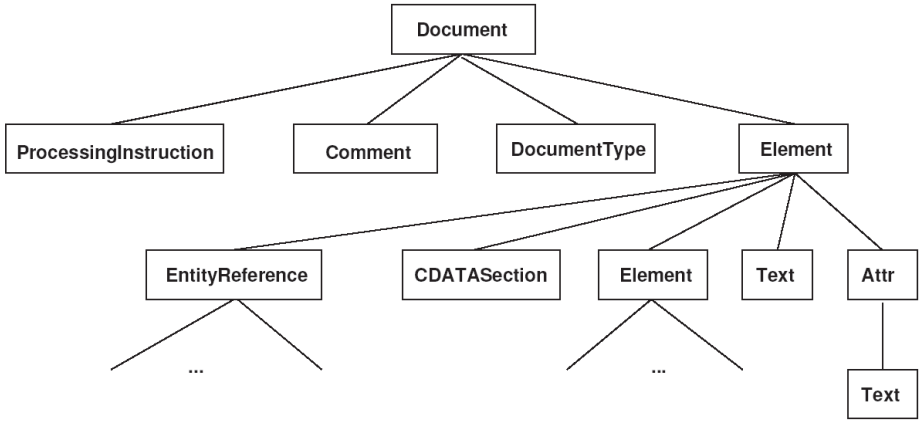


Abb. 10–1 *Ein XML-Dokument als DOM-Baum*

Jedes dieser Interfaces repräsentiert den entsprechenden Knotentyp aus dem XML-Dokument und stellt eine Liste von Methoden zur Verfügung, die für diesen Knoten sinnvoll sind. Z. B. liefert die Methode `getNodeValue()` für einen Text-Knoten das entsprechende Textelement als String zurück.

Weiterhin spezifiziert DOM die Interfaces `NodeList` und `NamedNodeMap`. `NodeList` stellt eine **geordnete Liste** von `Node`-Objekten dar. Zum Beispiel alle Sohnknoten eines Elements. `NamedNodeMap` stellt eine **ungeordnete Menge** von `Node`-Objekten dar, die über ihre Namen referenziert werden. Beispiel: alle Attribute eines Elements. Beide Listen sind sog. *live*-Objekte, das heißt, Änderungen an dem DOM-Baum werden ohne weitere Aktualisierung in den Listen reflektiert.

10.3 Beschreibung der Objekte und Methoden

Im Standard werden die Eigenschaften und Methoden aller DOM-Objekte beschrieben. Der Standard [15, 16] spricht dabei von DOM Core Interfaces. Wir geben hier eine Beschreibung der wichtigsten Teile. Weitere Details können dem Standard entnommen werden. Bei der Arbeit mit einer bestimmten Sprachschnittstelle wird man immer wieder auf deren Dokumentation zugreifen müssen, um Details wie Rückgabewert, Fehler-

fälle und Argumente zu bekommen. Für unsere Beispiele verwenden wir die Dokumentation von Xerces [20] auf der Seite `http://xml.apache.org/xerces2-j/javadocs/api/org/w3c/dom/package-summary.html`.

10.3.1 DOMException

Repräsentiert den Auftritt eines Fehlers. Dabei werden bestimmte Fehlercodes festgelegt. So signalisiert z. B. die Konstante `DOMException.INDEX_SIZE_ERR` die Verwendung eines nicht passenden Indexes.

10.3.2 DOMImplementation

Dieses Interface repräsentiert die Eigenschaften einer Implementierung und soll einer Anwendung ermöglichen, die Unterstützung bestimmter Funktionen abzufragen. Die Interface-Funktion enthält als einzige Methode:

```
boolean hasFeature(String feature, String version)
```

Zulässige Angaben für `feature` in DOM-Level 1 sind `XML` und `HTML` (Großschreibung!).

10.3.3 DocumentFragment

Dieses Interface dient zur Darstellung von Dokumentteilen. Nützlich z. B. für die Implementierung von Funktionen wie *Cut&Paste*. Wir gehen auf seine Methoden hier nicht ein.

10.3.4 Document

Dieses Interface repräsentiert das gesamte XML-Dokument. Da alle Knoten innerhalb eines `Document`-Nodes eine Bedeutung haben, deklariert dieses Interface die Methoden zum Erzeugen aller Knotentypen. Erzeugte Knoten können dann innerhalb anderer Knoten eingefügt werden (siehe weiter unten). `Document` deklariert die folgenden Methoden:

```
public DocumentType getDoctype()
public DOMImplementation getImplementation()
public Element getDocumentElement()
public Element createElement(String tagName)
public DocumentFragment createDocumentFragment()
public Text createTextNode(String data)
public Comment createComment(String data)
public CDATASection createCDATASection(String data)
public ProcessingInstruction
  createProcessingInstruction(String target, String data)
public Attr createAttribute(String name)
public EntityReference createEntityReference(String name)
public NodeList getElementsByTagName(String tagname)
```

Die Methode `getDocType()` liefert ein Objekt vom Typ `DocumentType`, das die `DOCTYPE`-Deklaration im Dokument darstellt. Fehlt diese Deklaration im Dokument so wird `null` zurückgeliefert. Die `create`-Methoden liefern eine Instanz des jeweiligen Knotentyps zurück. `getElementsByTagName()` liefert eine geordnete Liste aller Elemente im Dokument mit dem Namen `tagname`. `getDocumentElement()` liefert den Wurzelknoten als `Element`-Objekt zurück.

10.3.5 Node

Dieses Interface repräsentiert alle Knotentypen. Die folgenden Konstanten repräsentieren den jeweiligen Knotentyp eines `Node`-Objekts:

```
ELEMENT_NODE                = 1;
ATTRIBUTE_NODE              = 2;
TEXT_NODE                   = 3;
CDATA_SECTION_NODE          = 4;
ENTITY_REFERENCE_NODE       = 5;
ENTITY_NODE                 = 6;
PROCESSING_INSTRUCTION_NODE = 7;
COMMENT_NODE                = 8;
DOCUMENT_NODE               = 9;
DOCUMENT_TYPE_NODE          = 10;
DOCUMENT_FRAGMENT_NODE      = 11;
NOTATION_NODE               = 12;
```

`Node` deklariert die folgenden Methoden:

- **`public String getNodeName()`**
  liefert den Namen eines Knotens zurück.
- **`public String getNodeValue()`**
  liefert den Wert eines Knotens zurück (siehe weiter unten).
- **`public void setNodeValue(String nodeValue)`**
  setzt den Wert eines Knotens.
- **`public short getNodeType()`**
  liefert den Knotentyp zurück.
- **`public Node getParentNode()`**
  liefert einen Verweis auf den Vaterknoten.
- **`public NodeList getChildNodes()`**
  liefert eine geordnete Liste aller Unterknoten.
- **`public Node getFirstChild()`**
  liefert den ersten Unterknoten zurück.
- **`public Node getLastChild()`**
  liefert den letzten Unterknoten zurück.
- **`public Node getPreviousSibling()`**
  liefert den unmittelbar davorliegenden (Bruder-)Knoten zurück.
- **`public Node getNextSibling()`**
  liefert den unmittelbar nächsten (Bruder-)Knoten zurück.
- **`public NamedNodeMap getAttributes()`**
  liefert eine ungeordnete Liste aller Attribute.
- **`public Document getOwnerDocument()`**
  liefert einen Verweis auf das `Document`-Objekt des Dokuments zu dem `Node` gehört.
- **`public Node insertBefore(Node newChild, Node refChild)`**
  fügt `newChild` vor `refChild` ein.
- **`public Node replaceChild(Node newChild, Node oldChild)`**
  ersetzt `oldChild` durch `newChild`.
- **`public Node removeChild(Node oldChild)`**
  entfernt `oldChild`.
- **`public Node appendChild(Node newChild)`**
  fügt `newChild` als letzten Unterknoten ein.

- **`public boolean hasChildNodes()`**
  liefert `true`, falls `Node` Unterknoten hat.
- **`public Node cloneNode(boolean deep)`**
  erzeugt eine Kopie von `Node`.

Da `Node` verschiedene Knotentypen repräsentiert, hängt das Verhalten der Methoden vom Typ des Knotens ab. So erzeugen einige dieser Methoden sogar einen Fehler, falls sie bei dem jeweiligen Knoten keinen Sinn machen, z. B. der Aufruf der Methode `appendChild()` bei einem `Text`-Knoten. Andere Methoden liefern unterschiedliche Werte in Abhängigkeit des Knotentyps. Die Werte für `nodeName`, `nodeValue` und `attributes` (sowohl bei `get`- als auch `set`-Methoden) sind für einige Knotentypen in der folgenden Tabelle aufgelistet:

Interface	`nodeName`	`nodeValue`	`attributes`
`Attr`	Attributname	Attributwert	*`null`*
`CDATASection`	`#cdata-section`	Inhalt des CDATA-Abschnitts	*`null`*
`Comment`	`#comment`	Inhalt des Kommentars	*`null`*
`Document`	`#document`	*`null`*	*`null`*
`Element`	Elementname	*`null`*	Liefert `NamedNodeMap`
`Text`	`#text`	Textinhalt	*`null`*

10.3.6 Attr

`Attr` repräsentiert einen Attribut-Node. Zusätzlich zu den vererbten Methoden von `Node` werden die folgenden Methoden deklariert:

- **`String getName()`**
  liefert den Namen des Attributs zurück.
- **`boolean getSpecified()`**
  liefert `true`, falls das Attribut explizit im Dokument spezifiziert ist (kein Standardwert), sonst `false`.
- **`String getValue()`**
  liefert den Wert des Attributs zurück.
- **`void setValue(String value)`**
  setzt den Wert des Attributs auf `value`.

10.3.7 Element

`Element` repräsentiert einen Element-Node und stellt zusätzlich die folgenden Methoden zur Verfügung:

- **`String getTagName()`**
  liefert den Elementnamen zurück.
- **`String getAttribute(String name)`**
  liefert den Wert des Attributs mit dem Namen `name` zurück. `null`, falls nicht vorhanden.
- **`void setAttribute(String name, String value)`**
  setzt den Wert des Attributs `name` auf `value`. `name` wird neu erzeugt, falls es noch nicht existiert.
- **`void removeAttribute(String name)`**
  löscht das Attribut `name` aus der Attributliste des Elements.
- **`Attr getAttributeNode(String name)`**
  liefert ein `Attr`-Objekt für das Attribut `name` zurück. `null`, falls `name` nicht existiert.
- **`Attr setAttributeNode(Attr newAttr)`**
  fügt einen neuen Attribut-Node als Unterknoten des Elements ein.
- **`Attr removeAttributeNode(Attr oldAttr)`**
  entfernt den Attributknoten `oldAttr` aus der Liste der Attributknoten des Elements.

- **`NodeList getElementsByTagName(String name)`**
  liefert eine geordnete Liste aller Nachfahrenelemente mit dem Namen `name`.

10.3.8 Text und Comment

`Text` repräsentiert Zeichenketten innerhalb von XML-Dokumenten, sowohl als Inhalt von Elementen, als auch als Wert von Attributen. Falls der Text keinen weiteren Markup enthält, wird der gesamte Text als ein einziges `Text`-Objekt dargestellt. Andernfalls kann es auch benachbarte Textknoten geben, die sich mit `normalize` wieder verschmelzen lassen.

DOM definiert das Interface `CharacterData`. Es dient als eine Oberklasse für alle Objekte, die Zeichenketten repräsentieren. Die meisten Methoden von `Text` werden daher von `CharacterData` geerbt. Wir erwähnen hier die folgenden Methoden:

- **`String getData()`**
  liefert den Inhalt als `String` zurück.
- **`void setData(String data)`**
  setzt den Wert auf `data`.
- **`int getLength()`**
  liefert die Länge der Zeichenkette zurück.

Das Interface `Comment` wird ebenfalls von `CharacterData` abgeleitet. Es enthält keine zusätzlichen Methoden. Jedoch verhalten sich die `CharacterData`-Methoden bei Knoten vom Typ `Comment` naturgemäß anders als bei `Text`-Knoten (siehe oben).

10.3.9 NodeList

`NodeList` dient zur Aufnahme von mehreren Knoten in einer geordneten Liste. Dadurch können z. B. alle Kindknoten eines Elements in einer Schleife durchlaufen werden. Z. B. liefert die Methode `Node.getChildNodes()` alle Unterknoten eines beliebigen Knotens als ein `NodeList`-Objekt zurück.

`NodeList` stellt die folgenden Methode zur Verfügung:

- **`Node item(int i)`**
  liefert den i-ten Eintrag der Liste als `Node` zurück.
- **`int getLength()`**
  liefert die Anzahl der Einträge der Liste zurück.

Durch die beiden Methoden kann man die Liste in einer Schleife durchlaufen und die einzelnen Einträge bearbeiten (siehe Beispiel weiter unten).

10.3.10 NamedNodeMap

`NamedNodeMap` stellt eine **ungeordnete Kollektion** von `Node`-Objekten dar, die über ihre Namen referenziert werden, z. B. alle Attribute eines Elements. Es bietet die folgenden Methoden:

- **`Node getNamedItem(String name)`**
  liefert den Knoten mit dem Namen `name` zurück.
- **`Node setNamedItem(Node node)`**
  fügt den neuen Knoten `node` zu der Kollektion hinzu.
- **`Node removeNamedItem(String name)`**
  entfernt den Knoten mit dem Namen `name` aus der Kollektion.
- **`Node item(int i)`**
  liefert den i-ten Eintrag der Kollektion als `Node` zurück.
- **`int getLength()`**
  liefert die Anzahl der Einträge der Kollektion zurück.

10.4 Beispiel: Serialisieren eines DOM-Baumes

Im folgenden Beispiel entwickeln wir die Klasse `DOMSerializer`, die einen DOM-Baum als XML-Dokument serialisiert.[1] Dabei behandeln wir nur Element-, Text- und Attributknoten. Kommentare, Prozessoranweisungen usw. werden ignoriert.

1. Natürlich stellen Xerces und Xalan solche Klassen zur Verfügung, die eine viel schönere Ausgabe produzieren.

```
class DOMSerializer {

  private int indent;
  private Node rootNode;

  /** Konstuktor */
  public DOMSerializer(Document doc) {
    indent = 0;
    rootNode = (Node) doc;
  }

  /**
  * Schreibt eine Zeile in OutputStream. Davor werden
  * 3*indent leere Zeichen geschrieben.
  */
  private void writeIndentLine(String line,PrintStream to){
    for (int i = 0; i < indent ; i++) {
      to.print("   ");
    }
    to.println(line);
  }

  /**
  * Erzeuge aus einem DOM-Baum eine (XML-)Textausgabe.
  */
  private void printNode(Node node, PrintStream to) {
    switch (node.getNodeType()) {
    case Node.DOCUMENT_NODE:
      Document doc = (Document) node;
      printNode(doc.getDocumentElement(), to);
      break;
    case Node.ELEMENT_NODE:
      String name = node.getNodeName();
      String elementStr = "<" + name;
      NamedNodeMap attrs = node.getAttributes();
      for(int i = 0; i < attrs.getLength();i++) {
        Node current = attrs.item(i);
        elementStr = elementStr + " " +
          current.getNodeName() + "=\"" +
          current.getNodeValue() + "\"";
      }
      elementStr = elementStr + ">";
      writeIndentLine(elementStr, to);

      NodeList children = node.getChildNodes();
```

```
    if(children != null) {
      indent++;
      for (int i = 0; i < children.getLength(); i++) {
        printNode(children.item(i), to);
      }
      indent--;
    }
    writeIndentLine("</" + name + ">", to);
    break;
   case Node.TEXT_NODE:
    writeIndentLine(node.getNodeValue(), to);
    break;
   case Node.PROCESSING_INSTRUCTION_NODE:
    break;
   case Node.ENTITY_REFERENCE_NODE:
    break;
   case Node.DOCUMENT_TYPE_NODE:
    break;
   }
  }

  public void serialize(PrintStream to) {
   printNode(rootNode, to);
  }

}
```

Zum Testen dieser Klasse schreiben wir ein einfaches Programm, das ein XML-Dokument aus einer Datei in einen DOM-Baum umwandelt und dieses schließlich (mit einfacher Formatierung) auf der Standardausgabe ausgibt:

```
import java.io.PrintStream;
import org.w3c.dom.Document;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import org.w3c.dom.NamedNodeMap;
import org.apache.xerces.parsers.DOMParser;

class DOMSerializer { ... }

public class XMLWriter {

  private Document xmltoDom (String uri) {
   DOMParser parser = new DOMParser();
```

```
    Document res = null;
    try {
      parser.parse(uri);
      res = parser.getDocument();
    } catch (Exception e) {
      System.out.println("Error during parsing");
    }
    return res;
  }

  public void printOut(String uri) {
    Document xmlDom = xmltoDom(uri);
    DOMSerializer serializer = new DOMSerializer(xmlDom);
    serializer.serialize(System.out);
  }

  public static void main(String[] args) {
    if (args.length == 1) {
      XMLWriter xmlW = new XMLWriter();
      xmlW.printOut(args[0]);
    } else {
      System.out.println("usage XMLWriter uri");
      System.exit(1);
    }
  }

}
```

10.5 Beispiel: Erzeugen eines DOM-Baumes

In diesem Beispiel erzeugen wir aus einer internen Datenstruktur `Mitarbeiter` einen neuen DOM-Baum, und dann serialisieren wir diesen DOM-Baum in einer externen Datei.

```
import java.io.File;
import java.io.FileOutputStream;
import java.io.PrintStream;
import java.io.IOException;
import org.w3c.dom.Document;
import org.w3c.dom.Node;
import org.w3c.dom.Element;
import org.w3c.dom.Attr;
import org.w3c.dom.NodeList;
import org.w3c.dom.NamedNodeMap;
```

```
import org.apache.xerces.dom.DocumentImpl;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.OutputKeys;
import javax.xml.transform.stream.StreamResult;
import javax.xml.transform.Transformer;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.TransformerException;
import
 javax.xml.transform.TransformerConfigurationException;
import javax.xml.transform.TransformerException;

public class XMLServerSim {

 private static final String SHARE_FILE_NAME =
  "share.xml";

 // Wir verwenden die etwas bessere Serialisierungsmethode
 private void printNode (Node node, PrintStream to)
  throws IOException, TransformerConfigurationException,
         TransformerException {
  TransformerFactory tfactory =
   TransformerFactory.newInstance();
  Transformer serializer = tfactory.newTransformer();
  serializer.setOutputProperty(OutputKeys.METHOD,
                               "xml");
  serializer.setOutputProperty(OutputKeys.INDENT,
                               "yes");
  serializer.setOutputProperty(OutputKeys.ENCODING,
                               "ISO-8859-1");
  serializer.transform(new DOMSource(node),
                       new StreamResult(to));
 }

 public void start() {
  Mitarbeiter k1 = new Mitarbeiter ("1", "Ahmad",
                                    "Morad");
  Mitarbeiter k2 = new Mitarbeiter ("2", "Wegner",
                                    "Lutz");
  Mitarbeiter k3 = new Mitarbeiter ("3", "Fisher",
                                    "Burkhardt");
  try {
   File outputFile = new File(SHARE_FILE_NAME);
   PrintStream to =
    new PrintStream(new FileOutputStream(outputFile));
   Document mitarbDom = new DocumentImpl();
```

```
    Element mitarbeiter =
      mitarbDom.createElement("mitarbeiter");
    mitarbeiter.appendChild(k1.asNode(mitarbDom));
    mitarbeiter.appendChild(k2.asNode(mitarbDom));
    mitarbeiter.appendChild(k3.asNode(mitarbDom));
    mitarbDom.appendChild(mitarbeiter);
    printNode((Node) mitarbDom, to);
    to.close();
   }
   catch (Exception ex) {
    ex.printStackTrace();
   }
  }

  public static void main(String[] args) {
   XMLServerSim server = new XMLServerSim();
   server.start();
  }

}
```

Die Klasse `Mitarbeiter` könnte wie folgt aussehen:

```
import org.w3c.dom.Document;
import org.w3c.dom.Element;

public class Mitarbeiter {

  private String id, name, vorname;

  public Mitarbeiter (String id, String name, String
                      vorname) {
   this.id = id;
   this.name = name;
   this.vorname = vorname;
  }

  public Element asNode (Document doc) {
   Element mitarb = doc.createElement("mitarb");
   Element newElement = null;
   newElement = doc.createElement("id");
   newElement.appendChild(doc.createTextNode(this.id));
   mitarb.appendChild(newElement);
   newElement = doc.createElement("name");
   newElement.appendChild(
    doc.createTextNode(this.name));
```

```
    mitarb.appendChild(newElement);
    newElement = doc.createElement("vorname");
    newElement.appendChild(
      doc.createTextNode(this.vorname));
    mitarb.appendChild(newElement);
    return mitarb;
  }

}
```

Das erzeugte XML-Dokument sieht dann wie folgt aus (siehe `printNode()`):

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<mitarbeiter>
  <mitarb>
    <id>1</id>
    <name>Ahmad</name>
    <vorname>Morad</vorname>
  </mitarb>
  <mitarb>
    <id>2</id>
    <name>Wegner</name>
    <vorname>Lutz</vorname>
  </mitarb>
  <mitarb>
    <id>3</id>
    <name>Fisher</name>
    <vorname>Burkhardt</vorname>
  </mitarb>
</mitarbeiter>
```

11 SAX

SAX (*Simple API for XML*) ist eine XML-Schnittstelle, die eine sequentielle Abarbeitung eines Dokuments ermöglicht. Die Abarbeitung erfolgt ohne dass zunächst das ganze Dokument in den Hauptspeicher eingelesen wird. Für viele Aufgaben ist dieser Ansatz effizienter als die DOM-Schnittstelle, die immer das ganze Dokument vor der Verarbeitung lädt.

Im Gegensatz zur DOM-Schnittstelle ist SAX durch die Zusammenarbeit mehrerer Entwickler aus der Open-Source-Gemeinde entstanden. Trotzdem gilt SAX als Quasi-Standard und wird von den meisten Parsern unterstützt. Die offizielle Seite für den SAX-Standard ist: `http://sax.sourceforge.net` [43]. Dort findet man ebenfalls die Java-Dokumentation für SAX. Ein gängiges Textbuch stammt von David Brownell [42].

Das Hauptprinzip von SAX ist das Auffassen eines XML-Dokuments als Folge von *Ereignissen*. Der Anwendungsentwickler implementiert für diese Ereignisse die entsprechenden *Callback-Methoden* und erreicht damit, dass der Parser diese Methoden beim Eintritt dieser Ereignisse aufruft.

Um zu verstehen, wie ein XML-Dokument aus Sicht der SAX-Schnittstelle wirkt, betrachten wir das folgende Dokument:

```
<?xml version="1.0"?>
<doc>
  <para>Hello, world!</para>
</doc>
```

SAX stellt das Dokument als Folge von Ereignissen wie folgt dar:

```
start document
start element: doc
start element: para
characters: Hello, world!
end element: para
end element: doc
end document
```

Zur Implementierung der Ereignismethoden ist jeweils ein sog. *Handler* zu implementieren. Ein Handler ist eine Menge von *Callbacks*, die der SAX-Parser bei dem Eintritt von bestimmten Ereignissen aufruft.

Der SAX-2.0-Standard definiert vier verschiedene Handler-Interfaces: `ContentHandler`, `ErrorHandler` (Fehlerbehandlung), `DTDHandler` (Meldungen über Notationen und nicht-geparste Entitäten) und `EntityResolver` (Auflösung von externen Entities). Wir beschränken uns hier auf die beiden wichtigeren Handler: `ContentHandler` und `ErrorHandler`.

In der Regel entwickelt man eine Klasse, die eine Methode aus diesen Handler-Interfaces **implementiert** nach dem Muster:

```
class MyContentHandler implements ContentHandler {...}
```

Man **registriert** dann diese Handler-Implementierung bei dem SAX-Parser mit den Methoden `setContentHandler()`, `setErrorHandler()` usw. und erreicht damit, dass die implementierten Methoden beim Eintritt von bestimmten Ereignissen vom SAXParser aufgerufen werden.

Das Verwenden eines SAX-Parsers geschieht in der Regel nach dem folgenden Muster:

```
// Implementieren der Handler-Klassen (Content, Error etc.)
class MyContentHandler implements ContentHandler {...}
...

// Erzeugen des SAX-Parsers
XMLReader xr = new SAXParser();

// Erzeugen des ContentHandlers
MyContentHandler conHandler = new MyContentHandler();

// Erzeugen des ErrorHandlers
```

```
MyErrorHandler errHandler = new MyErrorHandler();

// Registrieren der Handler bei dem SAX-Parser
xr.setContentHandler(conHandler);
xr.setErrorHandler(errHandler);

// Den SAX-Parser aufrufen
xr.parse(new InputSource(dateiname));
...
```

11.1 ContentHandler

Das Interface `ContentHandler` deklariert alle Methoden, die ein SAX-Parser während des normalen Parsens aufruft. Das Interface deklariert die folgenden Methoden, die beim Auftritt der passenden Ereignisse aufgerufen werden:

- **`void startDocument()`**
  Beginn des Dokuments. Diese Methode wird als erste vor allen anderen Methoden aufgerufen.
- **`void endDocument()`**
  Das Ende des Dokuments. Wird als letzte Methode nach dem Abschluss des Parsens aufgerufen.
- **`void startElement(String uri, String localName, String qName, Attributes atts)`**
  Beginn eines Elements. Diese Methode wird aufgerufen, wenn der Parser mit der Verarbeitung eines Elements beginnt. Dabei bezeichnet `uri` den Namensraum, zu dem das Element gehört (eventuell leerer String), `localName` den eigentlichen Elementnamen, `qName` den vollen Namen (`namespace:localName`) des Elements und `atts` eine Liste der Attribute des Elements.
- **`void endElement(String uri, String localName, String qName)`**
  Wird aufgerufen beim Ende eines Elements.
- **`void characters(char[] ch, int start, int length)`**
  Lesen von Zeichenketten. Wird aufgerufen, wenn der Parser eine Zeichenkette liest. In dem Feld `ch[]` wird die Zeichenkette gespeichert, die den gesamten Inhalt des Vaterknotens enthält. `start` bezeichnet

den Anfang des Strings in diesem Feld, `length` die Länge der Zeichenkette.

Aus diesen drei Argumenten kann man ganz einfach ein String-Objekt wie folgt erzeugen:

```
String s = new String(ch, start, length);
```

- **`void ignorableWhitespace(char[] ch, int start, int length)`**
  Liefert Zeichenketten, die aus sog. Whitespaces bestehen, falls man diese Zeichen braucht. In der Regel werden sie jedoch ignoriert.
- **`void processingInstruction(String target, String data)`**
  Auftritt einer Prozessorinstruktion. `target` enthält den Namen der Prozessorinstruktion, `data` enthält die weiteren Daten der Prozessorinstruktion. Für die XML-Deklaration `<?xml version='...'?>` wird diese Methode **nicht** aufgerufen, da es sich hierbei nicht um eine Prozessorinstruktion handelt.
- **`void setDocumentLocator(Locator locator)`**
  Registriert ein Objekt zur Ermittlung der Position des Parsers im Dokument (welche Zeile, welches Zeichen). Meistens verwendet man diese Methode, um eine lokale Referenz auf den `Locator` zu speichern.
- **`void skippedEntity(String name)`**
  Wenn der Parser Entities ignoriert, wird diese Methode aufgerufen. Bei Xerces ist dies z. B. nicht der Fall.

Zusätzlich ist anzumerken, dass Xerces uns einen einfachen `DefaultHandler` zur Verfügung stellt, der die Methoden mit einem Standardverhalten implementiert. Wir können also diesen `DefaultHandler` als Basis verwenden und nur die Methoden überschreiben, die wir mit einem neuen Verhalten versehen wollen:

```
public class MyHandler extends DefaultHandler {...}
```

Im folgenden Beispiel geben wir die auftretenden Ereignisse eines XML-Dokuments aus:

```
import java.io.FileReader;
import org.xml.sax.XMLReader;
import org.apache.xerces.parsers.SAXParser;
```

```
import org.xml.sax.Attributes;
import org.xml.sax.InputSource;
import org.xml.sax.helpers.XMLReaderFactory;
import org.xml.sax.helpers.DefaultHandler;

public class MySAXApp extends DefaultHandler {

  public MySAXApp () {
    super();
  }

  public void startDocument () {
    System.out.println("Start document");
  }

  public void endDocument () {
    System.out.println("End document");
  }

  public void startElement (String uri, String name,
                            String qName, Attributes atts){
    System.out.println("Start element: " + qName);
  }

  public void endElement (String uri, String name,
                          String qName) {
    System.out.println("End element: " + qName);
  }

  public void characters (char ch[], int start, int length){
    if (length > 0) {
      System.out.print("Characters: ");
      for (int i = start; i < start + length; i++) {
        System.out.print(ch[i]);
      }
      System.out.println();
    }
  }

  public static void main (String args[])
    throws Exception {
    // XMLReader xr = XMLReaderFactory.createXMLReader();
    XMLReader xr = new SAXParser();
    MySAXApp handler = new MySAXApp();
    xr.setContentHandler(handler);
```

```
   xr.setErrorHandler(handler);
   // Parse alle übergebenen Dateien
   for (int i = 0; i < args.length; i++) {
     FileReader r = new FileReader(args[i]);
     xr.parse(new InputSource(r));
   }
  }

}
```

Bemerkung: Wir haben einen Parser mit `XMLReader xr = new SAXParser();` direkt erzeugt und uns auf einen bestimmten SAXParser festgelegt. In der Regel kann man durch den Aufruf von `XMLReaderFactory.createXMLReader();` den verwendeten Parser vom Benutzer bzw. durch die Systemeinstellungen bestimmen lassen.

Wir betrachten als Beispiel das folgende XML-Dokument `hallo.xml`:

```
<?xml version='1.0' standalone='yes'?>
<greetings>
   <greeting lang="de">
      Hallo Welt!
   </greeting>
   <greeting lang="en">
      Hello World!
   </greeting>
</greetings>
```

Mit dem Aufruf `java MySAXApp hallo.xml` erhalten wir die Ausgabe:

```
Start document
Start element: greetings
Characters:

Start element: greeting
Characters:
      Hallo Welt!
Characters:

End element: greeting
Characters:

Start element: greeting
Characters:
      Hello World!
```

```
Characters:

End element: greeting
Characters:

End element: greetings
End document
```

Man sieht an diesem Beispiel ganz schön, dass Whitespace-Zeichen hier zum Inhalt der Elemente gehören. Sie können als „ignorable Whitespaces“ nur erkannt werden, wenn eine DTD oder ein XML-Schema eingebunden ist.

11.2 ErrorHandler

Das Interface `ErrorHandler` dient zur Implementierung von Callback-Methoden für die Fehlerbehandlung. Der SAX-Parser ruft dann die Methoden dieses Interfaces beim Eintritt der jeweiligen Fehler auf. Das Interface deklariert die folgenden Methoden, welche die drei Fehlerarten vom XML-Standard behandeln:

- **`void warning(SAXParseException exception)`**
  Wird aufgerufen bei Eintritt von Warnungen. Warnungen können in den im XML-Standard genannten Fällen ausgegeben werden, zum Beispiel wenn in der DTD Attribute für nicht deklarierte Elementtypen angegeben werden; dies ist jedoch kein Fehler.
- **`void error(SAXParseException exception)`**
  Wird aufgerufen bei Eintritt von „normalen“ (nicht kritischen) Fehlern. Ein nicht kritischer Fehler ist eine Verletzung der Regeln der XML-Spezifikation, wobei die Konsequenzen nicht definiert sind.
  Zum Beispiel ist es ein Fehler, wenn die Validierung aktiviert ist und das Dokument nicht der angegebenen Grammatik genügt.
- **`void fatalError(SAXParseException exception)`**
  Wird aufgerufen beim Eintritt von „kritischen“ Fehlern. Kritische Fehler sind solche, die den Parsing-Prozess aufhalten. Meistens ist dies der Fall, wenn das Dokument nicht wohlgeformt ist.

Das übergebene Argument vom Typ `SAXParseException` enthält alle Informationen (Zeilennummer, Fehlermeldung etc.) über den Fehler und kann diese mit den entsprechenden Methoden zurückliefern.

In dem folgenden Beispiel geben wir einige Informationen über jeden Fehler oder jede Warnung aus:

```
class MyErrorHandler implements ErrorHandler {

  public void warning(SAXParseException exception)
    throws SAXException {
    System.out.println("**Warnung**\n" +
      " Zeile: " + exception.getLineNumber() + "\n" +
      " Meldung: " + exception.getMessage());
    throw new SAXException("Warnung");
  }

  public void error(SAXParseException exception)
    throws SAXException {
    System.out.println("**nicht kritischer Fehler**\n" +
      " Zeile: " + exception.getLineNumber() + "\n" +
      " Meldung: " + exception.getMessage());
    throw new SAXException("nicht kritischer Fehler");
  }

  public void fatalError(SAXParseException exception)
    throws SAXException {
    System.out.println("**kritischer Fehler**\n" +
      " Zeile: " + exception.getLineNumber() + "\n" +
      " Meldung: " + exception.getMessage());
    throw new SAXException("kritischer Fehler");
  }

}
```

Wir registrieren diesen `ErrorHandler` bei dem SAX-Parser aus dem obigen Beispiel mit:

```
MyErrorHandler errHandler = new MyErrorHandler();
xr.setErrorHandler(errHandler);
```

Um einen Fehler zu produzieren, ändern wir z. B. den Tag-Namen `<greetings>` in `<greeting>` um und bekommen die Ausgabe:

```
Start document
```

```
Start element: greeting
Characters:

Start element: greeting
Characters:
      Hallo Welt!
Characters:

End element: greeting
Characters:

Start element: greeting
Characters:
      Hello World!
Characters:

End element: greeting
Characters:

**kritischer Fehler**
 Zeile: 9
 Meldung: The end-tag for element type "greeting" must end
with a '>' delimiter.
Exception in thread "main" org.xml.sax.SAXException:
kritischer Fehler
 at MyErrorHandler.fatalError(MySAXApp.java:36)
 at org.apache.xerces.util.ErrorHandlerWrapper.fatalError
 at org.apache.xerces.impl.XMLErrorReporter.reportError
 at org.apache.xerces.impl.XMLErrorReporter.reportError
 at org.apache.xerces.impl.XMLScanner.reportFatalError
 at org.apache.xerces.impl.XMLNSDocumentScannerImpl.
    scanEndElement
 at org.apache.xerces.impl.XMLDocumentFragmentScannerImpl
    $FragmentContentDispatcher.dispatch
 at org.apache.xerces.impl.XMLDocumentFragmentScannerImpl.
    scanDocument
 at org.apache.xerces.parsers.XML11Configuration.parse
 at org.apache.xerces.parsers.XML11Configuration.parse
 at org.apache.xerces.parsers.XMLParser.parse
 at org.apache.xerces.parsers.AbstractSAXParser.parse
 at MySAXApp.main(MySAXApp.java:88)
```

11.3 Beispiel: Counter

Das folgende Beispiel `Counter` hat seinen Ursprung in einem Java-Beispielprogramm zur Xerces-Bibliothek. Das Programm bekommt ein XML-Dokument als Argument, parst dieses und gibt eventuelle Fehlermeldungen aus. Bei Erfolg werden Anzahl der Elemente, Zeichen und Attribute sowie der Zeitaufwand ausgegeben.

Die auf einem DOM-Parser basierende Variante dieses Beispiels wird in den Übungen unter dem Alias `xmlParse` zur Prüfung von XML-Dokumenten eingesetzt.

```
import java.io.PrintWriter;
import org.xml.sax.Attributes;
import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;
import org.xml.sax.XMLReader;
import org.xml.sax.helpers.XMLReaderFactory;
import org.xml.sax.helpers.DefaultHandler;

public class Counter extends DefaultHandler {

  /** Default-Parser*/
  protected static final String DEFAULT_PARSER_NAME =
            "org.apache.xerces.parsers.SAXParser";
  protected long fElements;
  protected long fAttributes;
  protected long fCharacters;
  protected long fIgnorableWhitespace;
  protected long fTagCharacters;

  /** Default constructor. */
  public Counter() {} // <init>()

/** Prints the results. */
  public void printResults(PrintWriter out, String uri,
                           long time) {
    out.print(uri);
    out.print(": ");
    out.print(time);
    out.print(" ms");
    out.print(" (");
    out.print(fElements);
    out.print(" elems, ");
```

```
  out.print(fAttributes);
  out.print(" attrs, ");
  out.print(fIgnorableWhitespace);
  out.print(" spaces, ");
  out.print(fCharacters);
  out.print(" chars)");
  out.println();
  out.flush();
 }

 public void startDocument() throws SAXException {
  fElements            = 0;
  fAttributes          = 0;
  fCharacters          = 0;
  fIgnorableWhitespace = 0;
  fTagCharacters       = 0;
 }

 public void startElement(String uri, String local,
                          String raw, Attributes attrs)
  throws SAXException {
  fElements++;
  fTagCharacters++; // open angle bracket
  fTagCharacters += raw.length();
  if (attrs != null) {
   int attrCount = attrs.getLength();
   fAttributes += attrCount;
   for (int i = 0; i < attrCount; i++) {
    fTagCharacters++; // space
    fTagCharacters += attrs.getQName(i).length();
    fTagCharacters++; // '='
    fTagCharacters++; // open quote
    fTagCharacters++; // close quote
   }
  }
  fTagCharacters++; // close angle bracket
 }

 public void characters(char ch[], int start, int length)
  throws SAXException {
  fCharacters += length;
 }

 public void ignorableWhitespace(char ch[],
                                 int start, int length)
```

```
  throws SAXException {
  fIgnorableWhitespace += length;
}

public void processingInstruction(String target,
                                  String data)
  throws SAXException {
  fTagCharacters += 2; // "<?"
  fTagCharacters += target.length();
  if (data != null && data.length() > 0) {
    fTagCharacters++; // space
  }
  fTagCharacters += 2; // "?>"
}

public void warning(SAXParseException ex)
  throws SAXException {
  printError("Warning", ex);
}

public void error(SAXParseException ex)
  throws SAXException {
  printError("Error", ex);
}

public void fatalError(SAXParseException ex)
  throws SAXException {
  printError("Fatal Error", ex);
}

protected void printError(String type,
                          SAXParseException ex) {
  System.err.print("[");
  System.err.print(type);
  System.err.print("] ");
  if (ex == null) {
    System.out.println("!!!");
  }
  String systemId = ex.getSystemId();
  if (systemId != null) {
    int index = systemId.lastIndexOf('/');
    if (index != -1)
      systemId = systemId.substring(index + 1);
    System.err.print(systemId);
  }
```

```
  System.err.print(':');
  System.err.print(ex.getLineNumber());
  System.err.print(':');
  System.err.print(ex.getColumnNumber());
  System.err.print(": ");
  System.err.print(ex.getMessage());
  System.err.println();
  System.err.flush();
 }

 public static void main(String argv[]) {
  if (argv.length != 1) {
   System.out.println("usage: Counter filename");
   System.exit(1);
  }
  String XMLFile = argv[0];
  Counter counter = new Counter();
  PrintWriter out = new PrintWriter(System.out);
  XMLReader parser = null;
  try {
   parser = XMLReaderFactory.createXMLReader(
                              DEFAULT_PARSER_NAME);
  } catch (Exception e) {
   System.err.println(
              "error: Unable to instantiate parser ("+
              DEFAULT_PARSER_NAME+")");
  }
  parser.setContentHandler(counter);
  parser.setErrorHandler(counter);
  try {
   long timeBefore = System.currentTimeMillis();
   parser.parse(XMLFile); // Parsen
   long timeAfter = System.currentTimeMillis();
   long time = timeAfter - timeBefore;
   counter.printResults(out, XMLFile, time);
  }
  catch (SAXParseException e) {
   // ignore
  }
  catch (Exception e) {
   System.err.println("error: Parse error occurred - "+
                      e.getMessage());
   Exception se = e;
   if (e instanceof SAXException) {
    se = ((SAXException)e).getException();
```

```
    }
    if (se != null)
      se.printStackTrace(System.err);
    else
      e.printStackTrace(System.err);
  }
 }

}
```

Mit dem Aufruf `java Counter greetings.xml` bekommen wir die Ausgabe:

```
greetings.xml: 234 ms (3 elems, 2 attrs, 9 spaces, 45 chars)
```

12 XML-RPC und SOAP

Die auch von uns formulierte Aussage, XML entwickle sich zum universellen Datenaustauschformat, eröffnet natürlich die Frage, inwieweit XML als Protokollsprache dienen kann. Tatsächlich gibt es mit XML-RPC (*Remote Procedure Call*) [45] einen XML-basierten Standard (allerdings nicht W3C unterstützt) für entfernte Prozeduraufrufe. Zu nennen sind ferner SOAP (*Simple Object Access Protocol*) [4, 46] als Teil einer größeren *Web Services* Initiative [44] innerhalb des W3C, die in Richtung Middleware geht.

Grundsätzlich ist *Remote Procedure Call* ein synchrones Kommunikations- und Prozesssynchronisierungskonzept, bei dem das wohlbekannte Prinzip der Prozedur- oder Funktionsaufrufe auf verteilte Umgebungen erweitert wurde. Der Aufrufer (Auftraggeber) ruft eine Prozedur (Dienstleistung) eines anderen Rechners auf und übergibt die Aufrufparameter. Der Aufrufer wartet dann auf die Beendigung des Unterauftrags. Bei Beendigung durch den Dienstgeber erhält er eine Bestätigung und ggf. Rückgabeargumente.

In realen Implementierungen, z.B. den bekannten UNIX RPC-Paketen als Teil der Interprozesskommunikation (IPC), wirken RPCs eher unhandlich, weil die Fragen der Bekanntmachung der Dienste und Adressen, die Probleme mit unterschiedlichen Datentypen (abhängig von den entsprechenden Wirtssprachen, häufig C) und Repräsentationen zwischen heterogenen Rechnerarchitekturen sowie die Fragen des Ausfalls eines Teilnehmers geregelt werden müssen.

Mit XML versucht der neue Standard[1] die Schwierigkeiten von RPC dadurch zu umgehen, dass die Nachrichten zwischen Aufrufer (Client) und Dienstanbieter (Server), die über HTTP verbunden sind, im XML-Format ausgetauscht werden. Das *Kodieren* und *Dekodieren* der Nachrichten kann somit von einem XML-Parser und einer XML-RPC-Schnittstelle übernommen werden.

Der XML-RPC-Standard wird von verschiedenen Unternehmen vorangetrieben, eine Beschreibung findet sich auf der Seite `http://www.xmlrpc.com`.

12.1 Der Standard

Der XML-RPC-Standard basiert auf dem HTTP-Protokoll zur Übertragung von Nachrichten. Das macht die Implementierung einfacher, da die meisten Programmiersprachen HTTP unterstützen. Zur Übertragung einer Nachricht wird die HTTP-POST-Methode verwendet. Eine XML-RPC-Nachricht sieht somit wie folgt aus:

```
POST /RPC2 HTTP/1.0
User-Agent: Frontier/5.1.2 (WinNT)
Host: betty.userland.com
Content-Type: text/xml
Content-length: 181

<?xml version='1.0'?>
<methodCall>
  ...
</methodCall>
```

Der erste Teil der Nachricht (bis zur leeren Zeile) ist der sog. Header einer HTTP-Nachricht. Er enthält Informationen über die Nachricht (Art, Datenformat, Länge etc.). Danach folgt die eigentliche Nachricht: das XML-RPC-Dokument.

XML-RPC ist ein sehr einfaches Protokoll. Es definiert eine XML-Sprache für Prozeduraufrufe und die Übermittlung von Rückgabewerten.

---

1. Wir verwenden hier den von Morad Ahmad eingeführten Begriff „Standard“, weisen aber nochmals darauf hin, dass nur SOAP ein W3C-Standard ist.

Der Server, der eine Nachricht erhält, analysiert mit Hilfe eines XML-Parsers einen Prozeduraufruf und ermittelt die eigentliche Methode, die aufgerufen werden soll. Falls eine solche Methode existiert, wird sie ausgeführt. Mittels der XML-Bibliothek werden die Rückgabewerte in das XML-RPC-Format gewandelt und an den Client übertragen. Auf der Clientseite läuft ein ähnlicher Vorgang ab.

12.1.1 Methodenaufrufe

Methodenaufrufe werden mit dem Element `<methodCall>` definiert. Das folgende Beispiel (Name eines US-Bundesstaates für eine gegebene Staatennummer liefern) zeigt einen Methodenaufruf in XML-RPC:

```
<?xml version='1.0'?>
<methodCall>
  <methodName>examples.getStateName</methodName>
  <params>
    <param>
      <value><i4>41</i4></value>
    </param>
  </params>
</methodCall>
```

Damit rufen wir die Methode `getStateName` des *Handlers* `examples` beim Server auf.

Parameter und Datentypen

Damit Parameter zwischen verschiedenen Programmiersprachen und Plattformen reibungslos ausgetauscht werden, legt XML-RPC die Parameterbeschreibung und die erlaubten Datentypen fest. Darüberhinaus werden die Eigenschaften eines jeden Datentyps beschrieben. Ein `Integer` kann z. B. in unterschiedlichen Programmiersprachen 8, 16 oder 32 Bit lang sein.

Das Element `<params>` dient zur Parameterbeschreibung der Methode. Jeder Parameter wird durch ein `<param>`-Element angegeben. Jedes `<param>`-Element enthält ein `<value>`-Unterelement, das wiederum ein Unterelement besitzt, das den Datentyp des Parameters beschreibt.

Mögliche Typen sind atomare Werte (zum Beispiel Zahl oder Zeichenkette), komplexe Datentypen (<struct>) und Felder (<array>). Ein **String** wird angenommen, falls kein Datentyp angegeben wird. XML-RPC definiert die folgenden atomaren Datentypen:

Element	Datentyp	Beispiel
<i4> oder <int>	4-Byte-Integer mit Vorzeichen	-12
<boolean>	0 (false) oder 1 (true)	1
<string>	String	Hallo
<double>	Double Fließkommazahl	-12.214
<dateTime.iso8601>	Datum/Zeit	19980717T14:08:55
<base64>	Base64-Kodierung	eW91IGNhbid0IHJ=

Weiterhin werden komplexe Datentypen unterstützt. Ein <struct> hat verschiedene <member>-Elemente, die diese Struktur beschreiben. Jedes <member>-Element besteht aus einem <name>- und einem <value>-Element. Werte können wiederum einfache oder komplexe Datentypen oder Felder sein. Das folgende Beispiel enthält eine Struktur mit zwei Elementen:

```
<struct>
  <member>
    <name>lowerBound</name>
    <value><i4>18</i4></value>
  </member>
  <member>
    <name>upperBound</name>
    <value><i4>139</i4></value>
  </member>
</struct>
```

Ein <array> dient zur Beschreibung eines Feldes. Es enthält eine Liste von Werten, wobei diese nicht unbedingt denselben Datentyp haben müssen. Die einzelnen Elemente werden über ihren **Index** angesprochen. Beispiel:

```
<array>
  <data>
    <value><i4>12</i4></value>
    <value><string>Egypt</string></value>
    <value><boolean>0</boolean></value>
    <value><i4>-31</i4></value>
  </data>
</array>
```

12.1.2 Rückgaben

Das Ergebnis der Ausführung einer Prozedur (oder Methode) wird in einem XML-Dokument an den Aufrufer (den Client) zurückgeliefert. Für die Beschreibung der Rückgabe dient das Element <methodResponse>. Das folgende Beispiel zeigt seinen Aufbau:

```
<?xml version="1.0"?>
<methodResponse>
  <params>
    <param>
      <value><string>Montana</string></value>
    </param>
  </params>
</methodResponse>
```

<methodResponse> kann ebenfalls Fehlermeldungen enthalten. Dazu dient das Element <fault>.

12.2 XmlRpc-Schnittstelle

Über die XmlRpc-Schnittstelle kann man einen sog. XmlRpc-Server und XmlRpc-Client entwickeln, die miteinander XML-RPC-Dokumente austauschen. Diese Schnittstelle verbirgt die technischen Details und ermöglicht damit einem Programmierer ohne XML-Kenntnisse, XML-RPC zu verwenden.

Auf der Clientseite kodiert die Schnittstelle die Aufrufe in der XML-RPC-Sprache und verschickt sie an den Server mittels des HTTP-Protokolls. Auf der Serverseite wird der XML-Text gefiltert und in die entsprechenden Methodenaufrufe umgewandelt. Die Resultate vom Server werden wiederum in XML-RPC-Dokumente umgewandelt und an den Client übertragen, wo sie wieder durch die XmlRpc-Schnittstelle in die interne Darstellung des Clients umgewandelt werden.

12.2.1 Installieren und Verwenden der XmlRpc-Schnittstelle

In unseren Beispielen greifen wir wieder auf die Implementierung der Apache-Gemeinde unter `ws.apache.org/xmlrpc` zurück.[1] Nachdem wir das Paket heruntergeladen haben, kopieren wir die `JAR`-Archive in ein beliebiges Verzeichnis und fügen sie der `CLASSPATH`-Variable hinzu. Alternativ können die Archive auch in das Verzeichnis `jre/lib/ext` unterhalb der Java-Installation kopiert werden. In diesem Fall ist die Anpassung der `CLASSPATH`-Variable nicht nötig.

In Java-Programmen können wir dann die Klassen des Pakets mit

```
import org.apache.xmlrpc.*;
```

einbinden. Einige dieser Klassen besprechen wir in diesem Kapitel.

Die Schnittstelle stellt unter anderem die Klassen `XmlRpcServer` und `XmlRpcClient` zur Verfügung, die zur Implementierung von Servern und Clients dienen. Weiterhin gibt es die Klasse `WebServer`, die einen einfachen Webserver implementiert (HTTP-Protokoll).

Für das Parsen der XML-Dokumente (ein- und ausgehende Nachrichten) muss ein SAX-Parser spezifiziert werden. Da die Schnittstelle keinen bestimmten festlegt, müssen wir auf Server- und Clientseite zuerst den SAX-Parser bestimmen:

```
XmlRpc.setDriver("org.apache.xerces.parsers.SAXParser");
```

1. Wir verwenden hier die Version 2 dieser Implementierung.

Serverimplementierung

Zum Erzeugen eines Servers können wir eine Instanz der Klasse `WebServer` erzeugen. Als Argument geben wir den Port `8081` an, auf dem der Server lauschen soll:

```
WebServer server = new WebServer(8081);
```

Als letztes können mehrere Handler registriert werden, so dass sie von Clients verwendet werden können:

```
server.addHandler("hello", new HelloHandler());
```

Dabei bezeichnet `hello` den Namen, mit dem die Clients diesen Handler ansprechen können. Zusätzlich kann man mit der Methode `acceptClient` bestimmen, welche Clients Kontakt mit diesem Server aufnehmen dürfen, worauf wir hier aber nicht im Detail eingehen. Schließlich muss der Server noch gestartet werden:

```
server.start();
```

Clientimplementierung

Die Entwicklung von Clients ist noch einfacher. Wir müssen zuerst einen Client mit der Klasse `XmlRpcClient` erzeugen:

```
XmlRpcClient client = new
  XmlRpcClient("http://localhost:8081");
```

Das Argument gibt den Servernamen und den Port an, auf dem der Server läuft. Danach können wir mit der Methode `execute(`*`method`*`, `*`args`*`)` Methoden bei dem entfernten Server ausführen. Die Rückgabewerte von `execute` sind die gelieferten Werte der Servermethoden.

12.3 Beispiel

In diesem Beispiel entwickeln wir einen einfachen XmlRpc-Server und -Client. Der Server bietet die einzige Methode `sayHello`, die von dem Handler `hello` implementiert wird. Der Client ruft diese Methode beim Starten mit dem ersten Programmargument auf.

12.3.1 HelloServer

```
import org.apache.xmlrpc.WebServer;
import org.apache.xmlrpc.XmlRpc;
import org.apache.xerces.parsers.SAXParser;

public class HelloServer {

  public static void main(String args[]) {
    try {
      XmlRpc.setDriver(
             "org.apache.xerces.parsers.SAXParser");
      // Erzeuge Server
      WebServer server = new WebServer(8081);
      // Registrieren Handler-Klassen
      server.addHandler("hello", new HelloHandler());
      System.out.println("Warte auf Anfragen ...");
      server.start();
    } catch(Exception ex) {
      ex.printStackTrace();
    }
  }
}
```

12.3.2 HelloHandler

```
public class HelloHandler {

  public String sayHello(String str) {
    return "Hello " + str;
  }

}
```

12.3.3 HelloClient

```
import java.util.Vector;
import org.apache.xmlrpc.XmlRpc;
import org.apache.xmlrpc.XmlRpcClient;
import org.apache.xerces.parsers.SAXParser;

public class HelloClient {
  public static void main(String args[]) {
    if (args.length != 1) {
```

```
    System.out.println("usage: java HelloClient name");
    System.exit(1);
   }
   try {
    XmlRpc.setDriver(
           "org.apache.xerces.parsers.SAXParser");
    XmlRpcClient client =
            new XmlRpcClient("http://localhost:8081");
    Vector params = new Vector();
    params.addElement(args[0]);
    String result = (String)
            client.execute("hello.sayHello", params);
    System.out.println("Antwort vom Server: " + result);
   }
   catch(Exception ex) {
    ex.printStackTrace();
   }
  }
}
```

Wir kompilieren die beiden Programme mit:

```
javac HelloServer.java
javac HelloClient.java
```

Server starten:

```
>java HelloServer &
Warte auf Anfragen ...
```

Client aufrufen:

```
>java HelloClient XML
Antwort vom Server: Hello XML
```

12.4 SOAP

SOAP ist die W3C-Version des Remote Procedure Calls. Nach Ansicht einiger der ehemaligen XML-RPC-Mitglieder enthält SOAP einige unnötig aufgeblähte Elemente mit seltsamen Features, andererseits scheint sich SOAP durchzusetzen.

SOAP, derzeit in Version 1.2 als W3C-Recommendation [46] verfügbar, unterstützt XML Schema, eine Kombination von *structs* und *arrays*

sowie benutzerdefinierte Typen. SOAP erlaubt keine DTDs. SOAP implementiert ein zustandsloses Nachrichtenaustausch-Protokoll.

12.4.1 Botschaftenstruktur

Eine SOAP-Botschaft ist ein gewöhnliches XML-Dokument mit den folgenden Elementen:

- ein `<Envelope>`-Element, welches das XML-Dokument als SOAP-Botschaft kennzeichnet (Umschlag, Wurzelelement).
- ein optionales `<Header>`-Element mit Header-Information, spez. für Transaktionen und Authentifizierung.
- ein `<Body>`-Element mit Aufruf- und Antwort-Informationen.
- ein optionales `<Fault>`-Element, das Informationen zu Fehlern enthält, die möglicherweise beim Nachrichtenaustausch aufgetreten sind.

Alle diese Elemente sind im Standardnamensraum für den SOAP *envelope* vereinbart:

```
http://www.w3.org/2003/05/soap-envelope
```

Der optionale Namensraum für die Kodierungen und Datentypen ist

```
http://www.w3.org/2003/05/soap-encoding
```

Für Fehlermeldungen bei Prozeduraufrufen wird der folgende Namensraum eingesetzt:

```
http://www.w3.org/2003/05/soap-rpc
```

Der grundsätzliche Aufbau ist demnach:

```
<?xml version="1.0"?>
<env:Envelope
  xmlns:env="http://www.w3.org/2003/05/soap-envelope">
 <env:Header>
   ...
 </env:Header>
 <env:Body>
   ...
```

```
   <env:Fault>
     ...
   </env:Fault>
   ...
  </env:Body>
</enc:Envelope>
```

12.4.2 Der SOAP-Kopf (Header)

Die Kindelemente des `<Header>`-Elements können

- ein `encodingStyle`-Attribut
- ein `role`-Attribut mit den Werten

  `http://www.w3.org/2003/05/soap-envelope/role/next`

  `http://www.w3.org/2003/05/soap-envelope/role/none`

  `http://www.w3.org/2003/05/soap-envelope/role/ultimateReceiver`
- ein `mustUnderstand`-Attribut mit den Werten true oder false
- ein `relay`-Attribut, auch mit true oder false

enthalten. Es folgt ein Beispiel mit zwei Elementen im Header.

```
<env:Envelope
   xmlns:env="http://www.w3.org/2003/05/soap-envelope">
  <env:Header>
    <n:ext xmlns:n="http://example.org/ext"
           env:mustUnderstand="true">
      <n:someint>1</n:someint>
    </n:ext>
    <m:ext2 xmlns:m="http://example.org/ext2"
      env:role
       ="http://www.w3.org/2003/05/soap-envelope/role/next"
      env:mustUnderstand="true">
      <m:someint>1</m:someint>
    </m:ext2>
  </env:Header>
  <env:Body>
    <my:message xmlns:my="http://www.mycompany.com/myns">
      <my:text>Hello World!</my:text>
    </my:message>
  </env:Body>
</env:Envelope>
```

12.4.3 Der SOAP-Rumpf (Body)

Der Body einer SOAP-Nachricht enthält die anwendungsspezifischen Anfragen, Übergabewerte und eventuelle Fehlermeldungen. Es wird empfohlen, dass das `env:Body`-Element mindestens ein Unterelement enthält und einen eigenen Namensraum eröffnet.

Die folgende Abfrage ermittelt einen Preis.

```
<?xml version="1.0"?>
<env:Envelope
     xmlns:env="http://www.w3.org/2003/05/soap-envelope">
  <env:Body>
    <m:GetPrice xmlns:m="http://www.w3schools.com/prices">
      <m:Item>Apples</m:Item>
    </m:GetPrice>
  </env:Body>
</env:Envelope>
```

Eine mögliche Antwort könnte wie folgt aussehen.

```
<?xml version="1.0"?>
<env:Envelope
     xmlns:env="http://www.w3.org/2003/05/soap-envelope">
  <env:Body>
    <m:GetPriceResponse
       xmlns:m="http://www.w3schools.com/prices">
      <m:Price>1.90</m:Price>
    </m:GetPriceResponse>
  </env:Body>
</env:Envelope>
```

12.4.4 Fehlerfälle

Das Body-Element kann ein Fault-Element enthalten, das über einen aufgetretenen Fehler Aufschluss gibt. Der Fehler kann beim Aufrufer entstanden sein (falsche Parameter, kein gültiges XML, usw.) oder der Empfänger des Aufrufs kann den RPC nicht verarbeiten, obwohl dieser gültig ist.

```
<?xml version='1.0'?>
<env:Envelope
     xmlns:env="http://www.w3.org/2003/05/soap-envelope"
```

```
     xmlns:rpc="http://www.w3.org/2003/05/soap-rpc">
 <env:Body>
  <env:Fault>
   <env:Code>
    <env:Value>env:Sender</env:Value>
    <env:Subcode>
     <env:Value>rpc:BadArguments</env:Value>
    </env:Subcode>
   </env:Code>
   <env:Reason>
    <env:Text xml:lang="en-US">Processing error</env:Text>
    <env:Text xml:lang="cs">Chyba zpracování</env:Text>
   </env:Reason>
   <env:Detail>
    <e:myFaultDetails
       xmlns:e="http://travelcompany.example.org/faults">
     <e:message>Name does not match card number</e:message>
     <e:errorcode>999</e:errorcode>
    </e:myFaultDetails>
   </env:Detail>
  </env:Fault>
 </env:Body>
</env:Envelope>
```

Das `env:Fault`-Element enthält immer zwei vorgeschriebene Unterelemente, `env:Code` und `env:Reason`, sowie optional anwendungsspezifische Informationen in einem `env:Detail`-Unterelement.

Das `env:Code`-Unterelement in `env:Fault` hat selbst wieder ein Pflichtelement, `env:Value`, dessen Inhalt in der SOAP-Spezifikation (siehe *SOAP Part 1 section 5.4.6*) angegeben ist. Im Beispiel oben wird mit „`env:Sender`“ der Aufrufer als Fehlerverursacher angegeben.

Zugleich enthält das Beispiel (aus dem SOAP Primer) ein `env:Subcode`, der angibt, dass ein RPC-spezifischer Fehler, hier `rpc:BadArguments`, definiert in *SOAP Part 2 section 4.4*, der Grund für die Nichtbearbeitung des Kundenwunsches ist.

12.4.5 Übertragungsprotokolle

Prinzipiell ist SOAP mit vielen Übertragungsprotokollen kombinierbar. Das Standardisierungsdokument zeigt die Nutzung im Rahmen von HTTP

GET und POST, nennt aber auch eine mögliche Nutzung über E-Mail mit SMTP. Auf die Details verzichten wir hier.

Literaturverzeichnis

[1] W3C: *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. W3C Recommendation, 26. November 2008, URL: `http://www.w3.org/TR/2008/REC-xml-20081126/`

[2] W3C: *Scalable Vector Graphics (SVG) 1.1 (Second Edition)*. W3C Recommendation, 16. August 2011, URL: `http://www.w3.org/TR/2011/REC-SVG11-20110816/`

[3] W3C: *W3C Math Home*. URL: `http://www.w3.org/Math`

[4] W3C: *Simple Object Access Protocol (SOAP)*. URL: `http://www.w3.org/TR/SOAP`

[5] SNELL, JAMES, DOUG TIDWELL, and PAVEL KULCHENKO: *Programming Web Services with SOAP*. O'Reilly & Associates, Inc., 2002.

[6] WINER, DAVE: *XML-RPC Specification*. URL: `http://www.xmlrpc.com/spec`, 1999.

[7] RAY, ERIK T.: *Einführung in XML*. O'Reilly & Associates, Inc., 2001.

[8] W3C: *XML Schema Part 0: Primer (Second Edition)*. W3C Recommendation, 28. Oktober 2004, URL: `http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/`

[9] W3C: *The Extensible Stylesheet Language (XSL)*. URL: `http://www.w3.org/Style/XSL`.

[10] W3C: *XSL Transformations (XSLT) Version 1.0*. W3C Recommendation, 16. November 1999, URL: `http://www.w3.org/TR/1999/REC-xslt-19991116`

[11] TIDWELL, DOUG: *XML-Dokumente transformieren mit XSLT*. O'Reilly & Associates, Inc., 2003.

[12] W3C: *XML Path Language (XPath) Version 1.0*. W3C Recommendation, 16. November 1999, URL: `http://www.w3.org/TR/1999/REC-xpath-19991116/`

[13] W3C: *XML Pointer Language (XPointer)*. W3C Working Draft, 16. August 2002, URL: `http://www.w3.org/TR/2002/WD-xptr-20020816/`

[14] W3C: *XML Linking Language (XLink) Version 1.1*. W3C Recommendation, 6. Mai 2010, URL: `http://www.w3.org/TR/2010/REC-xlink11-20100506/`

[15] W3C: *Document Object Model (DOM) Level 1 Specification (Second Edition) Version 1.0*. W3C Working Draft, 29. September 2000, URL: `http://www.w3.org/TR/2000/WD-DOM-Level-1-20000929/`

[16] *W3C: Document Object Model (DOM) Level 2 Core Specification Version 1.0*. W3C Recommendation, 13. November 2000, URL: `http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/`

[17] *W3C: Document Object Model (DOM) Level 3 Core Specification Version 1.0*. W3C Recommendation, 7. April 2004, URL: `http://www.w3.org/TR/DOM-Level-3-Core/`

[18] *XT*. URL: `http://www.blnz.com/xt/index.html`.

[19] *xerces*. URL: `http://xml.apache.org/xerces2-j/ index.html`.

[20] *xalan*. URL: `http://xml.apache.org/xalan-j/ index.html`.

[21] ECKSTEIN, ROBERT: *XML Pocket Reference*. O'Reilly & Associates, Inc., 1999.

[22] W3C: *XML Schema Part 1: Structures (Second Edition)*. W3C Recommendation, 28. Oktober 2004, URL: `http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/`

[23] W3C: *XML Schema Part 2: Datatypes (Second Edition)*. W3C Recommendation, 28. Oktober 2004, URL: `http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/`

[24] VLIST, ERIK VAN DER: *XML Schema*. O'Reilly & Associates, Inc., 2002.

[25] W3C: *XQuery 1.0: An XML Query Language*. W3C Recommendation, 23. Januar 2007, URL: `http://www.w3.org/TR/2007/REC-xquery-20070123/`

[26] W3C: *XML Path Language (XPath) 2.0 (Second Edition).* W3C Recommendation, 14. Dezember 2010, URL: `http://www.w3.org/TR/2010/REC-xpath20-20101214/`

[27] W3C: *XQuery 1.0 and XPath 2.0 Functions and Operators (Second Edition)*. W3C Recommendation, 14. Dezember 2010, URL: `http://www.w3.org/TR/2010/REC-xpath-functions-20101214/`

[28] HOMER, ALEX: *XML IE5 Programmer's Reference*. Wrox Press Ltd, Birmingham (UK), 1999

[29] WOLFGANG LEHNER und HARALD SCHÖNING: *XQuery - ein Überblick*, Datenbank-Spektrum 11/2004

[30] WOLFGANG LEHNER und HARALD SCHÖNING: *XQuery - Grundlagen und fortgeschrittene Methoden,* dpunkt-Verlag, Heidelberg, 2004

[31] HARALD SCHÖNING: *XML und Datenbanken*, Tutorium 2 der Datenbank-Tutorientage 2001, Deutsche Informatik-Akademie, Bonn.

[32] ANDREW EISENBERG and JIM MELTON: *Advancements in SQL/XML*, ACM SIGMOD Record, Vol. 33, No. 3, September 2004, pp. 79-86

[33] K. KÜSPERT, G. SAAKE und L. WEGNER: Duplicate Detection and Deletion in the Extended NF2 Data Model, Proc. Third Int. Conference on Foundations of Data Organization and Algorithms, Paris, Juni 1989, W.Litwin and H.J.Schek (Eds), Springer LNCS 367 (1989) 83-100

[34] G. SAAKE, V. LINNEMANN, P. PISTOR und L. WEGNER: Sorting, Grouping, and Duplicate Elimination in the Advanced Information Management Prototype, Proc. 15th Int. Conference on Very Large Data Bases, Amsterdam (Aug. 1989) 307-316

[35] J. E. FUNDERBURK, S. MALAIKA, and B. REINWALD: *XML programming with SQL/XML and XQuery*, IBM SYSTEMS JOURNAL, Vol. 14, No. 4 (2002) pp. 642-665

[36] CAN TÜRKER: SQL:1999 & SQL:2003 - Objektrelationales SQL, SQLJ & SQL/XML, dpunkt.verlag, 2003

[37] ISO/IEC 9075-14:2003 Information Technology - Database Languages - SQL - Part 14: XML-Related Specifications (SQL/XML)

[38] Tutorial: XLink Basics, `http://www.melonfire.com/community/columns/trog/article.php?id=90`

[39] Tutorial zu XPointer, `http://www.zvon.org/xxl/XPointerTutorial/Output/index.html`

[40] W3C: *XPointer element() Scheme*. W3C Recommendation, 25. März 2003, URL: `http://www.w3.org/TR/2003/REC-xptr-element-20030325/`

[41] W3C: *XML Protocol Working Group*, `http://www.w3.org/2000/xp/Group/`

[42] DAVID BROWNEL: *SAX2*, O'Reilly, 1st Edition 2002

[43] *About SAX: Simple API for XML*, http://sax.sourceforge.net/

[44] W3C: *Web Services Activity*, http://www.w3c.org/2002/ws/

[45] XML-RPC Home page, http://www.xmlrpc.com/

[46] W3C: *SOAP Version 1.2 Part 0: Primer (Second Edition).* W3C Recommendation, 27. April 2007, URL: http://www.w3.org/TR/2007/REC-soap12-part0-20070427/

[47] ANDREW EISENBERG und JIM MELTON: XQuery 1.0 is Nearing Completion, ACM SIGMOD Record, Vol. 34, No. 4 (Dec. 2005) 78-84

[48] JIM MELTON und STEPHEN BUXTON: Querying XML - XQuery, XPath, and SQL/XML in Context, Morgan Kaufmann 2006, 814 S.

[49] LUTZ WEGNER: Quicksort for Equal Keys, IEEE Trans. on Computers, Vol. 34, No. 4 (April 1985) 362-367

[50] JONATHAN ROBIE. *XQuery: A Guided Tour*, unter http://www.datadirect.com/developer/xquery/xquerybook/index.ssp.