daniel KOCH

XML
FÜR WEBENTWICKLER

EIN PRAKTISCHER EINSTIEG

HANSER

Koch

XML für Webentwickler

Daniel Koch

XML für Webentwickler

Ein praktischer Einstieg

HANSER

Der Autor:

Daniel Koch, Berlin

Bibliografische Information Der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

Gesamtlektorat: Fernando Schneider
Sprachlektorat: Sandra Gottmann, Münster-Nienberge
Herstellung: Stefanie König
Coverconcept: Marc Müller-Bremer, www.rebranding.de, München
Coverrealisierung: Stephan Rönigk
Datenbelichtung, Druck und Bindung: Kösel, Krugzell
Ausstattung patentrechtlich geschützt. Kösel FD 351, Patent-Nr. 0748702
Printed in Germany

ISBN 978-3-446-42256-8

www.hanser.de/computer

Inhalt

1 XML – ein Blick zurück und voraus

XML hat die (Online-)Welt verändert. Das zeigen die zahlreichen Technologien, die sich rund um XML etabliert haben. Denken Sie nur an AJAX, Newsfeeds oder ganz allgemein an das Web 2.0. All das wäre ohne XML weder denk- noch umsetzbar. Und XML kommt auch dort zum Einsatz, wo man es auf den ersten Blick gar nicht ahnen würde. Oder hätten Sie gedacht, dass sich das iPhone von Apple über eine einfache XML-Datei konfigurieren lässt? Es steckt also weitaus mehr hinter XML, als man gemeinhin vermutet.

Wie weit die Extensible Markup Language bereits in die Programmier- und Entwicklerwelt Einzug gehalten hat, ist vielen gar nicht bewusst. Und doch ist eine Welt ohne XML heute kaum noch vorstellbar. Denn diese Sprache dient mittlerweile als Format für Konfigurationsdateien, wird für den Im- und Export in den verschiedensten Programmen verwendet und kommt im Internet genauso wie innerhalb von Datenbanken zum Einsatz.

Um XML verstehen zu können, ist es hilfreich, wenn man weiß, woher die Sprache eigentlich kommt. Und genau hier setzt dieses Kapitel an: Es beschreibt, woher XML kommt und wo die Zukunft dieser Sprache liegt, stellt eine Auswahl XML-basierter Sprachen vor und zeigt, welche Werkzeuge bei der täglichen Arbeit mit XML hilfreich sind.

1.1 Die Idee hinter XML

Zunächst einmal ist die Frage interessant, wie und warum es eigentlich zur Entwicklung von XML gekommen ist. Dafür muss man verstehen, dass es sich bei XML nicht nur um eine Technologie handelt, sondern dahinter eine komplexe Idee steht. Als Format dient XML zunächst einmal dazu, Daten zu speichern und diese zwischen sehr unterschiedlichen Systemen austauschbar zu machen. Das ist allerdings nur der eine Aspekt. Der andere ist, dass sich um XML herum eine Reihe unterschiedlicher Standards entwickelt hat. Um einige dieser Sprachen kümmert sich das *World Wide Web Consortium* (W3C,) dessen Webseite unter *http://w3.org/* verfügbar ist.

Abbildung 1.1 Die „Verantwortlichen“ für XML

Dort finden Sie zum Beispiel die Sprachen SVG und SMIL. Mehr zu solchen „abgeleiteten“ Sprachen erfahren Sie im weiteren Verlauf dieses Kapitels, hier sei aber schon so viel verraten: Beides sind Sprachen, die im XML-Umfeld entstanden sind.

Eine der wichtigsten Fragen stellt sich unweigerlich jedem, der von XML hört: Wo liegen eigentlich die Vorteile dieser Sprache im Vergleich zu anderen Formaten? Hier hilft ein kurzer Ausflug in die Historie.

Die ersten elektronischen Formate kümmerten sich mehr darum, wie Daten ausgegeben werden sollten. Struktur und Sinn von Dokumenten spielten hingegen keine beziehungsweise nur eine untergeordnete Rolle. Klassische Formatierungssprachen dienten eher dazu, Dokumente anzusehen oder/und diese auszudrucken.

Diese Formate waren also eher für die Präsentation von Inhalten da. Die Datenstruktur blieb weitestgehend außen vor. Um dieses Problem zu lösen, wurde das Konzept der sogenannten generischen Kodierung entwickelt. Dabei kommen inhaltsorientierte Elemente zum Einsatz. Mit denen können bestimmte Inhalte mit verschiedenen Elementen logisch ausgezeichnet werden. Der erste große Schritt in diese Richtung war SGML (*Standard Generalized Markup Language*).

SGML wurde in den 70er- und 80er-Jahren entwickelt und während dieser Zeit hauptsächlich im Verlagswesen eingesetzt. Informationen zu SGML finden Sie unter anderem auf den Seiten des W3C unter *http://www.w3.org/MarkUp/SGML/*.

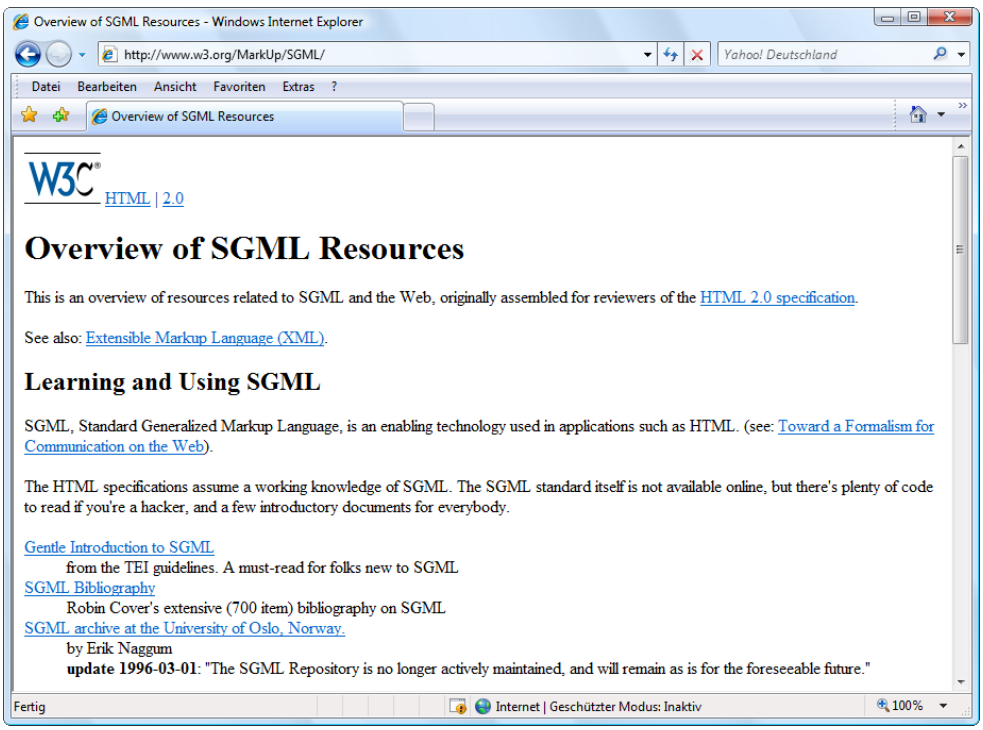

Abbildung 1.2 Das Zuhause von SGML

Genauso wie XML ist SGML im Grunde ein Werkzeug für die Entwicklung spezialisierter Auszeichnungssprachen. Allerdings ist der SGML-Sprachumfang wesentlich größer als der von XML und besitzt zudem eine deutlich strengere Syntax. Gleichzeitig ist SGML aber auch extrem flexibel. Das führt dazu, dass die Software für seine Verarbeitung komplex und somit auch entsprechend teuer ist. SGML wurde deswegen nur von großen Unternehmen und Organisationen eingesetzt, die sich die entsprechende Software für die SGML-Verarbeitung auch tatsächlich leisten konnten.

Der Durchbruch für die generische Kodierung kam in den frühen 90er-Jahren mit der *HyperText Markup Language*, kurz HTML. Bei HTML handelt es sich um einen SGML-Dokumenttyp für Hypertextdokumente. Die beiden größten Vorteile von HTML im Vergleich zu SGML sind:

- dass die Software, die für die Darstellung von HTML-Dokumenten benötigt wird, sehr einfach zu programmieren ist;
- dass sich HTML-Dokumente leicht kodieren lassen.

So rosig, wie das klingen mag, ist es dann allerdings doch nicht gewesen. Denn in gewisser Hinsicht war HTML wieder ein Schritt zurück. Um nämlich dieses hohe Maß an Unkompliziertheit erreichen zu können, mussten wichtige Prinzipien der generischen Kodierung geopfert werden. So ist zum Beispiel für sämtliche Einsatzzwecke lediglich ein einziger Dokumenttyp vorgesehen. Erschwerend kommt hinzu, dass viele der in HTML verwendeten Elemente ausschließlich der Präsentation von Daten dienen. Ein „schönes“ Beispiel dafür zeigt die folgende Syntax:

Listing 1.1 Vermischung von Daten und Präsentation

```
<font color="#999999" size="3">
   Ich bin HTML-Text
</font>
```

Elemente wie `font` & Co. dienen ausschließlich für die Präsentation von Daten. Mit dieser Verwässerung von HTML befand man sich also in einer Sackgasse. Um nun wieder zu den Idealen der generischen Codierung auch im WWW zurückzufinden, wurde versucht, SGML an das Web beziehungsweise das Web an SGML anzupassen. Diese Versuche scheiterten allerdings. Denn SGML ist schlichtweg zu umfangreich, um in einen WWW-Browser integriert werden zu können. Es musste also eine kleinere Sprache her, in der einerseits der verallgemeinernde Charakter von SGML enthalten blieb, die andererseits aber leicht zu erlernen war. Diese Überlegungen führten schlussendlich zur Entwicklung von XML.

1.2 Das Prinzip der Metasprachen

XML bietet die Möglichkeit, eigene Datenformate zu definieren. Diese folgen einem festgelegten Schema und garantieren somit Kompatibilität. Ebenso ermöglicht es XML, Sprachen zu erzeugen, die für eine umfangreiche Verbreitung, gerade auch im Softwareumfeld, geeignet sind. Für viele dieser Sprachen ist das W3C verantwortlich. Es gibt aber auch andere Institutionen und Unternehmen, die eigene XML-basierte Sprachen entwickeln und deren Verbreitung vorantreiben. Im folgenden Abschnitt werden, exemplarisch für die enorme Vielfalt der möglichen Anwendungen, einige Sprachen kurz vorgestellt.

Bei XML handelt es sich um eine Metasprache. Metasprache bedeutet in diesem Zusammenhang nichts anderes, als dass sich mit XML Untersprachen definieren lassen, die einen bestimmten Zweck erfüllen. Dokumente, die in einer dieser Untersprachen definiert werden, sind dann zwar immer noch XML-Dokumente, sie basieren allerdings nur noch auf dem eingeschränkten Befehlssprachschatz der Untersprache. Ausgedrückt beziehungsweise beschrieben werden diese Einschränkungen durch Schemasprachen wie DTD oder XML Schema. Das Grundprinzip von XML wird besonders deutlich, wenn man sich einmal den Namen Extensible Markup Language, für den die Abkürzung XML steht, „auf der Zunge zergehen lässt“. Ins Deutsche übersetzen lässt sich XML am besten mit *Erweiterbare Auszeichnungssprache*. Genau diese Erweiterbarkeit ist einer der wichtigsten Vorzüge von XML.

Wenn Sie sich einmal auf den Seiten des W3C umschauen, werden Ihnen dort zahlreiche XML-basierte Sprachen begegnen.

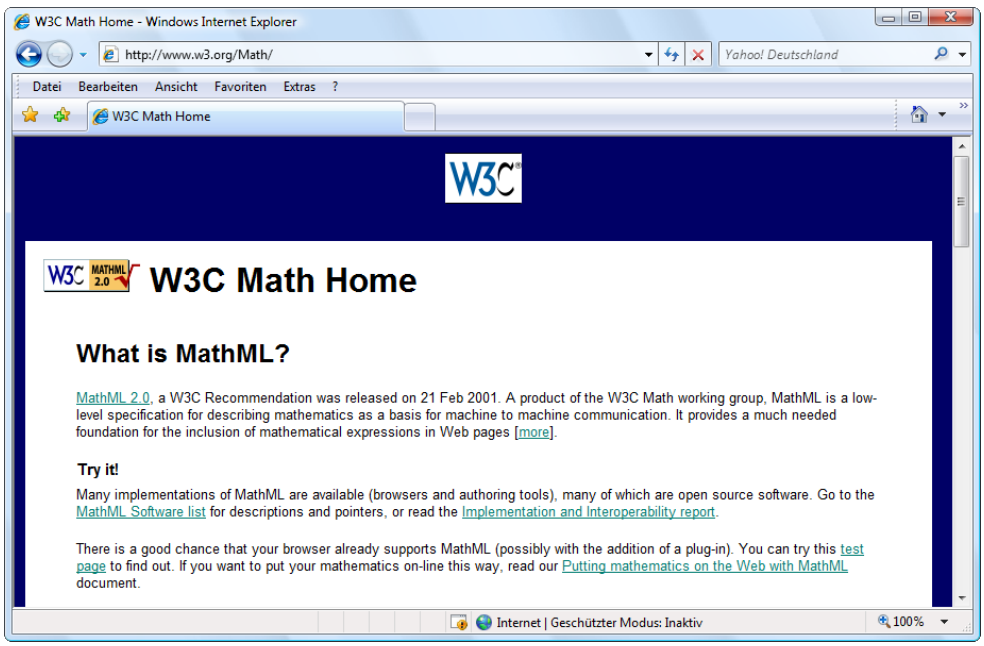

Abbildung 1.3 Eine typische XML-basierte Sprache ist MathML

Abbildung 1.3 zeigt die Startseite von MathML, einer Sprache zur Darstellung mathematischer Formeln und komplexer Ausdrücke. Wenn Sie sich regelmäßig auf den Seiten des W3C umsehen, werden Sie feststellen, dass immer wieder neue Sprachen hinzukommen.

Und noch etwas wird beim Blick auf die W3C-Seite deutlich: Das XML-Universum ist äußerst komplex. Dabei sind dort längst noch nicht alle XML-basierten Sprachen aufgeführt. So gibt es im XML-Umfeld eigentlich nichts, was es nicht gibt. Es existieren beispielsweise Sprachen für

- den Datenzugriff,
- die Ausgabe,
- die Strukturierung,
- die Kommunikation,
- die Formatierung und
- die Sicherheit.

Viele dieser Sprachen werden vom W3C standardisiert, andere Sprachen wiederum sind Eigenentwicklungen von Organisationen, Unternehmen oder sogar von Privatpersonen. Wie eingangs dieses Abschnitts versprochen, werden jetzt einige typische XML-basierte Sprachen kurz vorgestellt.

1.2.1 Multimedia ist Trumpf mit SMIL

Die *Synchronized Multimedia Language* (SMIL) ermöglicht die Präsentation multimedialer Anwendungen im Internet. Durch SMIL können hierbei andere Wege als bei bisherigen Anwendungen gegangen werden. Schließlich handelt es sich bei dieser Sprache um eine XML-Sprache, mit der sich multimediale Inhalte synchron wiedergeben lassen.

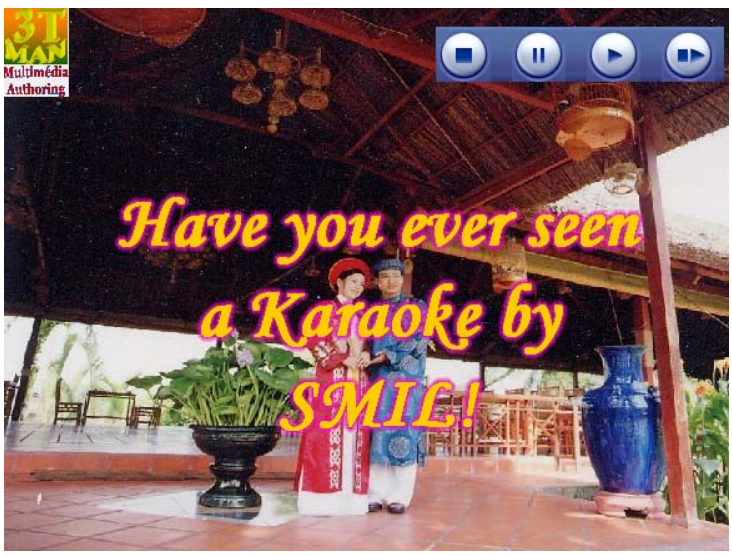

Abbildung 1.4 Eine kleine, aber feine SMIL-Anwendung

SMIL ist demnach für all die Bereiche interessant, in denen optisch ansprechende Inhalte präsentiert werden. Beispielhaft hierfür seien Messen, Kongresse und Meetings genannt. Weiterführende Informationen zu SMIL finden Sie auf den Seiten des W3C unter *http://www.w3.org/AudioVide/*.

Listing 1.2 Ein einfaches SMIL-Dokument

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE smil PUBLIC "-//W3C//DTD SMIL 2.0//EN"
                      "file:///Library/XML/smil/SMIL20.dtd">
<smil xmlns="http://www.w3.org/2001/SMIL20/Language">
   <head>
      <layout>
         <root-layout width="200" height="300" />
         <region id="red" top="10" />
         <region id="blue" top="50" />
      </layout>
   </head>
   <body>
      <img region="red" src="pic1.gif" dur="8" />
      <img region="blue" src="pic2.gif" dur="8s" />
   </body>
</smil>
```

Das Beispiel zeigt eine typische SMIL-Anwendung. Hier werden zwei Grafiken an unterschiedlichen Startpositionen angezeigt. Die Dauer der Anzeige wurde auf acht Sekunden begrenzt. Beachten Sie, dass die Betrachtung dieses Beispiels ein entsprechendes Werkzeug voraussetzt. Das kann unter anderem der RealPlayer sein.

1.2.2 Mit WML alles fürs Handy

Die *Wireless Markup Language* (WML) wurde speziell für die Darstellung von Inhalten auf mobilen Kleinstgeräten wie Handys usw. entwickelt. WML lehnt sich in seiner Syntax stark an die von XHTML an. Viele Elemente werden dem HTML-Entwickler bekannt sein. Dies macht es allen HTML-Erfahrenen leicht, ihre erste WML-Seite zu erstellen. Es gibt aber auch eine Vielzahl von Neuerungen. Als Beispiel seien hier die in WML üblichen `card`-Elemente genannt. Diese spezifizieren jeweils einen Display-Inhalt. Ein kleines Beispiel soll die Syntax von WML unter Verwendung des `card`-Elements veranschaulichen.

Listing 1.3 So gibt's was aufs Handy.

```
<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"
"http://www.wapforum.org/DTD/wml_1.1.xml">
<wml>
   <card>
      <p>
         <i>kursiver Text</i>
      </p>
   </card>
</wml>
```

Durch diese Syntax wird innerhalb eines Displays ein kursiver Text angezeigt. HTML-Entwicklern fällt sicherlich die syntaktische Nähe zu „ihrer" Sprache auf. WML bietet aber mehr als die bloße Darstellung von Texten; es können ebenso Grafiken und Formulare definiert werden. Wie die Entwicklung von WML weitergehen wird, lässt sich derzeit nicht voraussagen; zu schnelllebig ist die Technologie bei den mobilen Endgeräten. Allerdings sieht es momentan nicht sonderlich rosig für WML aus. Denn durch die Verbreitung von Smartphones wie BlackBerry und iPhone wächst die Anzahl der Geräte, die reguläre Webseiten problemlos darstellen können. Zudem fördert die Open Mobile Alliance mit XHTML Mobile Profile ein spezielles XHTML-Profil für mobile Geräte. Weiterführende Informationen zur WML finden Sie unter *http://www.wapforum.org/*.

1.2.3 Vektorgrafiken mit SVG

Im Jahr 2001 wurde die *Scalable Vector Graphics* (SVG) vom W3C als Standard verabschiedet. SVG ist eine XML-basierte Sprache zur Erstellung zweidimensionaler Vektorgrafiken. Im Gegensatz zu anderen Formaten wie Flash usw. liegen SVG-Dateien als ASCII-Code vor. Die Vorteile dieser Herangehensweise sind enorm: Nicht nur dass die Erstellung von SVG-Dateien keine besondere Software verlangt, auch die Größe der Dateien ist kleiner als die von Grafiken oder Flash-Animationen. Dieser Effekt kann durch GZIP noch verstärkt werden.

Listing 1.4 So einfach kommt man zur eigenen SVG-Grafik.

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20001102//EN"
"http://www.w3.org/TR/2000/CR-SVG-20001102/DTD/svg-20001102.dtd">
<svg xmlns="http://www.w3.org/2000/svg" width="300" height="200">
   <desc>
      Text in SVG platzieren
   </desc>
   <rect x="1" y="1" width="300" height="200"
         style="fill:none; stroke:black" />
   <circle cx="150" cy="100" r="90" style="fill:gray;" />
   <text x="50" y="40" style="font-family:verdana;
         font-size:21px; font-weight:bold;">
      Ein SVG-Beispiel
   </text>
   <text x="70" y="100" font-family="verdana"
         font-size="14px">
      einfache Anwendung
   </text>
</svg>
```

Auch SVG zeigt einige bekannte Elemente wie beispielsweise CSS-Anweisungen. Somit können HTML-erfahrende Entwickler leicht die ersten Anwendungen erstellen. SVG-Anwendungen können in HTML-Dateien eingebunden, aber auch als eigenständige Applikationen ausgeführt werden. Weiterführende Informationen zu SVG finden Sie unter *http://www.w3.org/Graphics/SVG/*.

1.2.4 Silverlight

Microsoft setzt in Silverlight ebenfalls auf XML, genauer gesagt auf die *eXtensible Application Markup Language* (XAML). Bei XAML handelt es sich um eine deklarative Sprache. Diese Sprache zeichnet all das aus, was auch XML auszeichnet. Darüber hinaus besitzt XAML aber auch ein spezielles Vokabular wie beispielsweise Canvas. Das sind Vokabulare, die in XAML definiert sind und die es so in XML nicht gibt.

Listing 1.5 Eine einfache Silverlight-Anwendung

```
<UserControl x:Class="SilverlightApplication5.Page"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Width="400" Height="300">
    <Grid x:Name="LayoutRoot"
       Background="White">
    </Grid>
</UserControl>
```

Zentrale Anlaufstelle für XAML ist die Seite *http://silverlight.net/*. Dort gibt es ausführliche Hinweise zu Silverlight und eben auch zu XAML.

1.3 Die Einsatzgebiete von XML

Ursprünglich wurde XML als ein Format entwickelt, mit dem sich strukturierte Inhalte im Internet austauschen lassen. Mittlerweile haben sich die Vorteile von XML allerdings so weit herumgesprochen, dass die Sprache auch in solchen Bereichen zum Einsatz kommt, an die man im ersten Moment vielleicht gar nicht denkt. Typische Beispiele hierfür sind das Dokumentmanagement und die Tatsache, dass XML mittlerweile auch als Austauschformat für Office-Programme verwendet wird.

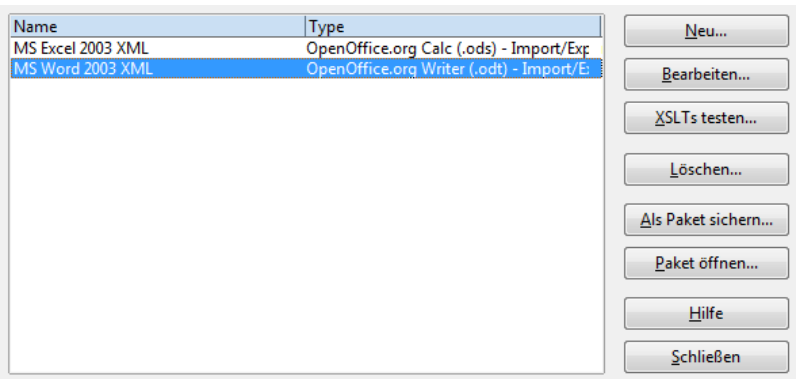

Abbildung 1.5 XML als Speicherformat in OpenOffice

Übrigens verwendet nicht nur OpenOffice.org XML als Austauschformat, auch Microsoft setzt in ein XML-basierte Format ein.

Weitere typische Beispiele für den XML-Einsatz sind

- das Web-Publishing,
- Konfigurationsdateien,
- Web-Services und die
- Druckindustrie.

Zunächst zum klassischen Fall, nämlich dem Web-Publishing. Darunter wird nichts anderes verstanden als Austausch und Darstellung von Dokumenten im WWW. In diesem Bereich tritt XML die Nachfolge von HTML an. Hier fällt als Stichwort natürlich XHTML, wobei mit XHTML die Reformulierung von HTML gemeint ist, die den XML-Regeln entspricht.

1.3.1 AJAX

Seit geraumer Zeit sorgt *Asynchronous JavaScript + XML* (AJAX) für Furore. Allerdings darf man bei all dem Hype, der um diese Technik gemacht wird, eines nicht vergessen: AJAX ist keineswegs neu. Die Technik, die dahintersteckt, existiert vielmehr schon eine ganze Weile. Allerdings gelang ihr der Durchbruch unter dem wenig blumigen Namen

XMLHttpRequest nicht. Erst mit dem Begriff AJAX und einem riesigen Marketing-Hype wurde eine breite Masse von Webentwicklern auf diese Technologie aufmerksam.

Eine der ersten und bekanntesten AJAX-Anwendungen dürfte sicherlich Google Suggest gewesen sein. Google Suggest ist eine Weiterentwicklung der Suchmaschine Google gewesen, durch die die Eingabe und Auswahl relevanter Suchbegriffe erleichtert wurde. Gibt man zum Beispiel *G* ein, wird eine Liste von Wörtern angezeigt, die mit *G* beginnen. Folgt als nächster Buchstabe *r*, zeigt Google Suggest nur noch Wörter an, die mit *Gr* beginnen. Durch den Einsatz von AJAX werden die eingetippten Buchstaben des Suchwortes im Hintergrund an den Client übertragen, ohne dass für die Übertragung ein erneuter Seitenaufruf im Browser nötig ist. Mittlerweile wird diese Funktion nicht mehr nur bei Google, sondern auch bei anderen Suchmaschinen verwendet.

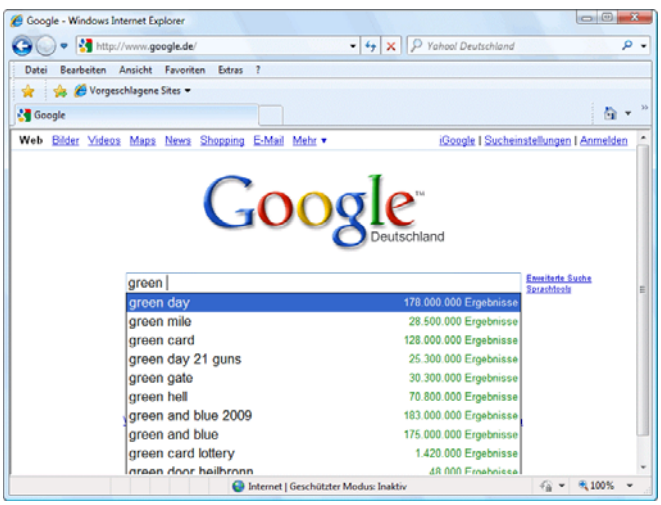

Abbildung 1.6 Google nutzt Ajax schon länger.

AJAX basiert auf einer Kombination verschiedener Technologien.

- HTML/XHTML
- CSS
- JavaScript
- REST
- SOAP
- XSLT
- XMLHttpRequest

Hinter AJAX steht die Grundidee, dass Daten zwischen Server und Client ausgetauscht werden können, ohne dass die aktuelle Seite neu geladen werden muss. Um zu verstehen, warum AJAX binnen kürzester Zeit so populär geworden ist, braucht man sich nur die Funktionsweise normaler Webanwendungen anzusehen. Dort wird jede auch noch so klei-

ne Eingabe des Nutzers an den Server gesendet, der daraufhin die jeweilige Webseite neu generiert und diese anschließend zurück an den Browser schickt.

AJAX geht hier neue Wege und vermeidet den klassischen Client-Server-Austausch. Das funktioniert prinzipiell wie beim normalen Austauschen bzw. Verändern von Inhalten einer Webseite über das DOM (Document Object Model). Auch dort muss die Seite zum Datenaustausch nicht neu geladen werden. AJAX erweitert dieses Prinzip dahingehend, dass die wechselnden Inhalte direkt vom Server geladen werden.

1.3.2 Web-Services

Als nächstes großes Einsatzgebiet kommen die Web-Services zum Einsatz. Was es damit auf sich hat, lässt sich gut anhand von eBay zeigen.

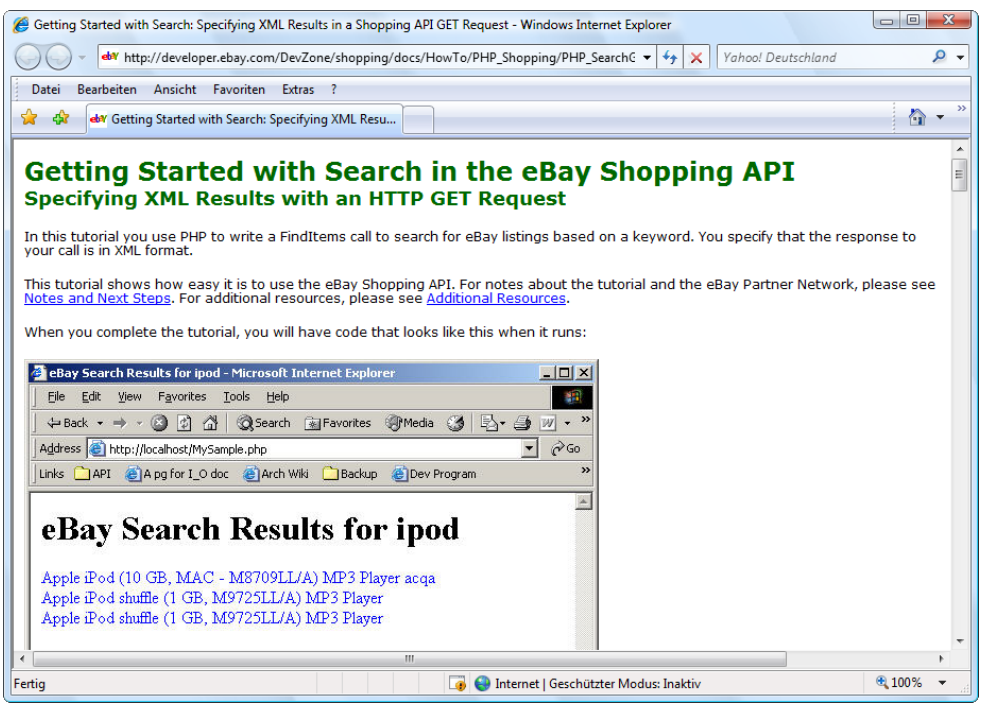

Abbildung 1.7 eBay stellt seine Auktionen auch per XML bereit.

eBay bietet entsprechende Web-Services an, durch die man als Seitenbetreiber eBay-Inhalte und -Auktionen in seine Webseite einbinden kann. Dadurch bekommt man die Möglichkeit, an jedem über seine Seite verkauften Artikel von eBay eine Provision zu erhalten. Die relevanten Daten werden dabei von eBay zur Verfügung gestellt; die so bereitgestellten Informationen liegen im XML-Format vor. Der Zugriff kann mittels verschiedener Technologien wie beispielsweise Java oder PHP erfolgen.

Listing 1.6 So klappt der Zugriff auf die eBay-API.

```
<?xml version="1.0" encoding="utf-8" ?>
<GetSearchResultsRequest xmlns="urn:ebay:apis:eBLBaseComponents">
   <RequesterCredentials>
      <eBayAuthToken>INSERT_TOKEN</eBayAuthToken>
   </RequesterCredentials>
   <Query>INSERT_QUERY</Query>
   <SearchLocationFilter>
      <CountryCode>DE</CountryCode>
      <Currency>EUR</Currency>
      <SearchLocation>
      <SiteLocation>
        <SiteID>Germany</SiteID>
      </SiteLocation>
   </SearchLocation>
   </SearchLocationFilter>
</GetSearchResultsRequest>
```

1.3.3 Druckindustrie

Auch wenn die Bedeutung elektronischer Medien immer mehr zunimmt, so spielt das gedruckte Dokument doch nach wie vor eine wichtige Rolle. Das gilt vor allem dann, wenn Informationen möglichst unkompliziert an eine große Zielgruppe weitergegeben werden sollen. Ein klassisches Beispiel dafür sind Kataloge. Gerade in diesem Bereich hat XML Einzug gehalten. Dabei legen die Anbieter ihre Daten im XML-Format ab, und diejenigen, die den Katalog „bauen“, können diese Daten in ein entsprechenden Programm wie beispielsweise InDesign importieren und weiterverarbeiten.

Abbildung 1.8 Ein Katalog wird in Adobe InDesign erstellt.

1.3.4 Konfigurationsdateien

Auch für Konfigurationsdateien wird XML verwendet. Ein typisches Beispiel dafür ist die *templateDetails.xml*, über die man die Eigenschaften von Joomla!-Templates definiert. Ein Beispiel:

Listing 1.7 Die Template-Konfiguration wird über XML geregelt.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE install PUBLIC "-//Joomla! 1.5//DTD template 1.0//EN"
"http://www.joomla.org/xml/dtd/1.5/template-install.dtd">
<install version="1.5" type="template">
   <name>hanser</name>
   <creationDate>02 April 2010</creationDate>
   <author>michael mayer</author>
     <authorEmail>kontakt@hanser.de</authorEmail>
     <authorUrl></authorUrl>
     <copyright></copyright>
     <license>GNU/GPL</license>
     <version>1.0.0</version>
     <description>Ein Template für Joomla 1.5</description>
     <files>
            <filename>css/template_weiss.css</filename>
            <filename>css/template_schwarz.css</filename>
            <filename>css/general.css</filename>
            <filename>css/ie6only.css</filename>
            <filename>css/ie7only.css</filename>
            <filename>css/index.html</filename>
            <filename>html/index.html</filename>
            <filename>images/index.html</filename>
            <filename>index.html</filename>
            <filename>index.php</filename>
            <filename>templateDetails.xml</filename>
            <filename>template_thumbnail.png</filename>
            <filename>images/betriebsgelaende.jpg</filename>
     </files>
...
</install>
```

Joomla! überprüft bei der Installation eines Templates, ob eine solche *templateDetails.xml* vorhanden ist. Anhand der innerhalb dieser Datei definierten Inhalte wird das Template eingerichtet.

Das iPhone von Apple setzt bei der Konfiguration ebenfalls auf einfachste XML-Syntax.

Listing 1.8 Und auch das iPhone spricht XML.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
     <key>PayloadContent</key>
     <array>
            <dict>
            ...
            </dict>
     </array>
     <key>PayloadDescription</key>
     <string>Profilbeschreibung.</string>
     <key>PayloadDisplayName</key>
     <string>Profilname</string>
```

```
        <key>PayloadOrganization</key>
        <string></string>
        <key>PayloadRemovalDisallowed</key>
        <false/>
        <key>PayloadType</key>
        <string>Configuration</string>
        <key>PayloadUUID</key>
        <string>5D420CD2-4E94-4EAF-8EAD-1080B5A3E237</string>
        <key>PayloadVersion</key>
        <integer>1</integer>
    </dict>
    </plist>
```

Diese beiden Beispiele zeigen, dass XML auch im Bereich der Konfiguration hilfreiche Dienste leistet.

1.4 XML und das semantische Web

Einen Blick in die Vergangenheit der XML-Entwicklung gab es auf den vorherigen Seiten. Wie aber sieht eigentlich die Zukunft dieser Sprache aus, und wo wird deren Haupteinsatzgebiet liegen? In diesem Zusammenhang taucht immer wieder der Begriff des Semantic Web auf. Was es mit diesem semantischen Web tatsächlich auf sich hat und welche Rolle XML hier spielen soll, sind nur zwei Aspekte, um die es in diesem Abschnitt geht.

Die Ansätze, die es rund um das semantische Web gibt, betreffen „normale“ XML-Entwickler derzeit kaum. Das ist auch der Grund, warum in diesem Buch dieses Thema nur kurz gestreift wird. Aber dennoch: Das Fundament des semantischen Webs wird XML bilden. Und die Grundlagen von XML und wie Sie diese Sprache praktisch einsetzen können, werden in diesem Buch ausführlich gezeigt.

1.4.1 Was Semantik bringt

Das semantische Web gehört zu den Zukunftsvisionen, über die bereits seit Längerem gesprochen wird. Dabei wird jedoch geflissentlich übersehen, dass es bereits heute durchaus brauchbare semantische Anwendungen gibt. Diese zeigen eindrucksvoll, wie das „Internet der Zukunft“ aussehen könnte.

Ein typisches Beispiel für das semantische Web sind Bildersuchmaschinen. Allerdings arbeiten die meisten Bildersuchmaschinen heute noch nicht semantisch. Gibt man z.B. in die Google-Bildersuche den Begriff *Kohl* ein, bekommt man ganz unterschiedliche Ergebnisse.

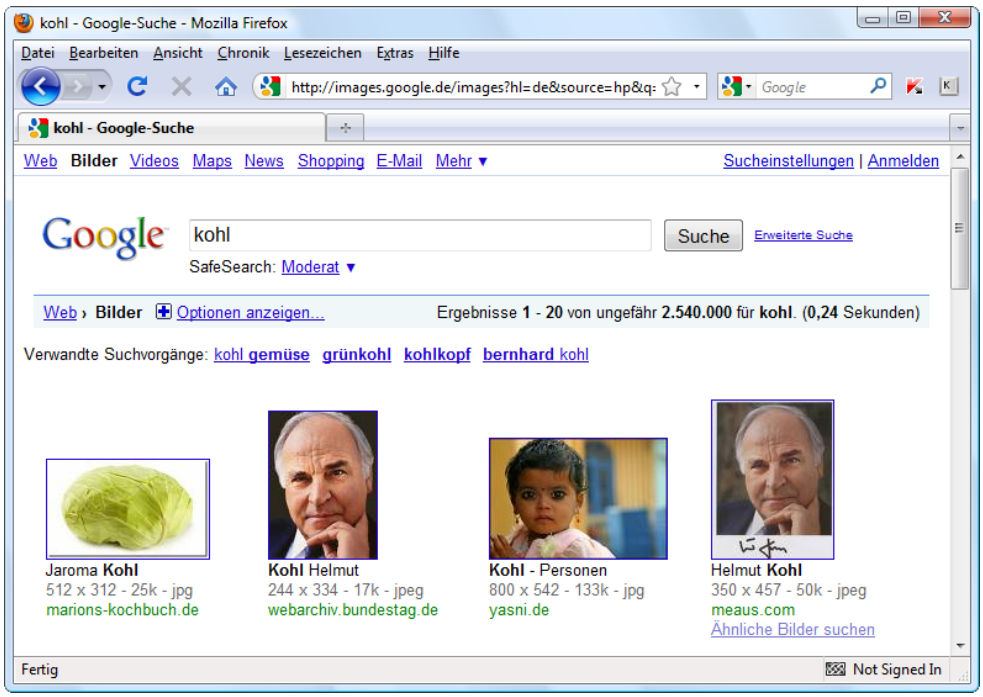

Abbildung 1.9 Die Ergebnisse sind gänzlich unterschiedlicher Natur.

Die Suchmaschine liefert für den Suchbegriff `Kohl` u.a. Ergebnisse für das Wintergemüse und den Altkanzler. Genau hier setzt das semantische Web an. Denn in diesem „Web der Zukunft“ oder dem „Web 3.0“ sollen Suchmaschinen erkennen, was auf Bildern zu sehen ist. Ziel des semantischen Webs ist es, die Bedeutung von Informationen – im vorliegenden Fall also die der Fotos – für Computer verwertbar zu machen. Das ist ein deutlicher Vorteil gegenüber dem heutigen WWW. Denn momentan können die bereitgestellten Informationen zwar von Menschen, nicht aber von Maschinen korrekt interpretiert werden. Im semantischen Web sollen sich Personen, Dinge und Orte hingegen in Beziehung zueinander setzen lassen. Damit aber nicht genug: Die Verknüpfung der einzelnen Informationen könnte zusätzlich zur Aufdeckung bis dato unerkannter Zusammenhänge führen.

Für viele Menschen ist das semantische Web reines Zukunftsdenken. Das stimmt so allerdings nicht. Denn bereits heute gibt es erste interessante Anwendungen, die sich auf das Grundprinzip des semantischen Webs stützen. Das gilt vor allem für den Bereich der neuen Generation von Bildersuchmaschinen wie z.B. ImageNotion (*http://www.imagenotion.com/*).

1.4.1.1 Auf dem Weg zum Web 3.0

Spricht man vom semantischen Web, fällt fast synonym der Begriff Web 3.0. Das trifft es allerdings nicht ganz. Vielmehr handelt es sich beim Web 3.0 um den Zusammenschluss zweier Webvarianten:

- Semantisches Web
- Web 2.0

Das semantische Web – und das wurde auf den vorherigen Seiten bereits gezeigt – versucht, die Qualität vorhandener Informationen hinsichtlich der Semantik zu verbessern. Der Begriff Web 2.0 beschreibt hingegen eine Reihe bestimmter Technologien, die vor allem für Interaktion stehen. Dazu gehören z.B. RSS-Feeds, AJAX und Webservices. Aber auch soziale Netzwerke sind charakteristisch für das Web 2.0.

Im Web 3.0 sollen beide Webvarianten zusammengefügt werden. Semantische Technologien und die sozialen Komponenten des Web 2.0 bilden in Zukunft das Social Semantic Web bzw. das Web 3.0.

1.4.1.2 Annotation und Ontologie

Die Eckpfeiler des semantischen Webs sind Annotationen und Ontologien. Maschinen – also z.B. auch Suchmaschinen – sind aber nur imstande, solche Informationen auszulesen, die auch tatsächlich vorhanden sind. Dank Annotationen können Informationen zusätzliche Meta-Daten zugewiesen werden. Diese Zusatzinformationen sind die Grundvoraussetzung für das semantische Web.

Anhand unterschiedlicher Beschreibungssprachen wie Mikroformaten oder RDF kann die Bedeutung von Inhalten, bis zu einem gewissen Grad, auch für Maschinen interpretierbar gemacht werden.

Hinter den Annotationen verbirgt sich das Ontologien-Konzept. Vergleichen lässt sich eine Ontologie am ehesten mit einer Datenbank. Auch bei Ontologien bilden Struktur und Inhalt das große Ganze. Informationen werden aber nicht einfach nur als Text in einer Datenbank gespeichert, sondern vorhandenes Wissen wird weiter interpretiert. Ein Beispiel soll diesen Aspekt verdeutlichen. Angenommen, man hat als Konzepte „Hanser, „Verlag“ und „Bücher“ sowie die Beziehungen „Hanser ist ein Verlag“ und „Verlage veröffentlichen Bücher“. Ein semantisches System könnte daraus schlussfolgern, dass „Hanser Bücher veröffentlicht“. Gäbe man nun in eine Bildersuchmaschine „Alle Bücher von Verlagen“ ein, würden auch die Bilder gefunden werden, die mit dem Konzept „Hanser“ annotiert sind.

1.4.1.3 Wichtige Sprachen für das semantische Web

Das semantische Web wird größtenteils auf XML bzw. auf XML-basierten Sprachen basieren. Für deren Entwicklung ist maßgeblich das W3C verantwortlich. Die zentrale Anlaufstelle rund um dieses Thema ist die W3C Semantic Web Activity (*http://www.w3.org/2001/sw/*). Auf dieser Seite finden Sie stets aktuelle Informationen zu den Semantikbestrebungen des W3C.

Momentan werden die folgenden W3C-Empfehlungen favorisiert, wenn es um den Aufbau von Ontologien und Annotationen geht:

- RDF – Eine Beschreibungssprache für Informationen einer Webressource.
- OWL – Eine Beschreibungssprache für Klassen und Relationen.
- SKOS – Eine formale Sprache für die Kodierung von Dokumentationssprachen.
- SPARQL – Ein Protokoll und eine Abfragesprache.

Diese Ansätze scheinen derzeit am ehesten für die Aufgaben, die das Web 3.0 stellt, gewachsen zu sein. Ob sie sich aber letztendlich tatsächlich durchsetzen werden, wird die Zukunft zeigen. Eines scheint aber festzustehen: Das Web 3.0 ist ohne XML nicht denkbar.

1.4.1.4 RDF bildet die Grundlage

Die Basis des semantischen Webs ist das vom W3C entwickelte Resource Description Framework (RDF). RDF setzt sich aus vier Komponenten zusammen:

- Ressourcen – Das sind die Dinge, die mittels RDF beschrieben werden. Ressourcen werden durch einen URI und eine optionale Anker-ID spezifiziert.
- Eigenschaften (Properties) – Das sind Attribute oder Beziehungen, die eine Ressource genauer beschreiben.
- Literale (Literals) – Die Literale können atomare Werte wie beispielsweise Unicode-Strings enthalten. Eigenschaften lassen sich durch Literale ausdrücken.
- Aussagen (Statements) – Mit Ressourcen und Eigenschaften können in RDF Aussagen formuliert werden, indem man der Eigenschaft einer Ressource einen Wert zuweist.

Aktuelle Entwicklungen zu RDF können Sie auf der Seite *http://www.w3.org/RDF/* verfolgen.

RDF wurde entwickelt, um die klassischen Prinzipien des Webs (Verlinkungen, Offenheit usw.) von Dokumenten auch auf andere Daten übertragen zu können. Da die Daten in RDF formal repräsentiert werden, sind sie auch von Maschinen auswertbar.

RDF ist ein Datenmodell für die Repräsentation von Metadaten, über die Ressourcen beschrieben werden können. Allerdings besitzt RDF selbst keine Möglichkeit, Strukturinformationen des generischen Konzepts zu beschreiben. Man kann also z.B. nicht angeben, welche Klassen von Ressourcen es gibt. Dafür gibt es dann wiederum RDF Schema. Dieses RFDS ermöglicht die Definition von Schemata, durch die sich Begriffe für bestimmte Einsatzgebiete in maschinenlesbarer Form festlegen lassen.

Das RDF-Modell ist nicht auf eine bestimmte Präsentation festlegt. Am häufigsten wird derzeit allerdings XML verwendet.

Listing 1.9 Eine einfache RDF-Syntax

```
<?xml version='1.0' encoding='ISO-8859-1'?>
  <rdf:RDF xmlns:rdf=http://www.w3.org/1999/02/22-rdf-syntax-ns#
      xmlns:rdfs=http://www.w3.org/TR/1999/PR-rdf-schema-19990303#
      xmlns:s0=http://www.w3.org/2000/PhotoRDF/dc-1-0#
      xmlns:s1=http://www.w3.org/2000/PhotoRDF/technical-1-0#
      xmlns:s2="http://sophia.inria.fr/~enerbonn/rdfpiclang#">
```

```
    <rdf:Description rdf:about="">
      <s0:creator>Bert Bos</s0:creator>
      <s0:relation>Marian in the Tarn</s0:relation>
      <s0:rights>Bert Bos</s0:rights>
      <s0:type>image</s0:type>
      <s0:identifier>990621</s0:identifier>
      <s0:coverage>Montredon-Labessonié (Tarn)</s0:coverage>
      <s0:date>1999-06-26</s0:date>
      <s1:camera>Canon Eos 5</s1:camera>
      <s2:xmllang>en</s2:xmllang>
      <s0:title>Marian with sheep</s0:title>
      <s0:subject>Landscape, Animal</s0:subject>
      <s0:publisher>Bert Bos</s0:publisher>
      <s0:description>Marian brings the sheep to the
      field in the morning. The lamb she carries was
      born that night.</s0:description>
      <s0:format>image/jpeg</s0:format>
    </rdf:Description>

  </rdf:RDF>
```

Ebenso gibt es aber auch eine verkürzte Syntax, die von Tim Berners-Lee entwickelt wurde. Diese Notation 3 ist deutlich einfacher als die XML-Variante. Ausführliche Informationen zu Notation 3 finden Sie unter *http://www.w3.org/DesignIssues/Notation3.html*.

1.4.1.5 OWL

Derzeit ist die Web Ontology Language (OWL) die am häufigsten eingesetzte Sprache für die Modellierung von Ontologien. Mit OWL können anhand einer formalen Beschreibungssprache Ontologien erstellt, veröffentlicht und verteilt werden. OWL stellt eine Plattform für die Entwicklung themenspezifischer Vokabulare dar, mit der sich Inhalte beschreiben lassen. Dabei setzt OWL RDF und RDFS ein und nutzt zusätzliche Vokabeln, mit denen sich Eigenschaften und Klassen beschreiben lassen.

Ebenso wie durch RDFS lassen sich in OWL Terme einer Domäne und deren Beziehung formal beschreiben. Hier weist OWL allerdings deutlich komplexere Funktionen auf, durch die sich die Beziehungen beschreiben lassen. Das W3C hat es sich zum Ziel gesetzt, mit OWL die Probleme, die bei RDF und RDFS auftauchen, zu lösen. Denn bei diesen beiden Ansätzen stand zunächst die Metadatenverwaltung im Vordergrund. Erst nachträglich erkannte man, dass sich darüber auch ontologische Strukturen abbilden lassen. Bei OWL stand nun exakt dieser Aspekt im Vordergrund.

Listing 1.10 Hier werden Konzepte und Klassen definiert.

```
<owl:Class rdf:ID="Geschlecht"/>
  <owl:Class rdf:ID="Mann"/>
  <owl:Class rdf:ID="Frau">
    <rdfs:subClassOf rdf:resource="#Mann"/>
    <owl:equivalentClass>
      <owl:Restriction>
        <owl:onProperty rdf:resource="#Geschlecht"/>
        <owl:hasValue rdf:resource="#maennlich" rdf:type="#Geschlecht"/>
      </owl:Restriction>
    </owl:equivalentClass>
  </owl:Class>
...
```

Diese Syntax zeigt, wie sich Konzepte und Klassen in OWL definieren lassen. Weiterführende Informationen zu OWL finden Sie unter *http://www.w3c.org/TR/owl-semantics*/.

1.4.1.6 SKOS

Das Simple Knowledge Organisation System (SKOS) ist eine auf RDF basierende formale Sprache für die Kodierung von Dokumentationssprachen. Mit SKOS können semantische Strukturen beschrieben und maschinenlesbar gemacht werden. SKOS wurde als modulare und erweiterbare Sprachfamilie entworfen, deren Einsatz so einfach wie möglich sein soll. Zentrale Anlaufstelle für diese Sprache ist die Seite *http://www.w3.org/2004/02/skos/*.

1.4.1.7 SPARQL

SPARQL (ein rekursives Synonym für SPARQL Protocol and RDF Query Language) ist eine Abfragesprache für RDF. Aktuelle Entwicklungen rund um SPARQL können auf der Seite *http://www.w3.org/2009/sparql/wiki/Main_Page* verfolgt werden. Hinter SPAQL verbirgt sich eine ganze Reihe von Empfehlungen.

- SPARQL Query Language for RDF – Das ist die Abfragesprache für RDF-Datenbestände.
- SPARQL Protocol for RDF – Hier wird das Protokoll beschrieben, durch das die Kommunikation mit RDF-Datenquellen, die über SPARQL angesprochen werden, definiert wird. Bei diesen Datenquellen handelt es sich um die sogenannten Anfrage-Endpunkte bzw. SPARQL-Endpoints.
- SPARQL Query Results XML Format – Definiert ein Format, in dem die SPARQL-Endpunkte Anfrage-Ergebnisse an den Client zurückliefern.

Für SPARQL gibt es bereits einige Implementierungen in den verschiedenen Sprachen. Mit ARC *(http://arc.semsol.org/*) steht z.B. eine entsprechende Lösung für PHP bereit.

1.5 XML-Editoren im Einsatz

Wenn Sie mit XML arbeiten, brauchen Sie natürlich auch ein entsprechendes Werkzeug. Genau an dieser Stelle kommen XML-Editoren zum Einsatz beziehungsweise können zum Einsatz kommen. Denn genau genommen können Sie Ihre XML-Dokumente auch mit ganz „primitiven“ Tools wie dem Windows-Standard-Texteditor anlegen. Komfortabel ist das aber freilich nicht.

Ob Sie einen speziellen XML-Editor oder einen klassischen Texteditor verwenden, bleibt zunächst natürlich Ihnen selbst überlassen. Sie können Ihre XML-Dokumente zum Beispiel sehr gut mit Ultra Edit entwickeln. Gerade dann, wenn Sie einfache XML-Anwendungen entwickeln, reicht dieser Editor allemal. Eine Testversion können Sie sich von der Seite *http://www.ultraedit.com/* herunterladen. Wenn Sie sich anschließend für den Kauf entscheiden, werden derzeit rund 34 Euro fällig.

1.5.1 XMLSpy

UltraEdit reicht in jedem Fall aus, um XML-Dokumente zu erstellen. Wollen Sie hingegen einen reinen XML-Editor einsetzen, empfiehlt sich beispielsweise XMLSpy von Altova.

Ein solcher spezieller Editor bietet sich immer dann an, wenn der XML-Code komplexer wird und über 10, 20 oder 30 Zeilen hinausgeht. Bei XMLSpy handelt es sich genau genommen weniger um einen Editor als vielmehr um eine komplexe Entwicklungsumgebung. Dabei können Sie mit XMLSpy XML-Dokumente transformieren, editieren und debuggen. Darüber hinaus hält dieser Editor solche Funktionen wie einen grafischen Schema-Designer, einen Code-Generator sowie einen Dateikonverter bereit.

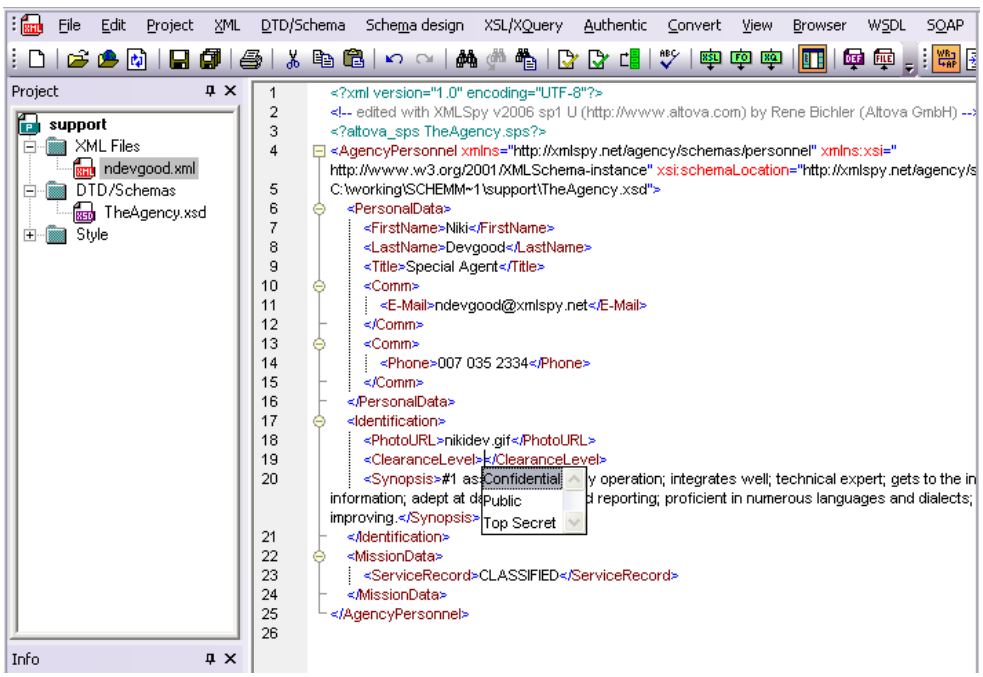

Abbildung 1.10 Die perfekte Arbeitsumgebung für XML-Entwickler

Herunterladen können Sie sich eine Testversion von der Seite *http://www.altova.com/de/*. Für die Vollversion werden dann in der Professional-Variante 399 und in der Enterprise-Version 799 Euro fällig.

1.5.2 XMLBlueprint

Ein weiterer sehr komfortabler Editor ist XMLBlueprint. Das Programm ist mit 70 US-Dollar vergleichsweise günstig. Das gilt umso mehr vor dem Hintergrund, dass XMLBlueprint zahlreiche interessante Funktionen zu bieten hat. Dazu gehören zunächst einmal die Dinge, die man von einem professionellen XML-Editor erwarten kann:

- Syntaxhervorhebung
- Validierung
- FTP-Funktion
- Unicode-Unterstützung

Neben diesen Basisfunktionen hat XMLBlueprint allerdings noch sehr viel mehr zu bieten. So kann man hiermit beispielsweise sehr einfach DTD, Relax NG Schema oder XML Schema erstellen. Dabei werden in einem Auswahlfeld die relevanten Elemente angezeigt, die man hinsichtlich des zugrunde liegenden XML-Dokuments verwenden kann. Nachdem man das Element ausgewählt hat, wird das Element automatisch geschlossen.

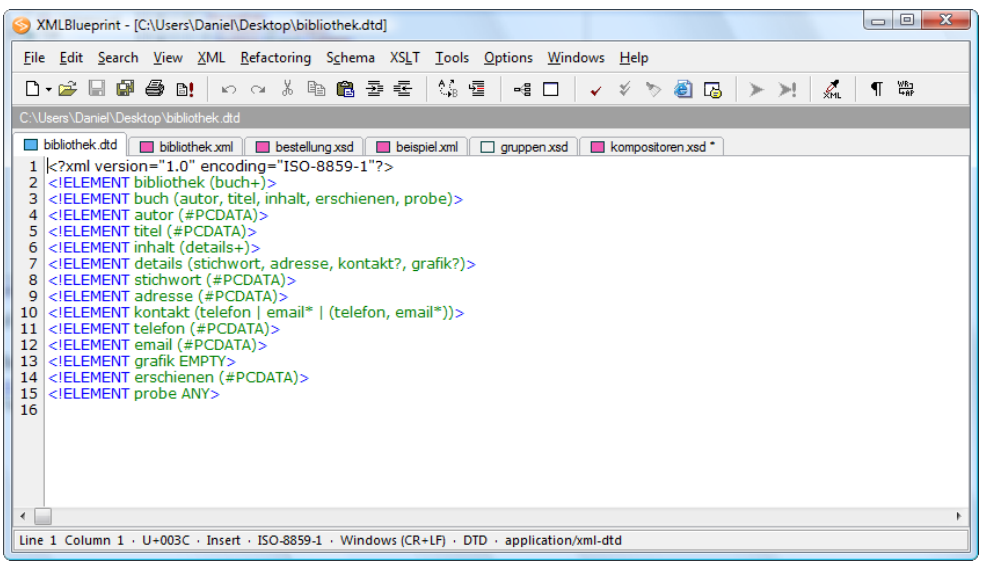

Abbildung 1.11 Hier wird gerade eine DTD bearbeitet.

Sämtliche Beispiele in diesem Buch wurden übrigens mit XMLBlueprint erstellt. Ausführliche Informationen zu diesem Editor finden Sie unter *http://www.xmlblueprint.com/*.

2

2 Die XML-Grundlagen

Bevor man mit XML arbeiten kann, gilt es zu verstehen, was die Sprache eigentlich ausmacht. In diesem Kapitel erfahren Sie alles Wissenswerte zu Syntax und Beschaffenheit Ihrer neuen Sprache. Den Anfang macht, wie so oft, ein klassischen *Hallo, Welt!*-Beispiel, das den Weg hin zu XML ebnet. Weiter geht es mit den Themen XML-Deklaration, DTDs, Elementen und Attributen. Sie sehen also, dass auf den folgenden Seiten eine ganze Menge neuer Dinge auf Sie zukommen wird. Die Grundlagen werden dabei so schnell wie möglich, aber gleichzeitig auch so gründlich wie nötig behandelt. Denn schließlich bilden sie das Fundament für Ihre späteren Arbeiten im XML-Umfeld.

2.1 Hallo, Welt zum Einstieg

Den Einstieg in die XML-Entwickler-Welt! soll, wie das in der Programmierwelt üblich ist, ein *Hallo, Welt!*-Beispiel ebnen. Zuvor jedoch ein wichtiger Hinweis zu der innerhalb dieses Buches – und auch in der XML-Welt – verwendeten Semantik. Denn Ihnen werden permanent die Begriffe Tag und Element begegnen, die leider immer wieder falsch interpretiert werden. Ein Beispiel:

```
<gruss art="begruessung">Hallo, Welt!</gruss>
```

Diese Syntax zeigt ein `gruss`-Element, und zwar `<gruss art="begruessung">Hallo, Welt!</gruss>`. Dieses Element setzt sich aus einem öffnenden `gruss`-Tag, also `<gruss>` bzw. vollständig `<gruss art="begruessung">`, dem Elementinhalt `Hallo, Welt!` und dem schließenden `gruss`-Tag `</gruss>` zusammen. Zudem besitzt das Element `gruss` das Attribut `art` mit dem Attributwert `begruessung`.

Dieses Beispiel zeigt, dass die meisten Leute Element meinen, wenn sie von Tag sprechen. In diesem Zusammenhang hilft ein Blick auf HTML. Dort werden Hyperlinks über das Element `a` definiert. Dieses Element `a` wird durch das öffnende `<a>`- und das schließende `</a>`-Tag ausgezeichnet. Sie sehen, dass es gar nicht so schwer ist, die Bezeichnungen Element und Tag richtig zu verwenden.

Ohne bereits jetzt ausführlich auf die XML-Syntax eingehen zu wollen, ein erster Beispielcode:

Listing 2.1 Hallo, Welt! auf XML.

```
<?xml version="1.0"?>
<welt>
   <gruss>Hallo, Welt!</gruss>
</welt>
```

Es handelt sich hierbei um ein sehr einfaches XML-Dokument. Sie sehen darin die beiden Elemente `welt` und `gruss` sowie den Inhalt des Elements `gruss`, nämlich die Zeichenfolge `Hallo, Welt!`. Legen Sie sich diese Datei in dem Editor Ihrer Wahl an, und speichern Sie sie beispielsweise unter dem Namen *welt.xml* ab. Diese Datei können Sie in Ihrem Browser aufrufen. Im Mozilla Firefox sieht das Ergebnis dann wie in **Abbildung 2.1** aus.

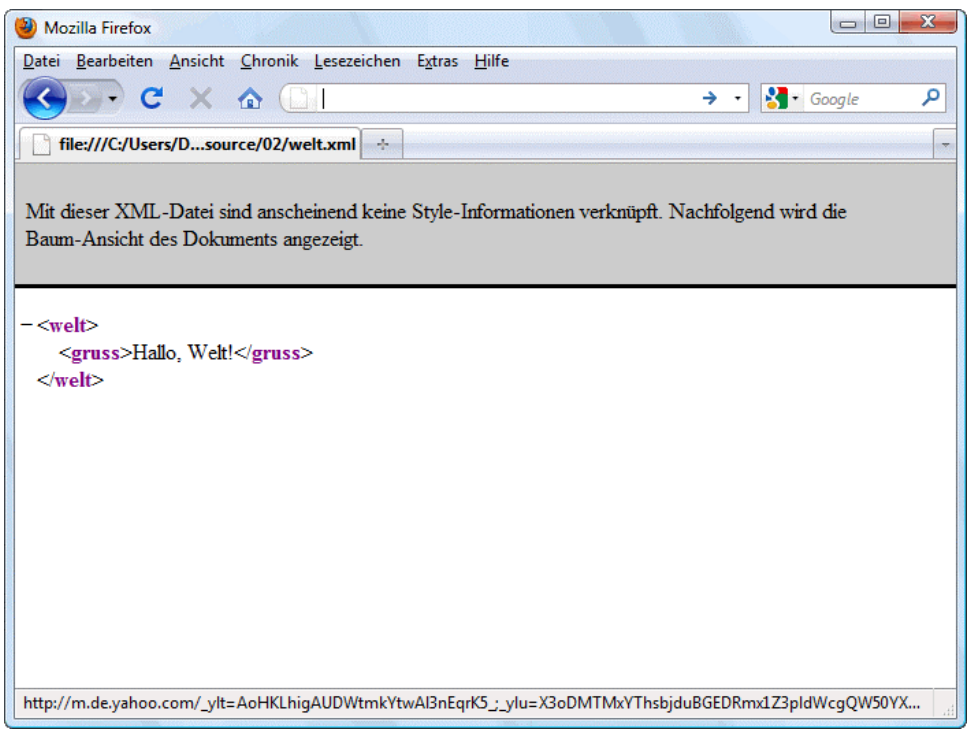

Abbildung 2.1 So sieht die Datei im Firefox aus.

Hier wird der sogenannte Dokumentenbaum angezeigt. Über das Minuszeichen können Sie die Elemente schließen. Mittels der daraufhin angezeigten Pluszeichen lassen sich diese Elemente dann wieder öffnen. Zusätzlich weist Sie der Firefox im oberen Fensterbereich darauf hin, dass mit dem Dokument keine Style-Informationen verknüpft sind. Das ist in diesem Beispiel zunächst tatsächlich so. Im weiteren Verlauf dieses Buches erfahren Sie, wie Sie solche Style-Informationen anlegen und mit Ihren XML-Dokumenten verknüpfen können.

Ganz ähnlich sieht die Datei aus, wenn sie im Internet Explorer aufgerufen wird.

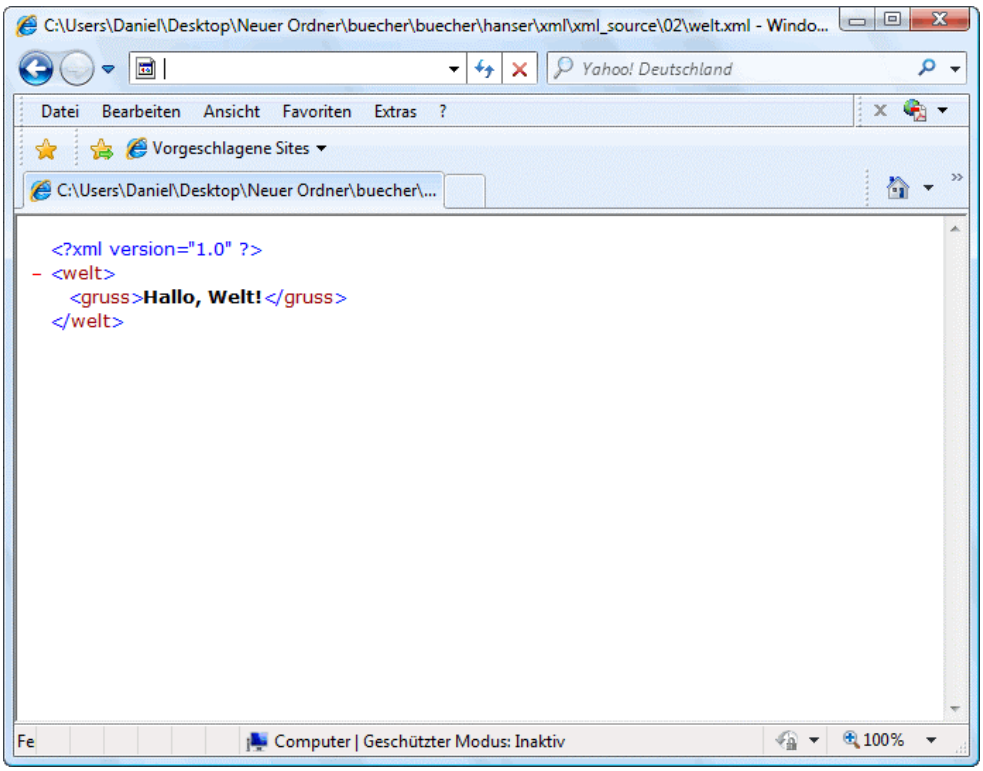

Abbildung 2.2 Die gleiche Datei im Internet Explorer

In diesem Browser wird ebenfalls der Dokumentenbaum angezeigt. Und genauso wie im Firefox lassen sich auch hier die Elemente über Plus- und Minuszeichen aus- und wieder einblenden. Der Internet Explorer verwendet für die Darstellung der Daten ein integriertes Stylesheet. So werden die Tags beispielsweise braun angezeigt, während die Elementinhalte schwarz sind.

Sie haben gesehen, wie beide Browser XML-Dokumente darstellen. Bislang funktionierte die Anzeige der Daten allerdings fehlerfrei. Anders gesagt: Das *Hallo, Welt!*-Beispiel war syntaktisch korrekt. Um Ihnen zu zeigen, was passiert, wenn das nicht der Fall ist, hier noch einmal das gleiche Dokument, in das nun aber absichtlich ein Fehler eingebaut wurde.

Listing 2.2 Hier hat sich ein Fehler eingeschlichen.

```
<?xml version="1.0"?>
<welt>
   <gruss>Hallo, Welt!</ggruss>
</welt>
```

Es handelt sich um einen kleinen Tippfehler. Anstelle des korrekten `</gruss>`-Tags wurde `</ggrus>` eingebaut. Solche Flüchtigkeitsfehler passieren in der Praxis recht häufig. Im

aktuellen Beispiel entspricht das End-Tag `</ggrus>` nicht mehr dem Start-Tag `<gruss>`. Wird dieses fehlerhafte Dokument im Browser aufgerufen, ergibt sich ein Anblick wie in **Abbildung 2.3**.

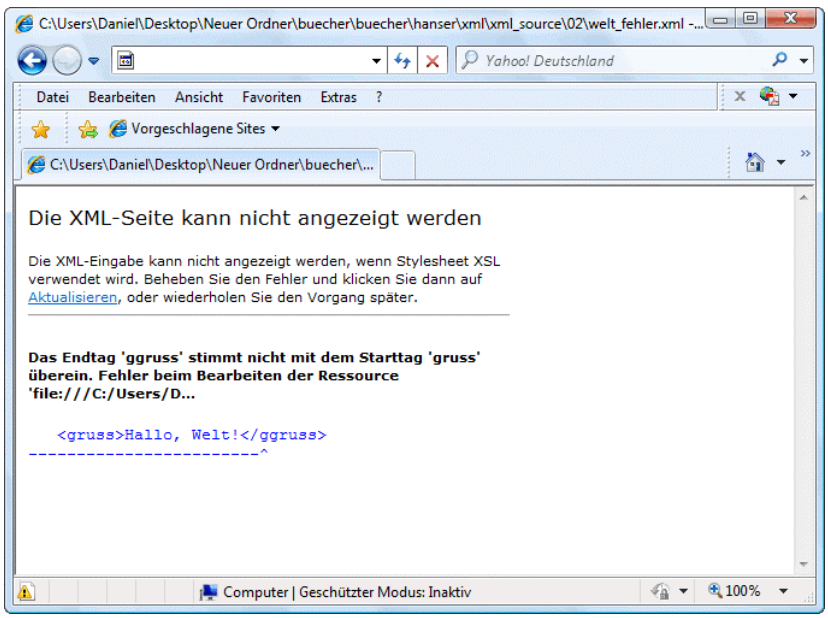

Abbildung 2.3 Hier stimmt etwas nicht.

Der Internet Explorer weist in einer Fehlermeldung darauf hin, dass Start- und End-Tag nicht übereinstimmen. Im Firefox sieht das ähnlich aus.

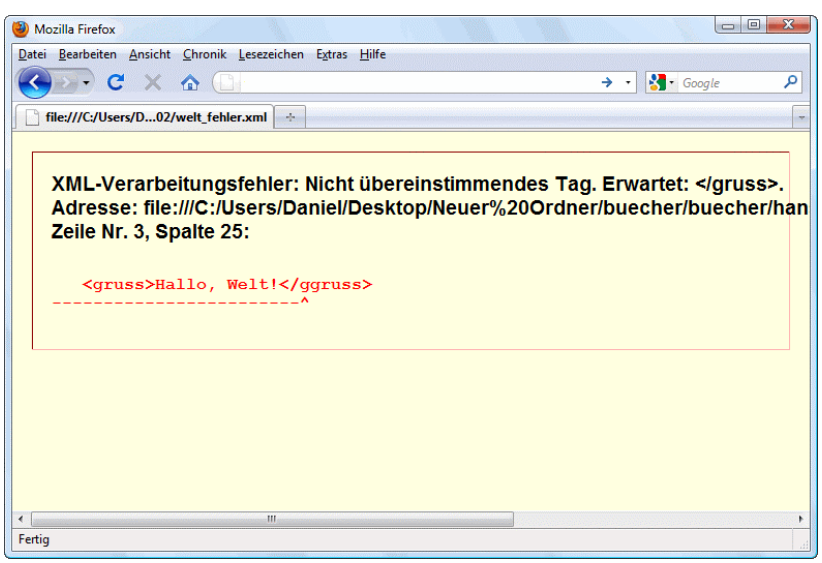

Abbildung 2.4 Der Firefox moniert die Syntax ebenfalls.

Auch hier wird also ein XML-Verarbeitungsfehler angezeigt. Die Fehlermeldungen sind bei der täglichen Arbeit natürlich äußerst hilfreich, schließlich sehen Sie auf diese Weise sofort, an welchen Stellen im XML-Dokument Sie gegebenenfalls nachbessern müssen.

2.2 Die XML-Deklaration

Damit ein XML-Dokument auch tatsächlich als solches zu erkennen ist, gibt es das `xml`-Element, das am Beginn von XML-Dokumenten notiert wird. Dieses Element wird als XML-Deklaration bezeichnet. Notwendig ist die XML-Deklaration nicht. Wird sie aber verwendet, muss sie in der ersten Zeile stehen, und weder Leerraum noch anderer Inhalt dürfen ihr vorangestellt werden.

Hier zunächst die Syntax dieses Elements:

```
<?xml version="1.0"?>
<!-- hier folgt der Inhalt der XML-Datei -->
```

Die XML-Deklaration ist eine alleinstehende Anweisung. Sie besitzt, anders als in XML üblich, kein schließendes Tag. Innerhalb der öffnenden und schließenden Klammer sind das erste und letzte Zeichen ein Fragezeichen. Hinter dem ersten Fragezeichen folgt die Zeichenfolge `xml`. Hierbei ist darauf zu achten, dass diese in Kleinbuchstaben notiert wird. Hinter `xml` können verschiedene Attribute angegeben werden. Notiert werden muss in jedem Fall das Attribut `version` mit dem Wert `1.0`.

Bei den möglichen Attributen wie `version`, `encodiging` usw. handelt es sich um Pseudo-Attribute. Diese Bezeichnung rührt daher, dass diese für XML-Parser technisch gesehen keine wirklichen Attribute sind. Welche dieser Pseudo-Attribute es neben `version` außerdem gibt, wird auf den folgenden Seiten gezeigt.

2.2.1 Den verwendeten Zeichensatz angeben

Sonderzeichen und Umlaute stellen in XML zunächst einmal ein Problem dar. Ohne eine geeignete Angabe zum Zeichensatz, lassen sich XML-Dokumente möglicherweise nicht darstellen, und es werden Fehlermeldungen erzeugt. Da beispielsweise XML-Entwickler aus dem deutschen Sprachraum nur schwer ohne Umlaute auskommen, muss innerhalb der XML-Deklaration eine entsprechende Anweisung zum Zeichensatz notiert werden. XML interpretiert, wenn nichts anderes angegeben wurde, den Inhalt einer Datei gemäß dem Zeichensatz UTF-8. Bei dieser Variante werden alle Zeichen in variable lange Kodierungen umgesetzt. Das hat den Vorteil, dass sich bei den ersten 128 Zeichen der Unicode mit den 7 Bit des ASCII-Codes deckt. Positiv kommt hinzu, dass man auch mit einfachen Texteditoren XML-Dateien erstellen kann.

Damit Umlaute korrekt angezeigt werden, muss der ISO-Zeichensatz 8859-1 angewandt werden. Die hierfür modifizierte XML-Deklaration sieht dann folgendermaßen aus:

Listing 2.3 Diese Syntax wird korrekt ausgegeben.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<welt>
   <gruss>Übermorgen fährt die Straßenbahn wieder.</gruss>
</welt>
```

Um auf eine XML-Datei einen bestimmten Zeichensatz anwenden zu können, muss innerhalb der XML-Deklaration das Attribut `encoding` stehen. Als Wert muss man den gewünschten Zeichensatz zuweisen. Tabelle 2.1 zeigt alle Zeichensätze, die von den gängigsten XML-Parsern[1] unterstützt werden.

Tabelle 2.1: Wichtige ISO-Zeichensätze

Standard	Zeichensatz
UTF-8	internationaler Zeichensatz mit 8 Bit Zeichenbreite
UTF-16	internationaler Zeichensatz mit 16 Bit Zeichenbreite
ISO-8859-1	Westeuropa, Lateinamerika
ISO-8859-2	Osteuropa
ISO-8859-3	Südeuropa
ISO-8859-4	Skandinavien, Baltikum
ISO-8859-5	Kyrillisch
ISO-8859-6	Arabisch
ISO-8859-7	Griechisch
ISO-8859-8	Hebräisch
ISO-8859-9	Türkisch
ISO-8859-10	Lappland

Der Internet Explorer ist, wenn er auf ein Problem mit dem Zeichensatz stößt, hinsichtlich der Fehlermeldung übrigens nicht sonderlich hilfreich.

Listing 2.4 Das führt im Internet Explorer zu einem Fehler.

```
<?xml version="1.0"?>
<welt>
   <gruss>Übermorgen fährt die Straßenbahn wieder.</gruss>
</welt>
```

Wie **Abbildung 2.5** zeigt, ist die Fehlermeldung irreführend bzw. nicht eindeutig.

[1] Ein XML-Parser ist ein Programm, das ein XML-Dokument „durchliest“, analysiert und einer entsprechenden Anwendung zur Verfügung stellt.

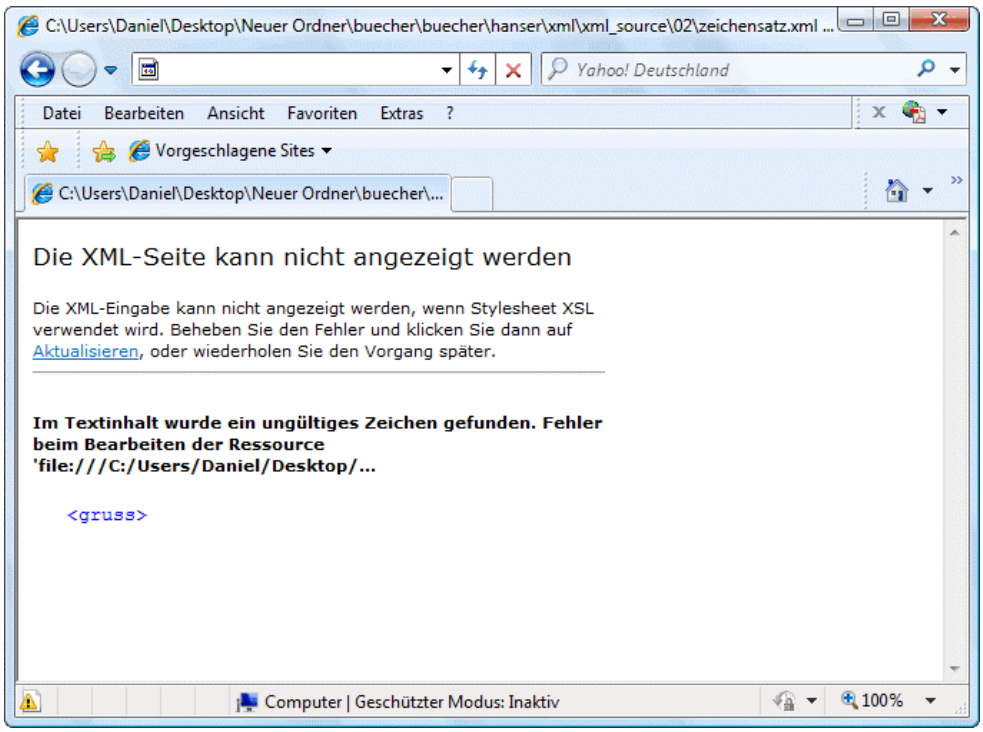

Abbildung 2.5 Die gleiche Datei im Internet Explorer

Achten Sie darauf, dass der Internet Explorer nicht das Element selbst, sondern lediglich dessen Inhalt moniert. Taucht eine solche Fehlermeldung auf, liegt das mit ziemlicher Sicherheit an einem falschen Zeichensatz bzw. an einem Zeichen, das vom verwendeten Zeichensatz nicht unterstützt wird.

2.2.2 Angaben zur Dokumenttypdefinition

Als weiteres Attribut der XML-Deklaration kann `standalone` verwendet werden.

```
<?xml version="1.0" standalone="yes"?>
<!-- hier folgt der Inhalt der XML-Datei -->
```

Durch die Zuweisung dieses Attributs innerhalb der XML-Deklaration können Sie dem Parser mitteilen, ob sich die aktuelle Datei auf eine externe DTD bezieht. Mit dem Wert `yes` geben Sie an, dass sich die DTD innerhalb der aktuellen XML-Datei befindet. Wurde die DTD indes in ein externes Dokument ausgelagert, muss hier `no` stehen.

Obwohl die beiden Attribute `standalone` und `encoding` optional sind, muss bei deren Notation eine bestimmte Reihenfolge eingehalten werden. Hinter dem Attribut `version`, was stets als erstes notiert werden muss, folgt `encoding` und als letztes `standalone`.

2.3 Elemente definieren

Genau wie in HTML werden auch in XML Elemente verwendet. XML unterscheidet allerdings zwischen Groß- und Kleinschreibung. Zudem existieren in HTML ausschließlich fest definierte Elemente. In XML können Sie diese hingegen frei definieren. So viele Freiheiten Sie dadurch auch haben, gleichzeitig wird Ihnen mehr Verantwortung hinsichtlich einer „sauberen“ Programmierung übertragen. Sehen Sie sich zunächst ein typisches XML-Element an.

```
<Buch>Streifzüge durch das Abendland</Buch>
```

Das einleitende Tag `<Buch>` muss exakt so wie das schließende `</Buch>`-Tag notiert werden. Das vorherige und nachfolgende sind zwar beides gültige HTML-Elemente, aus Sicht von XML handelt es sich hierbei allerdings um zwei verschiedene.

```
<buch>Streifzüge durch das Abendland</buch>
```

XML-Elemente unterliegen weiteren Konventionen:

- Elemente müssen mit einem Buchstaben oder einem Unterstrich beginnen.
- Elementnamen dürfen nur aus gültigen Zeichen bestehen. Man darf Buchstaben, Unterstriche, Bindestriche, Punkte und Ziffern verwenden. Der Einsatz von Umlauten und Sonderzeichen ist hingegen nicht erlaubt.
- Beachten Sie, dass Elemente nicht mit einer der beiden Zeichenfolgen `xml` oder `XML` beginnen dürfen. Diese sind für aktuelle und zukünftige XML-Spezifikationen reserviert.
- Im eigentlichen Sinn sind innerhalb von Elementen auch Doppelpunkte gestattet. Da diese Notation aber für die Definition von Namensräumen reserviert ist, sollten Doppelpunkte innerhalb von Elementen auch nur in diesem Zusammenhang eingesetzt werden.

Tabelle 2.2 zeigt Ihnen einige Beispiele für richtige und falsche XML-Elemente.

Tabelle 2.2: Richtige und falsche XML-Elemente

Tag	richtig oder falsch	Begründung
`<xml-Buch>`	falsch	Das Element beginnt mit den drei Buchstaben `xml`, die aber in der XML-Spezifikation als reserviert gekennzeichnet sind.
`<Buch-Titel>`	richtig	Der Bindestrich ist erlaubt.
`<Buch=Titel>`	falsch	Das Gleichheitszeichen darf in XML-Elementen nicht vorkommen.
`<Buch Titel>`	falsch	Leerzeichen sind nicht gestattet.
`<1Buch>`	falsch	XML-Elemente dürfen nicht mit einer Ziffer beginnen.
`<Buch1>`	richtig	Ziffern sind in XML-Elementen erlaubt, nur eben nicht als erstes Zeichen.

Noch einige Worte zur Groß- und Kleinschreibung von XML-Elementen: Es ist nicht festgelegt, in welcher Schreibweise Elemente notiert werden müssen. Sie als Entwickler sollten sich allerdings eine einheitliche Notation angewöhnen. Schreiben Sie entweder alle Elemente klein, alle groß oder alle nach den Regeln der Rechtschreibung. Um die Übersichtlichkeit der XML-Datei zu wahren, sollte die Schreibweise in jedem Fall konstant eingehalten werden.

Was passiert, wenn bei der Elementdefinition nicht auf Groß- und Kleinschreibung geachtet wird, zeigt **Abbildung 2.6**.

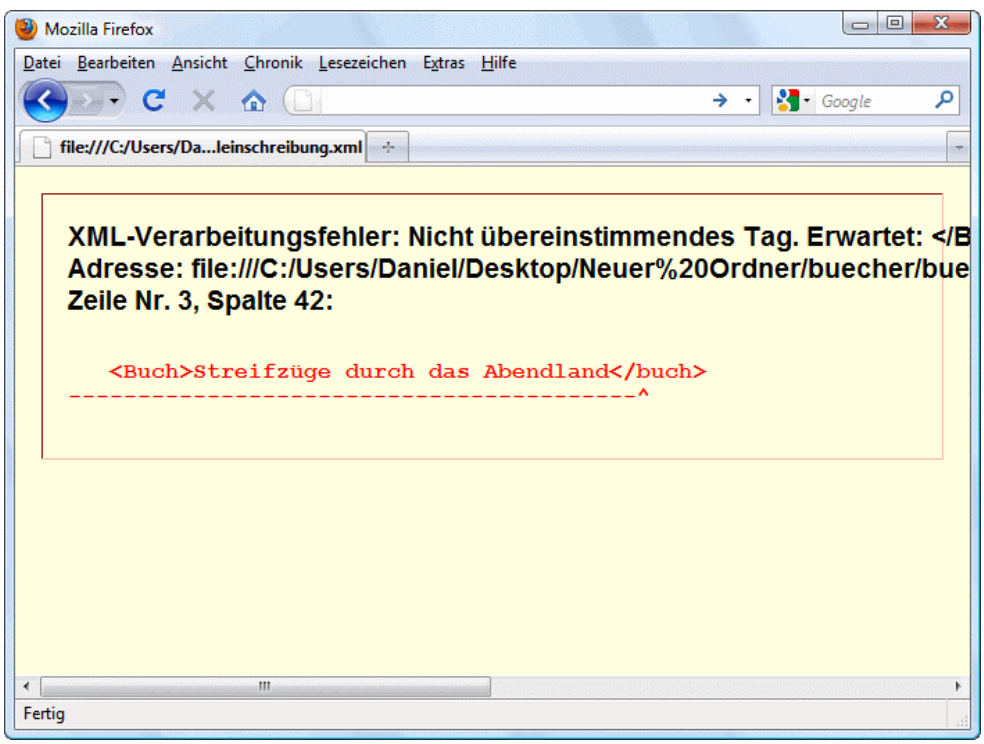

Abbildung 2.6 Hier wurde nicht auf Groß- und Kleinschreibung geachtet.

Sie sehen die Auswirkungen der Syntax. Der Firefox-Browser zeigt einen XML-Verarbeitungsfehler. Zusätzlich weist er darauf hin, dass ein nicht übereinstimmendes Tag enthalten ist. Und zwar steht als schließendes Tag hier `</buch>`, obwohl eigentlich `</Buch>` stehen müsste.

Zwischen Start- und dem End-Tag steht der Elementinhalt. In dem Beispiel

```
<buch>Streifzüge durch das Abendland</buch>
```

handelt es sich um eine normale Zeichenkette.

2.3.1 Leere Elemente kennzeichnen

In XML gibt es, anders als in HTML, keine alleinstehenden Elemente wie beispielsweise `br`. Erfahrene HTML-Entwickler, die sich bereits mit der XHTML-Spezifikation auseinandergesetzt haben, wissen, dass das `br`-Element in XHTML als `<br />` bzw. `<br></br>` notiert werden muss. Ähnlich verhält es sich in XML. Ein Beispiel:

```
<Name></Name>
```

Für diese Notation gibt es eine verkürzte Schreibweise. Der gleiche Effekt, nämlich die Kennzeichnung eines leeren Elements, wird auch mittels der nachfolgenden Variante erzielt:

```
<Name />
```

Um das `Name`-Element als leer zu kennzeichnen, wird vor der schließenden spitzen Klammer ein Schrägstrich notiert. Wie das Beispiel zeigt, wurde vor dem Schrägstrich ein Leerzeichen eingefügt. Dieses ist nicht zwingend erforderlich, hat sich aber als guter Stil etabliert.[2] Nicht nur, dass hierdurch die Übersichtlichkeit des Quelltextes erhöht wird, auch die Fehleranfälligkeit von XML-Parsern kann herabgesetzt werden. Könnte doch ein Parser bei dem Element `<Name/>` vermuten, dass der Schrägstrich ein ungültiges Zeichen ist. Durch das Einfügen eines Leerzeichens wird diese Gefahr gebannt.

2.3.2 Fehlerquelle verschachtelte Elemente

Ein Problem in XML bilden die sogenannten verschachtelten Elemente. Auch hierfür wieder ein Beispiel, und zwar eins aus HTML.

```
<b>Ich bin ein fetter Text</b>
```

Durch das `b`-Element wird der Elementinhalt fett angezeigt. Nun soll die Syntax dahingehend erweitert werden, dass das Wort `fett` zusätzlich kursiv erscheint.

```
<b>Ich bin ein <i>fetter</i> Text</b>
```

Dazu wird innerhalb des `b`- das `i`-Element notiert. Damit allerdings noch nicht genug. Die Syntax wird noch einmal erweitert. Und zwar soll das Wort `fett` jetzt auch noch unterstrichen angezeigt werden.

```
<b>Ich bin ein <i><u>fetter</u></i> Text</b>
```

Hierbei handelt es sich um Syntax, in der die Elemente korrekt verschachtelt wurden. Fehlerhaft hingegen ist die folgende Variante:

```
<b>Ich bin ein <i><u>fetter</i> </u>Text</b>
```

Hier wird das `u`-Element fälschlicherweise hinter dem schließenden `</i>`-Tag beendet. Korrekt ist hingegen: Das Element, das als letztes geöffnet wurde, muss als erstes wieder

[2] Letzten Endes ist das aber natürlich Geschmacksache. Fragen Sie zwei XML-Entwickler zu diesem Thema, und Sie werden drei verschiedene Antworten bekommen.

geschlossen werden. Nun würde in einem Browser die fehlerhafte HTML-Syntax nicht moniert werden. Das liegt daran, dass Browser fehlertolerant sind und sich an solchen „Kleinigkeiten“ nicht stören.

Anders sieht es in XML aus.

Listing 2.5 Diese Syntax ist korrekt.

```
<?xml version="1.0"?>
<buch>
   <autor>Ellis</autor>
</buch>
```

Wenn dieses Dokument im Browser aufgerufen wird, ergibt sich so etwas wie in **Abbildung 2.7**.

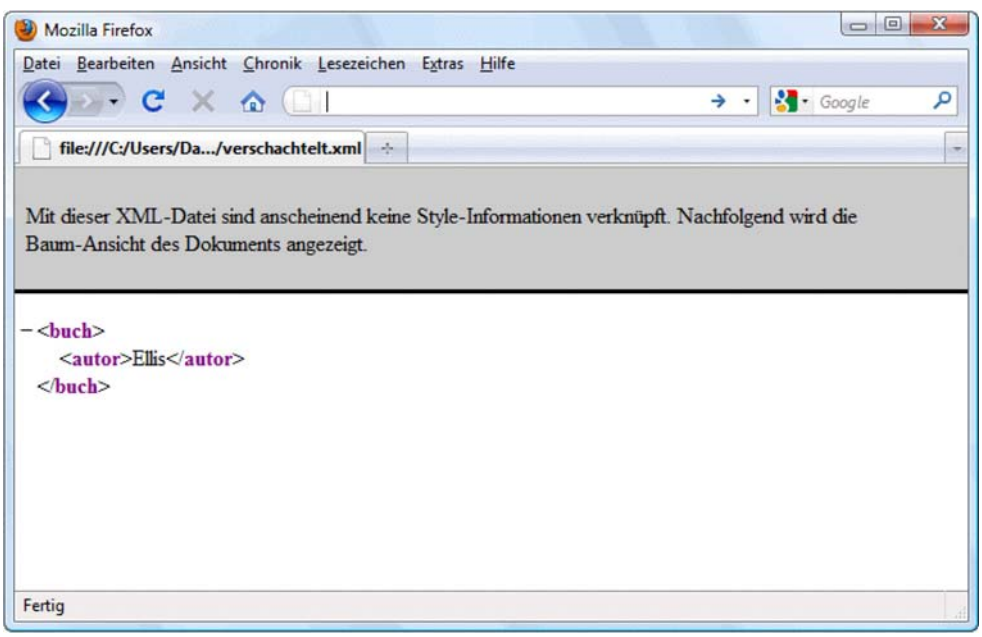

Abbildung 2.7 Der Firefox zeigt das Dokument korrekt an.

Es handelt sich um eine korrekte Syntax, die so vom Browser auch richtig interpretiert und angezeigt wird. Anders sieht es aus, wenn die Syntax fehlerhaft ist, die Elemente also falsch verschachtelt wurden.

Listing 2.6 Diese Syntax ist nicht korrekt.

```
<?xml version="1.0"?>
<buch>
   <autor>Ellis
</buch>
   </autor>
```

In diesem Beispiel wurde das schließende `</autor>`-Tag fälschlicherweise hinter das schließende `</buch>`-Tag eingefügt. Das genügt, um eine Fehlermeldung auszulösen.

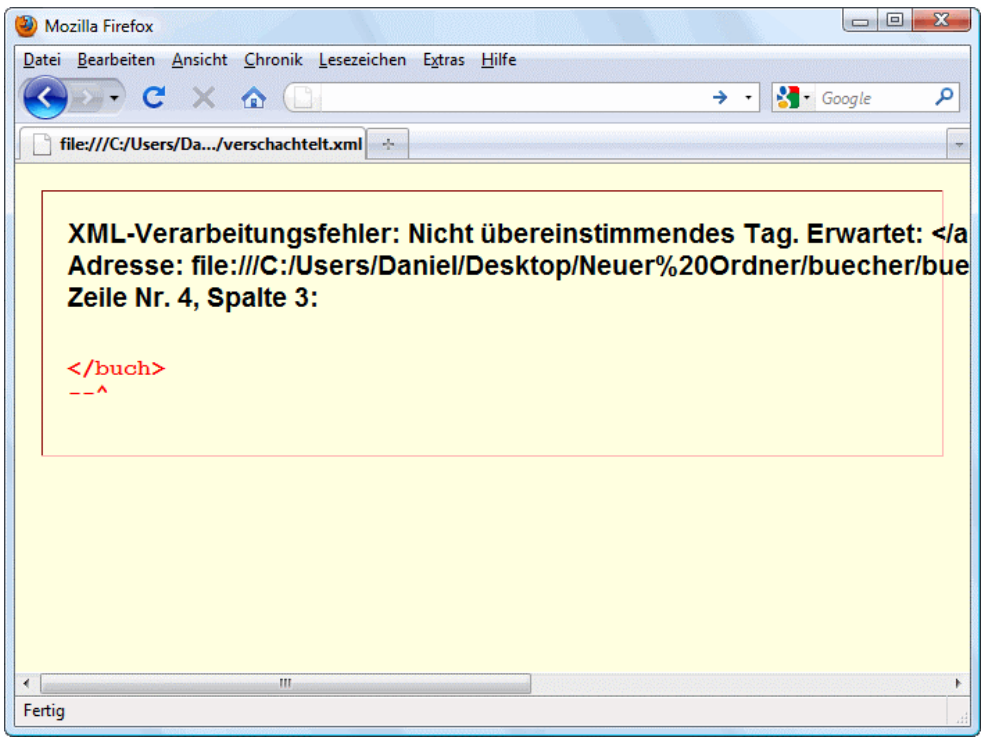

Abbildung 2.8 Jetzt wird ein Fehler generiert.

Wie erwartet, gibt der Browser einen Fehler aus. Er meldet ein nicht übereinstimmendes Tag. Richtigerweise wird anstelle des schließenden `</buch>`- das schließende `</autor>`-Tag erwartet.

In XML ist es wichtig, dass Sie auf die korrekte Schachtelung achten. Es muss immer das innere Element als erstes geschlossen werden.

2.3.3 Elemente mit Attributen detaillierter beschreiben

Alle Elemente, die in XML-Strukturen auftauchen, können mit beliebig vielen Attributen versehen werden. Sie kennen Attribute sicherlich aus HTML. Als Beispiel sei das Element `table` genannt. Dieses leitet in HTML-Dokumenten die Definition einer Tabelle ein. Um die Darstellung einer Tabelle zu beeinflussen, kennt das `table`-Element mehrere Attribute. Soll die Tabelle beispielsweise mit einem Rahmen versehen werden, wird das `border`-Attribut eingesetzt. Die Syntax `<table border>` reicht allerdings noch nicht aus. Erst wenn dem `border`-Attribut ein Attributwert, z.B. `1`, zugewiesen wird, kann der Rahmen dargestellt werden.

Die Verwendung von Attributen in XML zeigt das folgende Beispiel:

Listing 2.7 Der Autor ist männlich.

```
<?xml version="1.0"?>
<buch>
   <autor gs="maennlich">Ellis</autor>
</buch>
```

Innerhalb dieser Syntax wurde das `autor`-Element um das Attribut `gs`[3] erweitert, dem wiederum der Wert `maennlich` zugewiesen wurde. Jedes Attribut muss einen Attributwert besitzen. Die Notation von Attributen erfolgt immer auf die gleiche Art. Hinter dem Attributnamen steht ein Gleichheitszeichen, an das sich der Attributwert anschließt, der wiederum in Anführungszeichen gesetzt werden muss. Beachten Sie, dass die Anführungszeichen in XML Pflicht sind. Hier besteht ein gravierender Unterschied zu normalem HTML. Denn in HTML spielt bei der Zuweisung von Attributwerten das Setzen von Anführungszeichen keine Rolle.

Sie können einem Element mehrere Attribute zuweisen. Auch dazu wieder ein Beispiel:

Listing 2.8 Hier wurden gleich mehrere Attribute verwendet.

```
<?xml version="1.0"?>
<buch>
   <autor gs="maennlich" alter="49">Ellis</autor>
</buch>
```

Achten Sie darauf, dass die Attribute eindeutig sein müssen. Es darf nicht der gleiche Attributname innerhalb eines Start-Tags mehrfach verwendet werden. So etwas hier

Listing 2.9 Das wird nicht funktionieren.

```
<?xml version="1.0"?>
<buch>
   <autor gs="maennlich" gs="weiblich">Ellis</autor>
</buch>
```

wäre falsch und würde so natürlich auch überhaupt keinen Sinn machen, da sich die Attributwerte widersprechen.

[3] Wobei gs hier die Abkürzung für geschlecht ist.

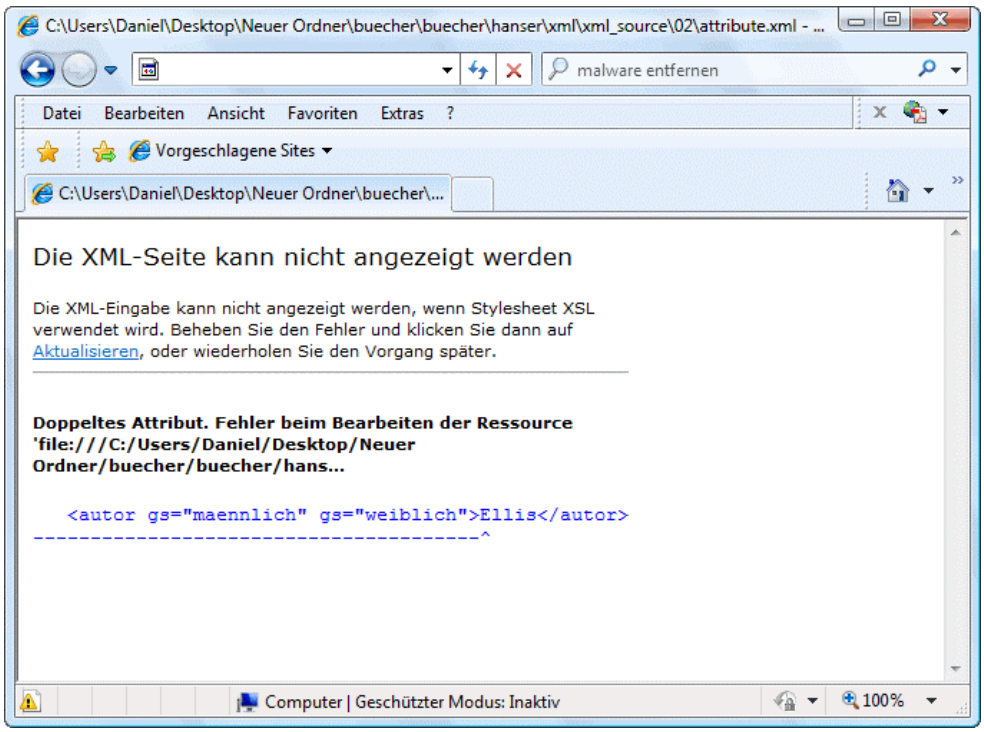

Abbildung 2.9 Auch der Internet Explorer mag dieses Dokument nicht.

Man kann aber den gleichen Attributnamen bei unterschiedlichen Elementen verwenden.

Listing 2.10 Diese Syntax ist korrekt.

```
<?xml version="1.0" ?>
<buch>
   <autor gs="maennlich">Ellis</autor>
   <lektor gs="weiblich">Michelle</lektor>
</buch>
```

Für den XML-Prozessor ist das Attribut unterscheidbar, da es verschiedenen Elementen zugewiesen wurde. Attribute können auch im Zusammenhang mit leeren Elementen verwendet werden. Dazu ebenfalls ein Beispiel:

Listing 2.11 Ein Attribut bei einem leeren Element

```
<?xml version="1.0"?>
<buch>
   <autor gs="maennlich">Ellis</autor>
   <kategorie fach="roman" />
</buch>
```

2.3.4 Was ist besser: Elemente oder Attribute?

Es existieren keine Regeln, ob Attribute oder Elemente verwendet werden sollen. Für Sie als Entwickler heißt das, dass Sie am Anfang die verschiedenen Varianten durchprobieren müssen. Andererseits sind Sie in der Entscheidungsfindung aber auch nicht an starre Vorgaben gebunden.

In der Praxis hat es sich durchgesetzt, Attribute zu verwenden, um Zusatzinformationen für Elemente anzugeben. Allerdings ist nicht immer sofort klar, welche Aspekte eines Dokuments beziehungsweise einer Datenstruktur man besser als ein Element oder als Attribut abbildet. Um das zu verdeutlichen, finden Sie nachfolgend zwei Beispieldokumente. Zunächst die Attributvariante:

Listing 2.12 Hier wurde auf Attribute gesetzt.

```
<buecher kategorie="roman">
   <titel>Die Firma</titel>
   <isbn>3-453-07117-4</isbn>
   <jahr>1990</jahr>
</buecher>
<buecher kategorie="sachbuch">
   <titel>PHP</titel>
   <isbn>3-8272-5524-4</isbn>
   <jahr>1998</jahr>
</buecher>
```

In dieser Syntax wurde, um die Bücher kategorisieren zu können, dem `buecher`-Element das Attribut `kategorie` zugewiesen. Die entsprechende Elementvariante könnte nun folgendermaßen aussehen:

Listing 2.13 Hier wurden Elemente verwendet.

```
<buecher>
   <kategorie>Roman</kategorie>
   <titel>Die Firma</titel>
   <isbn>3-453-07117-4</isbn>
   <jahr>1990</jahr>
</buecher>
<buecher>
   <kategorie>Sachbuch</kategorie>
   <titel>Die Firma</titel>
   <isbn>3-453-07117-4</isbn>
   <jahr>1990</jahr>
</buecher>
```

In diesem Beispiel wurde `kategorie` als ein Unterelement von `buecher` eingefügt. Im Unterschied zu Elementen können Attribute keine Unterattribute oder Unterelemente enthalten. Ein weiterer Nachteil von Attributen: Der Zugriff auf Attributwerte beispielsweise mittels DOM ist umständlicher als der auf Elemente.

Attribute haben aber auch Vorteile. So muss beispielsweise ihre Reihenfolge nicht wie bei Elementen fixiert werden.

2.3.5 Reservierte Attribute

Innerhalb der XML-Spezifikation gibt es zwei reservierte Attribute. Beide Attribute werden in diesem Abschnitt vorgestellt. Den Anfang macht `xml:lang`.

Listing 2.14 Hier wird das reservierte Attribut `xml:lang` verwendet.

```
<?xml version="1.0"?>
<buch>
   <gruss xml:lang="de">Halle, Welt!</gruss>
</buch>
```

Über dieses Attribut können Sie die Sprache angeben, die für den Elementinhalt verwendet wird. Im aktuellen Beispiel wird mit dem Attributwert `de` signalisiert, dass es sich um deutschsprachigen Inhalt handelt. Würde das Beispiel „multinational“ ausgebaut, könnte es folgendermaßen aussehen:

Listing 2.15 Verschiedene Sprachen werden gekennzeichnet.

```
<?xml version="1.0"?>
<buch>
   <gruss xml:lang="de">Hallo,Welt!</gruss>
   <gruss xml:lang="en">Hello,World!</gruss>
</buch>
```

Als Wert des Attributs `xml:lang` werden zweistellige Länderkürzel verwendet, die durch ISO 693-1 (*http://www.loc.gov/standards/iso639-2/php/code_list.php*) genormt sind. Man kann hier übrigens auch die sogenannten Subcodes für regionale Sprachen einsetzen. So können Sie also beispielsweise *en-US* oder *en-GB* verwenden.

Das zweite reservierte Attribut lässt sich wieder am besten anhand eines Beispiels vorstellen.

Listing 2.16 Leerzeichen im Text

```
<?xml version="1.0"?>
<buch>
   <gruss xml:space="default">H a l l o , W e l t !</gruss>
</buch>
```

Zwischen den einzelnen Zeichen der Zeichenkette `Hallo, Welt!` wurde jeweils ein Leerzeichen eingefügt. Über das reservierte Attribut `xml:space` können Sie versuchen, darauf Einfluss zu nehmen, wie ein XML-Prozessor mit Leerräumen innerhalb von XML-Elementinhalten umgehen soll. Bei Leerräumen handelt es sich um Leerzeichen wie im vorherigen Beispiel, aber auch Tabulatoren und Leerzeilen fallen in diese Kategorie. Als Werte des `xml:space`-Attributs sind erlaubt:

- `preverse` – Alle Leerräume sollen so erhalten bleiben, wie sie im Dokument vorkommen.
- `default` – Die Anwendung bzw. die Software kann die Leerräume so verwenden, wie es bei ihr voreingestellt ist.

Verlassen sollten Sie sich auf das Attribut `xml:space` allerdings nicht. Denn ob die entsprechende Anwendung den Anweisungen folgt, das können Sie im Vorfeld nicht wissen, sondern müssen es einfach ausprobieren.[4]

2.4 XML-eigene Zeichen maskieren: Entitäten

Sie können die speziellen Zeichen, die in XML für Markup[5] verwendet werden, nicht ohne Weiteres als Elementinhalt oder Attributwert einsetzen. Warum das so ist, zeigt folgendes Beispiel. Angenommen, Sie müssen ein XML-Dokument für ein Unternehmen anlegen.

Listing 2.17 Ob das mit dem Sonderzeichen gut geht?

Listing 2.18
```
<?xml version="1.0"?>
<unternehmen>
   <form>GmbH & Co. KG</form>
</unternehmen>
```

Innerhalb des Elements `form` wurde die Gesellschaftsform des Unternehmens angegeben. Das Dokument scheint auf den ersten Blick korrekt zu sein. Ruft man es allerdings im Browser auf, ergibt sich ein Anblick wie in **Abbildung 2.10**.

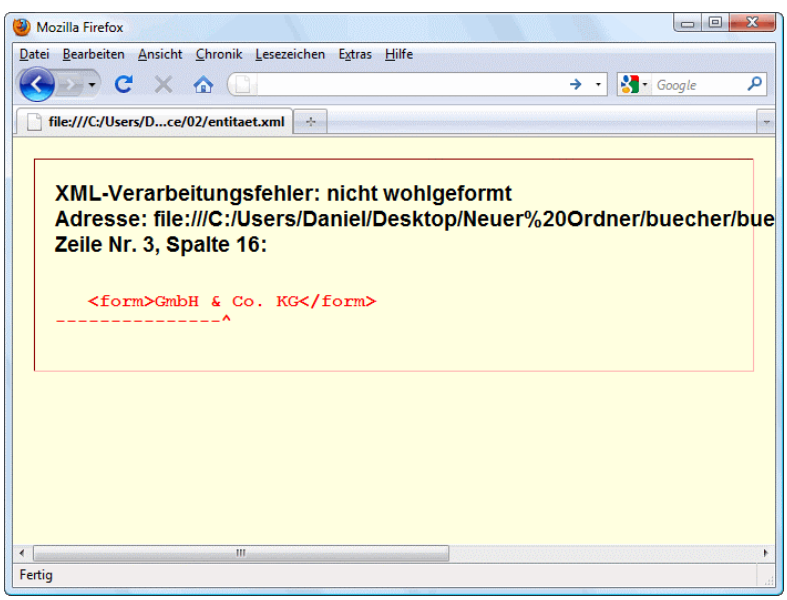

Abbildung 2.10 Der Internet Explorer gibt einen Fehler aus.

[4] Es sei denn, die Anwendung ist entsprechend gut dokumentiert.

[5] Eine kurze Begriffsdefinition: Bei einem Markup handelt es sich um eine Auszeichnung, eine Markierung.

Es stellt sich die Frage, warum der Browser einen Fehler ausgibt, obwohl das XML-Dokument scheinbar in Ordnung ist. Schuld an der Fehlermeldung ist das Zeichen &. Denn dieses Zeichen wird in XML normalerweise als Markup verwendet. Es gibt mehrere Möglichkeiten, Zeichen davor zu schützen, von einem XML-Prozessor als Markup ausgewertet zu werden.

Die eine Variante ist der Einsatz von Entitäten und das Verweisen auf diese Entitäten. Tabelle 2.3 zeigt die in XML verfügbaren Entitäten:

Tabelle 2.3: Die fünf Standard-XML-Entitäten

Entität	Entitätenreferenz	Beschreibung
Lt	<	< (kleiner als)
Gt	>	> (größer als)
amp	&	& (kaufmännisches UND)
apos	'	' (Apostroph oder einfaches Anführungszeichen)
quot	"	" (Doppelte Anführungszeichen)

Das eingangs gezeigte Beispiel kann entsprechend angepasst werden und sieht dann folgendermaßen aus:

Listing 2.19 Das Sonderzeichen wurde maskiert.

```
<?xml version="1.0"?>
<unternehmen>
   <form>GmbH &amp; Co. KG</form>
</unternehmen>
```

Wenn Sie dieses Dokument im Browser aufrufen, wird es richtig angezeigt.

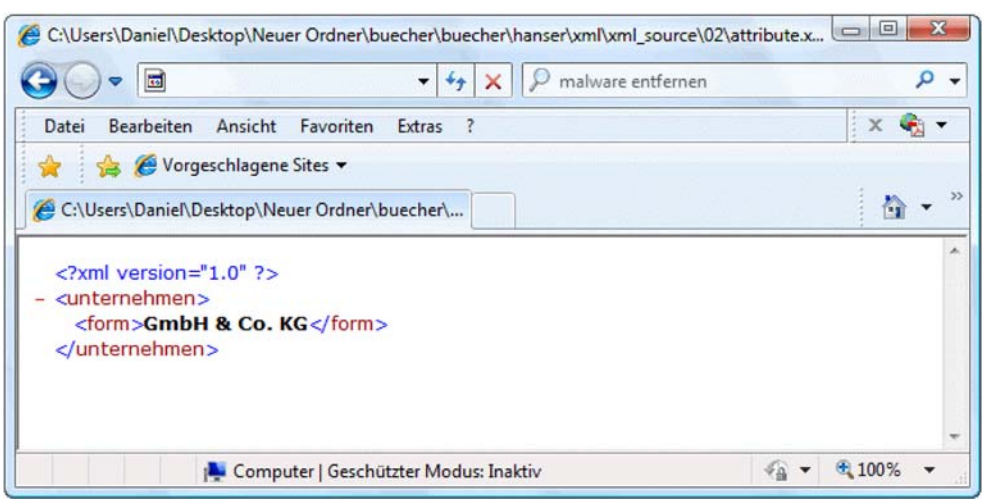

Abbildung 2.11 Jetzt stimmt die Anzeige.

Neben diesen in XML fest verankerten Entitäten können Sie auch eigene definieren. Das funktioniert ähnlich wie in HTML und kann z.B. innerhalb einer Dokumenttypdefinition (DTD) geschehen. Dazu dann aber später mehr. An dieser Stelle soll es zunächst darum gehen, wie Sie solche innerhalb einer DTD definierten Entitäten verwenden können. Angenommen, Sie haben in einer DTD anstelle von `GmbH` das Kürzel `GH` abgelegt. Dann kann der Ersetzungstext wie folgt verwendet werden:

Listing 2.20 So geht es auch.

```
<?xml version="1.0"?>
<unternehmen>
   <form>&GH;</form>
</unternehmen>
```

Um die in einer DTD definierten Kürzel zu verwenden, wird zunächst das kaufmännische UND gefolgt von dem Kürzel und einem abschließenden Semikolon notiert. (Beachten Sie, dass das gezeigte Beispiel in der jetzigen Form noch nicht lauffähig ist, da die entsprechende DTD fehlt.)

Eine weitere Möglichkeit, um solche Zeichen darzustellen, die sonst nicht zur Verfügung stehen, bieten die Zeichenreferenzen. Diese werden mithilfe von dezimalen oder hexadezimalen Zahlen definiert. Diese Zahlen verweisen auf den Unicode-Zeichensatz. Auch hierzu wieder ein Beispiel, durch das ein umgekehrtes Ausrufezeichen angezeigt werden soll.

Listing 2.21 So wird Unicode eingesetzt.

```
<?xml version="1.0"?>
<zeichen>
   <ausgabe>&#161;</ausgabe>
</zeichen>
```

Die Syntax ist fast mit der Schreibweise bei Entitätenreferenzen identisch. Einziger Unterschied: Dem `&` wird noch ein `#`-Zeichen angehängt. Unter *http://de.selfhtml.org/html/referenz/zeichen.htm* finden Sie eine Liste der verfügbaren Zeichen.

Wenn Entitätenreferenzen verwendet werden, muss das Dokument nach der Auflösung der Referenz weiterhin wohlgeformt sein. (Mehr zum Thema Wohlgeformtheit dann übrigens im weiteren Verlauf dieses Kapitels.) Aus diesem Grund sind Entitätenreferenzen auch nur deshalb erlaubt, weil in ihnen ein Bezug auf analysierte Daten (parsed data[6]) enthalten ist. Ein direkter Bezug auf ungeparste Daten (unparsed data) ist hingegen nicht möglich. Solche Bezüge lassen sich über Attributwerte vom Typ `ENTITY` oder `ENTITIES` herstellen. Auch zu diesem Aspekt dann später mehr.

[6] Analysierte Daten bestehen aus Zeichen, von denen einige Zeichenketten und andere Markup darstellen.

2.5 Zeit sparen mit CDATA-Abschnitten

Werden innerhalb eines Dokuments besonders viele Markup-Zeichen benötigt, können CDATA-Abschnitte (*Akronym für Character Data*, Zeichendaten) das richtige Mittel sein. Ein Beispiel:

Listing 2.22 Hier würde CDATA gut tun.

```
<?xml version="1.0"?>
<zeichenvorrat>
   <zeichen><<<<<<< Hallo, Welt! >>>>>>>></zeichen>
</zeichenvorrat>
```

In diesem Beispiel müssten Sie normalerweise die Zeichen < und > durch die entsprechenden Entitäten ersetzen. Wenn vergleichsweise viele solcher Zeichen ersetzt werden sollen, wird das zu aufwendig. Und wenn die Struktur des XML-Dokuments komplexer wird und Sie beispielsweise ganze Skripte für einen anderen Entwickler abbilden wollen, müssten Sie den gesamten Code mittels der Entitäten auszeichnen, was viel zu aufwendig ist. In solchen Fällen ist es praktischer, sogenannte CDATA-Abschnitte bzw. CDATA-Sections einzusetzen. CDATA-Abschnitte werden dazu verwendet, größere Textpassagen als reinen Text darzustellen. Bei solchen Texten versucht der Parser erst gar nicht, Markup-Informationen aus dem Inhalt des CDATA-Abschnitts zu extrahieren.

Das vorherige Beispiel auf CDATA-Syntax umgemünzt sieht folgendermaßen aus:

Listing 2.23 Ein CDATA-Abschnitt wurde definiert.

```
<?xml version="1.0"?>
<zeichenvorrat>
   <zeichen>
   <![CDATA[
      <<<<<<< Hallo, Welt! >>>>>>>>
   ]]>
    </zeichen>
</zeichenvorrat>
```

Wird dieses Dokument im Browser aufgerufen, werden die darin enthaltenen Zeichen exakt so angezeigt, wie sie im Quelltext stehen.

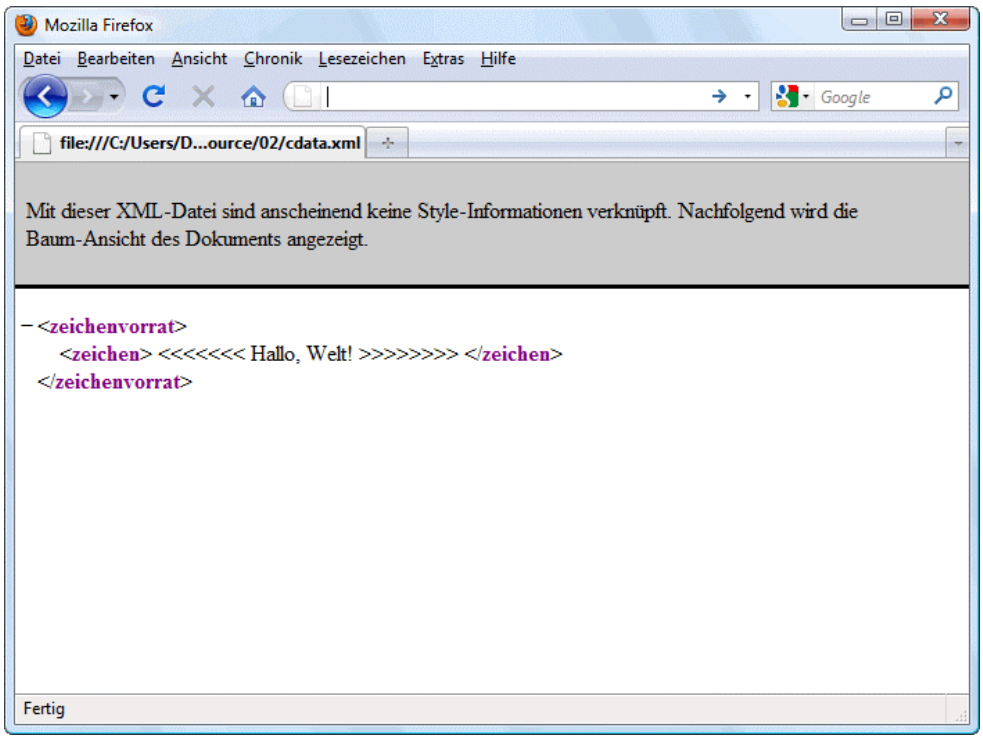

Abbildung 2.12 So werden die Zeichen auch ohne zusätzliche Maskierung korrekt angezeigt.

Sämtliche Zeichen, die sich innerhalb des CDATA-Abschnitts befinden, werden also nicht als Markup interpretiert, sondern so, wie sie im Code stehen, angezeigt. Sie können innerhalb eines solchen Abschnitts beliebige Zeichen einsetzen. Eine Einschränkung gibt es aber auch hier. Die Zeichenfolge

```
]]>
```

selbst darf innerhalb von CDATA nicht vorkommen, da hierdurch der CDATA-Abschnitt beendet wird. Soll die Zeichenfolge `]]>` dennoch dargestellt werden, teilt man üblicherweise den CDATA-Abschnitt in mehrere Teile auf und trennt dabei die Zeichenfolge vor dem `>`.

```
<![CDATA[Inhalt]]]>
<![CDATA[>Inhalt]]>
```

Auf diese Weise werden Syntaxfehler vermieden.

2.6 Kommentare für übersichtlichen Code

In Programmiersprachen dienen Kommentare der besseren Übersichtlichkeit von Programmcode. Das ist in XML ebenfalls so. Auch wenn man in XML natürlich sagen muss, dass die „sprechenden“ Elemente von XML-Dokumenten zu einem großen Teil oft selbstdokumentierend sind.[7]

Sie können in XML ein- und mehrzeilige Kommentare verwenden. Wie das funktioniert, zeigt das folgende Beispiel:

Listing 2.24 So blickt man durch.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!--
Der Aufbau einer umfangreichen
XML-Datei kann durch Kommentare
vereinfacht werden.
-->
<bestand>
<!-- Die Daten des ersten Buches -->

<buch>
   <Name>Streifzüge durch das Abendland</Name>
   <Preis>8 Euro</Preis>
   <Kategorie>Reisen</Kategorie>
</buch>
<!-- Die Daten des zweiten Buches -->

<buch>
   <Name>Mein Leben als Mann</Name>
   <Preis>9 Euro</Preis>
   <Kategorie>Belletristik</Kategorie>
</buch>

</bestand>
```

Genauso wie in HTML werden Kommentare in XML mit `<!` eingeleitet und enden auf `-->`. Bis auf die Zeichenfolge `--` dürfen sämtliche Zeichen innerhalb von Kommentaren verwendet werden.

Interessant sind Kommentare vor allem, wenn es darum geht, einzelne Elemente oder sogar ganze Elementblöcke auszukommentieren. So müssen Sie momentan überflüssige Elemente nicht sofort löschen, sondern weisen diese als Kommentare aus. Wenn Sie die Elemente später wieder benötigen, entfernen Sie die Kommentarzeichen wieder.

Dem XML-Parser ist es freigestellt, ob er die Kommentare für ein Programm zugänglich macht. So zeigen beispielsweise sowohl der Internet Explorer wie auch Firefox die Kommentare an.

[7] Wobei das letztendlich von der Wahl der Elementnamen abhängt.

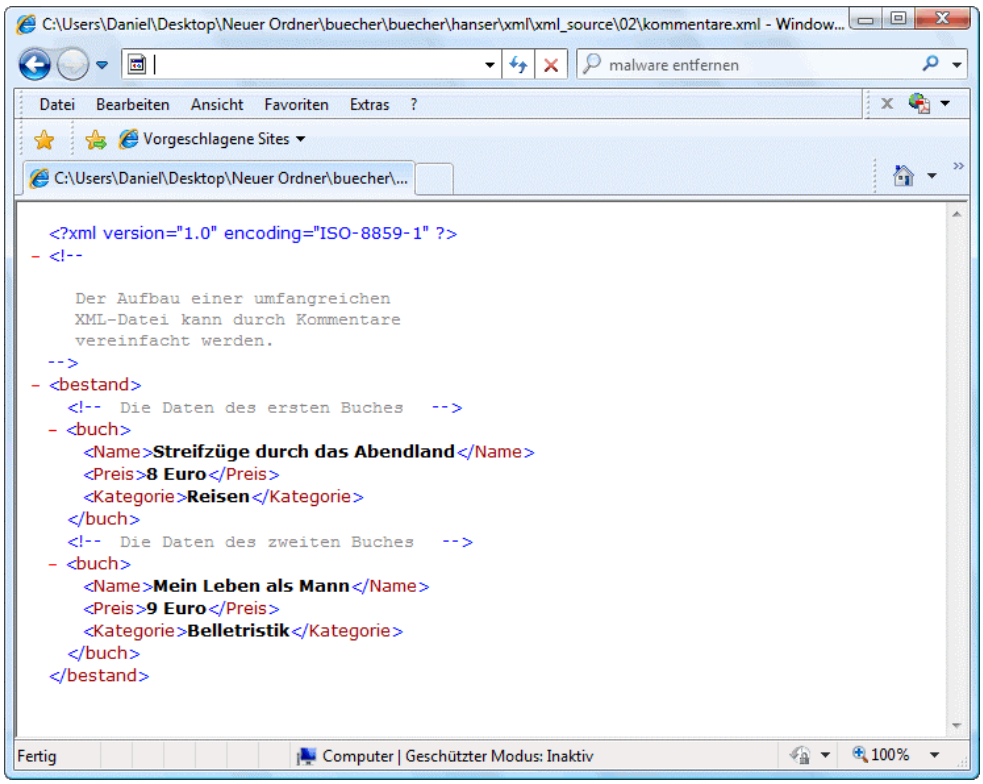

Abbildung 2.13 Kommentare werden auch in der Baumansicht angezeigt.

Im Firefox und im Internet Explorer erkennt man deutlich, dass es sich um Kommentare handelt. Noch ein wichtiger Hinweis in diesem Zusammenhang Es ist unbedingt darauf zu achten, dass Kommentare nicht vor der XML-Deklaration stehen dürfen, wenn sie eine verwenden. Wenn Sie hingegen keine XML-Deklaration angeben, darf das Dokument mit einem Kommentar beginnen.

2.7 Verarbeitungsanweisungen (Processing Instructions)

Sie können innerhalb von XML-Dokumenten Verarbeitungsanweisungen – die sogenannten Processing Instructions – definieren, die Anweisungen für den XML-Prozessor enthalten. Der Inhalt von Verarbeitungsanweisungen gehört nicht zu den Zeichendaten, aus denen das Dokument besteht. Verarbeitungsanweisungen werden über die Zeichenfolge `<?` eingeleitet und enden auf `?>`.

```
<?xml-stylesheet type="text/css" href="ausgabe.css"?>
```

Hierbei handelt es sich um eine Verarbeitungsanweisung für ein externes Stylesheet. Innerhalb des angegebenen Dokuments *ausgabe.css* sind die Formatierungsanweisungen für das aktuelle Dokument enthalten.

Welche Anweisungen innerhalb von `<? ?>` möglich sind, ist in der XML-Spezifikation nicht beschrieben. Das hängt ausschließlich vom verwendeten Prozessor ab und was dieser für Verarbeitungsanweisungen versteht.

Beachten Sie, dass Verarbeitungsanweisungen nicht nur für das XML-Dokument selbst angegeben werden, sondern auch mitten im XML-Dokument stehen und von anderen Technologien genutzt werden können. Ein typisches Beispiel dafür könnte der Einsatz von PHP sein.

Listing 2.25 PHP und XML im Zusammenspiel

```
<?xml version="1.0"?>
<zeichen>
   <ausgabe>
      <?php echo 'Hallo Welt'; ?>
   </ausgabe>
</zeichen>
```

Auch hierbei handelt es sich um eine Verarbeitungsanweisung. Die Verarbeitung von Inhalten wird sofort beendet, wenn die Zeichenfolge `?>` ermittelt wird.

2.8 Namensräume definieren

In diesem Abschnitt geht es um die sogenannten Namensräume. Um den Einstieg in diese Thematik zu erleichtern, bietet es sich an, sich das Ganze anhand eines Beispiels anzusehen.

Listing 2.26 Ein einfaches XML-Dokument

```
<?xml version="1.0"?>
<kontakte>
   <kontakt id="1">
      <vorname>Michael</vorname>
      <nachname>Mayer</nachname>
      <telefon nummer="232323" />
   </kontakt>
</kontakte>
```

Sie sehen ein typisches XML-Dokument. Dieses Dokument enthält unter anderem das `kontakte`-Element. Innerhalb dieses Elements befindet sich ein `kontakt`-Element, dem die ID `1` zugewiesen wurde. `kontakt` wiederum besitzt verschiedene, den jeweiligen Kontakt beschreibende Elemente wie beispielsweise `telefon`.

Parallel zu diesem gibt es noch ein weiteres XML-Dokument, bei dem es sich um eine – wenn auch kleine – Kundendatenbank handelt.

Listing 2.27 Und noch eine Datenbank.

```
<?xml version="1.0"?>
<kunden>
   <unternehmen id="1">
      <name>Hanser</name>
      <telefon>
         <vorwahl>1223</vorwahl>
```

```
            <nummer>345345345</nummer>
            <voiceoverip>Ja</voiceoverip>
         </telefon>
      </unternehmen>
   </kunden>
```

In dieser Datenbank stehen die entsprechenden Unternehmen. (Auch wenn es im aktuellen Beispiel zunächst einmal nur eins ist.) Für jedes Unternehmen existiert ein `telefon`-Element, das in einem anderen Kontext als im zuvor gezeigten Beispiel steht. Solange die beiden Dokumente voneinander getrennt sind, spielt es keine Rolle, dass in beiden XML-Strukturen jeweils ein `telefon`-Element vorhanden ist. Was aber passiert – und eben das ist in der Praxis gar nicht mal so selten –, wenn diese beiden Dokumente zusammengeführt werden sollen? In solchen Fällen kommt es dann aufgrund der gleichnamigen Elemente zu Problemen. Vermeiden lassen sich diese durch das Konzept der Namensräume. Mit einem solchen Namensraum werden die Elemente spezifiziert. Wie das funktioniert, wird anhand der beiden gezeigten Dokumente demonstriert, die zunächst einmal zusammengeführt werden.

Listing 2.28 Zwei Dokumente wurden zusammengewürfelt.

```
<?xml version="1.0"?>
<kontakte>
   <kontakt id="1">
      <vorname>Michael</vorname>
      <nachname>Mayer</nachname>
      <telefon nummer="232323" />
   </kontakt>
   <unternehmen id="1">
      <name>Hanser</name>
      <telefon>
         <vorwahl>1223</vorwahl>
         <nummer>345345345</nummer>
         <voiceoverip>Ja</voiceoverip>
      </telefon>
   </unternehmen>
</kontakte>
```

Das Zusammensetzen der beiden Dokumente führt zu einem Problem.[8] Denn innerhalb der XML-Struktur existieren jetzt zwei `telefon`-Elemente. Diese besitzen zwar den gleichen Namen, sind allerdings für völlig andere Einsatzzwecke gedacht. Während das eine lediglich die Telefonnummer enthält, werden im zweiten `telefon`-Element ausführliche Informationen zusammengefasst. Um der auslesenden Software mitzuteilen, dass es sich bei den beiden `telefon`-Elementen um zwei unterschiedliche handelt, kann ein Namensraum verwendet werden. Im aktuellen Beispiel wird dieser Namensraum dem `unternehmen`-Element zugewiesen, da dieses das zweite Telefonelement umschließt.

Listing 2.29 Und jetzt mit Namensräumen

```
<?xml version="1.0"?>
<kontakte>
   <kontakt id="1">
      <vorname>Michael</vorname>
```

[8] Damit es nicht zu Missverständnissen kommt. Syntaktisch gesehen ist das Dokument korrekt und wohlgeformt.

```
      <nachname>Mayer</nachname>
      <telefon nummer="232323" />
   </kontakt>
   <unternehmen id="1" xmlns="http://www.hanser.de/">
      <name>Hanser</name>
      <telefon>
         <vorwahl>1223</vorwahl>
         <nummer>345345345</nummer>
         <voiceoverip>Ja</voiceoverip>
      </telefon>
      </unternehmen>
</kontakte>
```

Namensräume werden durch `xmlns` definiert. Beachten Sie, dass es sich bei dem Namensraum im aktuellen Beispiel um einen URI handelt. Die angegebene Adresse muss nicht unbedingt existieren. Es ist zudem nicht Pflicht, für Namensräume URIs zu verwenden. Allerdings bietet es sich an, genau das zu tun, denn nur durch URIs können eindeutige Namensräume definiert werden. Bekanntermaßen sind URIs, die als Domainnamen existieren, immer eindeutig.[9] Prinzipiell kann zwar jede Art von Zeichenfolge eindeutig sein, dann muss aber jemand für die Eindeutigkeit sorgen. Und das ist bei normalen Zeichenketten eben normalerweise nicht gegeben. Der das Dokument verarbeitende Prozessor sieht übrigens nicht bei dem angegebenen URI nach, ob dort tatsächlich ein Namensraum abgelegt ist. URIs kommen also hauptsächlich deswegen zum Einsatz, weil sie eindeutig sind und sich gut merken lassen.

2.8.1 Den Standardnamensraum angeben

Ein Namensraum wird über das Attribut `xmlns` deklariert. Auf diese Weise wird einem Element ein Standardnamensraum zugewiesen. Alle Elemente, innerhalb des mit diesem Attribut ausgestatteten Elements, gehören zu diesem Namensraum.

Listing 2.30 Der Namensraum wurde angegeben.

```
<?xml version="1.0"?>
<kunden xmlns="http://www.hanser.de/">
   <unternehmen id="1">
      <name>Hanser</name>
      <telefon>
         <vorwahl>1223</vorwahl>
         <nummer>345345345</nummer>
         <voiceoverip>Ja</voiceoverip>
      </telefon>
   </unternehmen>
</kunden>
```

In diesem Beispiel ist der angegebene Namensraum `http://www.hanser.de/` für alle Elemente, die innerhalb von `kunden` stehen, gültig. Will man aber z.B. dem `telefon`- und dem ihm zugehörenden Elementen einen anderen Namensraum zuweisen, dann ist das möglich, indem man den globalen Namensraum überschreibt.

[9] Anders sähe es aus, wenn Sie als Namensraum `telefon` angeben würden. Dieser Namensraum wäre mit Sicherheit nicht eindeutig, da Sie oder andere Entwickler diesen Namensraum auch in einem anderen Zusammenhang verwenden könnten.

Listing 2.31 So lassen sich Namensräume überschreiben.

```
<?xml version="1.0"?>
<kunden xmlns="http://www.hanser.de/">
   <unternehmen id="1">
      <name>Hanser</name>
      <telefon xmlns="http://www.hanser-verlag.de/">
         <vorwahl>1223</vorwahl>
         <nummer>345345345</nummer>
         <voiceoverip>Ja</voiceoverip>
      </telefon>
   </unternehmen>
</kunden>
```

Dazu geben Sie innerhalb des `telefon`-Elements das Attribut `xmlns` an, weisen diesem als Wert aber einen anderen Namensraum zu. Für das Dokument bedeutet dies, dass alle Elemente, die innerhalb von `telefon` stehen, zu dem Namensraum `http://www.hanser-verlag.de/` gehören. Alle anderen Elemente gehören hingegen zu dem globalen Namensraum `http://www.hanser.de/`.

2.8.2 Das Namensraumpräfix

Häufig müssen innerhalb eines Elements Unterelemente aus verschiedenen Namensräumen miteinander kombiniert werden. Für eine solche flexible Zuordnung von Elementen und Unterelementen zu einem Namensraum setzt man die sogenannten Namensraumpräfixe ein. Auch dazu wieder ein Beispiel:

Listing 2.32 Das Präfix wurde gesetzt.

```
<?xml version="1.0"?>
<kunden xmlns:hv=http://www.hanser-verlag.de/
        xmlns:hr="http://www.hanser.de/">
   <unternehmen id="1">
      <name>Hanser</name>
      <hv:telefon>
         <vorwahl>1223</vorwahl>
         <hr:nummer>345345345</hr:nummer>
         <voiceoverip>Ja</voiceoverip>
      </hv:telefon>
   </unternehmen>
</kunden>
```

Hier wird zunächst ein Standardnamensraum `http://www.hanser-verlag.de/` angegeben. Dem wird das Präfix `hv` zugewiesen. Parallel dazu gibt es den Namensraum `http://www.hanser.de/`, der das Präfix `hr` erhält. Das Präfix schließt sich an `xmlns` hinter einem Doppelpunkt an. Um ein auf diese Weise definiertes Präfix verwenden zu können, wird dieses innerhalb des betreffenden Elements vor dem Elementnamen notiert.

Durch Präfixe lassen sich auch Attribute Namensräumen zuordnen.

```
<name hr:unternehmenid="gmbh">Hanser</name>
```

Auch dabei wird die bekannte Syntax mit dem Doppelpunkt verwendet. Es besteht zusätzlich die Möglichkeit, einen Standardnamensraum mit Namensraumpräfixen für bestimmte Attribute oder Elemente zu verbinden. Ein Beispiel:

Listing 2.33 Mehrere Präfixe im Einsatz.

```
<?xml version="1.0"?>
<kunden xmlns="http://www.hanser.de/"
        xmlns:hv="http://www.hanser-verlag.de/">
   <unternehmen id="1">
      <name>Hanser</name>
      <telefon>
         <vorwahl>1223</vorwahl>
         <hv:nummer>345345345</hv:nummer>
         <voiceoverip>Ja</voiceoverip>
      </telefon>
   </unternehmen>
</kunden>
```

In diesem Beispiel sind alle Elemente ohne das Präfix `hv` dem Standardnamensraum `http://www.hanser-verlag.de/` zugeordnet.
Dem Namensraum `http://www.hanser.de/` wurde lediglich das Element `hv:nummer` zugewiesen.

2.9 Das Prinzip der Wohlgeformtheit

Wenn Sie mit XML-Dokumenten arbeiten, sind zwei Dinge besonders wichtig: Sie müssen zunächst einmal auf die Wohlgeformtheit des XML-Dokuments achten. Das bedeutet, dass das Dokument allen XML-Regeln für Elemente und Attribute folgt. Zusätzlich sollte darauf geachtet werden, dass das Dokument valide, also korrekt strukturiert und nur die innerhalb der Struktur erlaubten Elemente und Attribute verwendet werden. Wenn XML-Dokumente weiterverarbeitet werden, werden sie geparst. Der dafür zuständige Parser überprüft in jedem Fall das XML-Dokument auf seine Wohlgeformtheit. Was es damit auf sich hat, zeigt das folgende Beispiel:

Listing 2.34 Dieses Beispiel ist korrekt.

```
<?xml version="1.0"?>
<kunden>
   <unternehmen id="1">
      <name>Hanser</name>
      <telefon>
         <vorwahl>1223</vorwahl>
         <nummer>345345345</nummer>
         <voiceoverip>Ja</voiceoverip>
      </telefon>
   </unternehmen>
</kunden>
```

Es handelt sich hierbei um ein normales XML-Dokument, das so auch korrekt im Browser angezeigt wird.

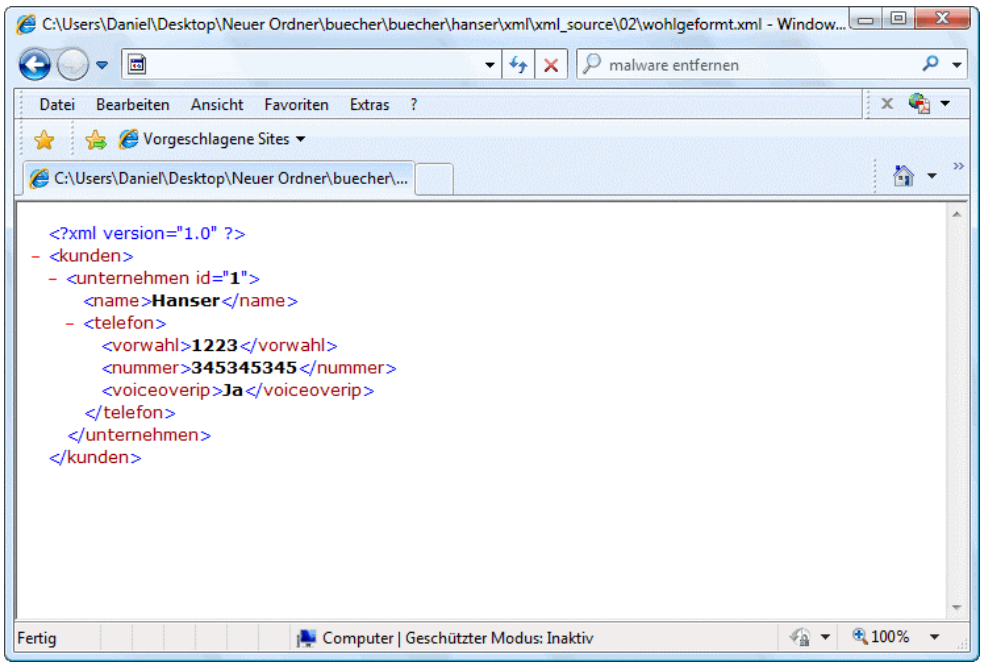

Abbildung 2.14 Ein wohlgeformtes Dokument

Dieses Dokument ist wohlgeformt. Was passiert aber, wenn es das nicht wäre? Dafür baue ich in das folgende Dokument einen Fehler ein. Und zwar wird anstelle des schließenden `<telefon>`-Tags `</telefon1>` notiert. Das ist ein Tippfehler, wie er immer mal wieder passiert.

Listing 2.35 Hier stimmt was nicht.

```
<?xml version="1.0"?>
<kunden>
   <unternehmen id="1">
      <name>Hanser</name>
      <telefon>
         <vorwahl>1223</vorwahl>
         <nummer>345345345</nummer>
         <voiceoverip>Ja</voiceoverip>
      </telefon1>
   </unternehmen>
</kunden>
```

Dadurch, dass das `telefon`-Element nicht korrekt geschlossen wird, führt die Anzeige des Dokuments im Browser zu einem Fehler.

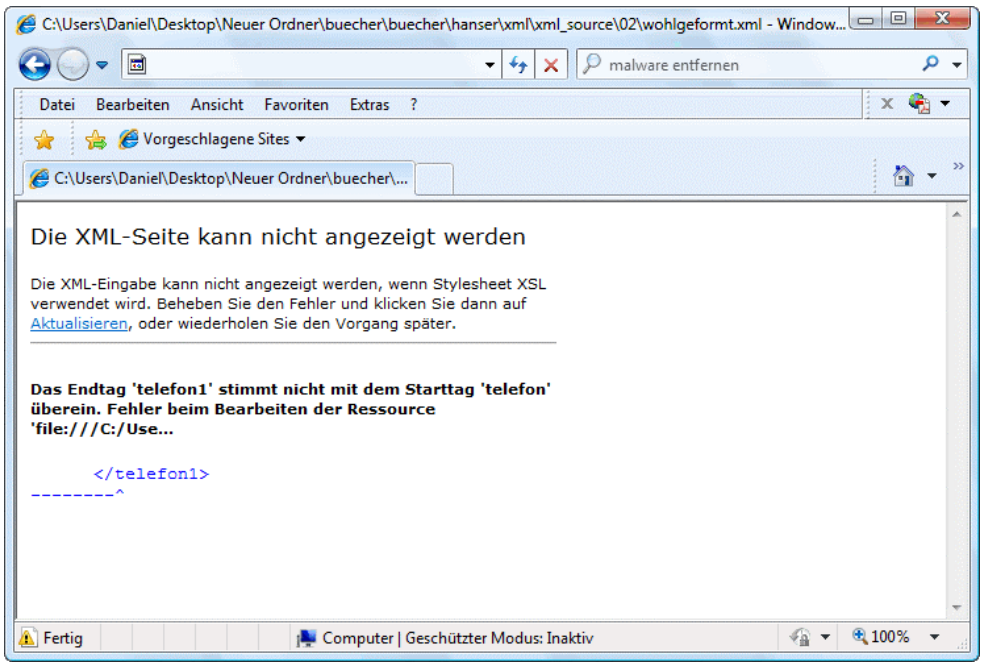

Abbildung 2.15 Dieses Dokument ist fehlerhaft.

Der Parser bzw. Browser erkennt, dass das schließende `</telefon1>`- nicht mit dem öffnenden `<telefon>`-Tag übereinstimmt. Das Dokument ist also nicht wohlgeformt, und es wird eine entsprechende Fehlermeldung ausgegeben. Das gilt übrigens nicht nur für den Browser, sondern auch für XML-Validatoren.

```
The name in the end tag of the element must match the element type in the
start tag.
```

Sowohl die XML-Standardisierungsgremien wie auch die XML-Entwicklergemeinde legen sehr großen Wert darauf, dass XML-Prozessoren auf Verstöße gegen die Wohlgeformtheitsregeln mit null Toleranz reagieren. Wie rigide die Prozessoren dabei vorgehen, wurde im vorherigen Beispiel deutlich, als der Browser einen „kleinen“ Tippfehler monierte. Wenn Sie im Vergleich dazu an HTML denken, sieht das dort ganz anders aus. Wird bei HTML ein Element nicht korrekt geschlossen, wird die Webseite trotzdem noch korrekt angezeigt.

Nun stellt sich die Frage, was denn eigentlich ein wohlgeformtes XML-Dokument ausmacht.

- Das Dokument muss mindestens ein Element, nämlich das Wurzelelement, enthalten. Dieses Element muss auf der obersten Ebene alleine stehen.
- Alle Elemente müssen geschlossen werden.
- Elemente müssen korrekt verschachtelt werden.

- XML unterscheidet zwischen Groß- und Kleinschreibung. Die Elemente `hanser` und `HANSER` sind also etwas Grundverschiedenes.
- Attributwerte müssen immer in Anführungszeichen gesetzt werden.
- Attribute dürfen innerhalb eines Elements nur einmal vorkommen.

Die Bezeichnung *Wohlgeformtheit* wird also hauptsächlich dafür verwendet, syntaktisch korrekte XML-Dokumente von sogenannten gültigen oder validen Dokumenten zu unterscheiden. Von einem gültigen Dokument wird gesprochen, wenn Dokumenttypen zum Einsatz kommen, in denen festgelegt ist, welche Elemente und Attribute in dem Dokument vorkommen dürfen. Sie kennen diesen Aspekt sicherlich aus HTML und XHTML. So ist in der XHTML-Spezifikation und deren DTD (Document Type Definition) festgelegt, welche Elemente und Attribute innerhalb einer XHTML-Datei verwendet werden dürfen. Bei der Validierung eines XHTML-Dokuments wird dann überprüft, ob es den Regeln für einen Dokumenttyp entspricht. Standardmäßig werden Dokumenttypen in Dokumenttypdefinitionen, also den DTDs, festgelegt. Als eine angemessene Alternative zu den DTDs hat sich inzwischen der XML-Standard XML Schema herauskristallisiert. Mehr zu DTDs und XML Schema dann im weiteren Verlauf dieses Buches.

3

3 Dokumenttypen beschreiben

Auf den folgenden Seiten dreht sich alles um die Dokumenttypdefinitionen, die DTDs. Bevor ins Detail gegangen wird, ein erste Blick auf die Frage, warum es Probleme gibt, wenn keine DTD vorhanden ist. Denn schließlich kann man XML-Dokumente auch schreiben, ohne dass eine DTD definiert wurde. Allerdings kommt es ohne DTD bei der Gestaltung von XML-Dokumenten schnell zu unschönen Auswüchsen, und es herrscht rasch Elementchaos, da eben nicht klar ist, welche Elemente und Attribute innerhalb der XML-Datei verwendet werden dürfen. Hier kommen die DTDs ins Spiel. Bei denen handelt es sich um nichts anderes als eine Beschreibung der Struktur des XML-Dokumenttyps.

3.1 Dokumenttypdefinitionen

Dokumenttypdefinitionen beschreiben die Regeln für erlaubte Elemente, Attribute usw., die auf eine Gruppe von XML-Dokumenten angewendet werden können. Wie so eine Dokumenttypdefinition aussieht, lässt sich sehr schön anhand von XHTML zeigen. Wenn Sie bereits XHTML-Dokumente angelegt haben, dann ist Ihnen das `DOCTYPE`-Element vertraut.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

Innerhalb dieses Elements finden Sie die Adresse der XHTML-DTD. Öffnen Sie die angegebene DTD unter *http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd* in einem Editor, und lassen Sie sich den Bereich `img` anzeigen.

Listing 3.1 Der Auszug für `img` aus der XHTML-DTD

```
<!--================== Images
============================================-->
<!--
   To avoid accessibility problems for people who aren't
   able to see the image, you should provide a text
   description using the alt and longdesc attributes.
   In addition, avoid the use of server-side image maps.
   Note that in this DTD there is no name attribute. That
```

```
   is only available in the transitional and frameset DTD.
-->

<!ELEMENT img EMPTY>
<!ATTLIST img
  %attrs;
  src          %URI;          #REQUIRED
  alt          %Text;         #REQUIRED
  longdesc     %URI;          #IMPLIED
  height       %Length;       #IMPLIED
  width        %Length;       #IMPLIED
  usemap       %URI;          #IMPLIED
  ismap        (ismap)        #IMPLIED
  >
```

Mit dem `img`-Element werden Grafiken in Webseiten eingebunden. Innerhalb der DTD ist exakt festgelegt, wie dieses Element arbeitet und welche Attribute ihm zugewiesen werden dürfen. Im aktuellen `img`-Fall steht hinter dem Elementnamen der Wert `EMPTY`. Das bedeutet, dass es sich bei `img` um ein leeres Element handelt, das demzufolge folgendermaßen aussieht:

```
<img />
```

Das ist aber längst noch nicht alles, was sich der DTD entlocken lässt. Im Bereich `ATTLIST` finden Sie die innerhalb von `img` möglichen Attribute. Angezeigt werden diese in der ersten Spalte. In der zweiten Spalte steht der Typ, der den Attributen zugewiesen werden kann. Denn jedes Attribut besitzt einen ganz bestimmten Datentyp. So bedeutet z.B. die folgende Anweisung, dass dem Attribut `src` ein URI zugewiesen werden muss.

```
src          %URI;
```

Und dann gibt es noch eine dritte Spalte. In der wird angegeben, ob ein Attribut innerhalb von `img` verwendet werden muss oder ob es angegeben werden kann. Wobei die mit `REQUIRED` gekennzeichneten Attribute Pflicht sind. `IMPLIED` hingegen bedeutet, dass das betreffende Attribut zwar angegeben werden kann, aber nicht angegeben werden muss.

Steht innerhalb eines XHTML-Dokuments diese Anweisung:

Listing 3.2 So sieht ein Verweis auf eine DTD aus.

```
<!DOCTYPE html
     PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

dann sollte der Browser die vorliegende Webseite daraufhin überprüfen, ob diese den in der DTD angegebenen Regeln folgt. Ist das nicht der Fall, sollte ein Verarbeitungsfehler ausgegeben werden.

Die Syntax und Semantik von Dokumenttypdefinitionen sind Bestandteil der XML-Spezifikation. Gerade dieser Aspekt führt immer wieder zu Kritik, da die DTD-Syntax selber kein XML ist.

Die klassischen Dokumenttypdefinitionen sind mittlerweile übrigens nicht mehr die einzige Möglichkeit, das Schema für ein Datenmodell zu beschreiben. Denn es gibt auch entsprechende Varianten in einer XML-Syntax. Die bekannteste ist hierbei sicherlich XML Schema. Mehr zu XML Schema dann im nächsten Kapitel.

Ähnlich wie die für XHTML gezeigte Dokumenttypdefinitionen können Sie auch für Ihre eigenen XML-Dokumente Dokumenttypdefinitionen anlegen. In dieser definieren Sie, welche Elemente und Attribute enthalten sein und welchen Datentyp Attributwerte haben müssen. Damit aber noch nicht genug: Über eine Dokumenttypdefinition können Sie z.B. auch festlegen, welche physischen Einheiten[1] ein Dokument besitzen soll.

3.2 Die Dokumenttypdeklaration

Im Lauf dieses Buches und im Umgang mit XML werden Sie immer wieder auf zwei ähnlich klingende Begriffe stoßen. Dennoch handelt es sich bei der Dokumenttypdeklaration um etwas völlig anderes als bei der Dokumenttypdefinition (DTD: Document Type Definition). Die Dokumenttypdeklaration dient dazu, den Bezug zu einer DTD herzustellen. Eine DTD beschreibt hingegen die Regeln für erlaubte Elemente, Attribute usw., die auf eine Gruppe von XML-Dokumenten angewandt werden können. Mehr zu diesem Thema erfahren Sie in Kapitel 3.

Um den Bezug zwischen einer Dokumenttypdeklaration und einer DTD herzustellen, gibt es zwei Möglichkeiten. Sie können die DTD entweder innerhalb der Dokumenttypdeklaration definieren, oder Sie referenzieren eine externe Datei, in der die DTD enthalten ist.

Die Deklaration muss hinter der XML-Deklaration, aber noch vor dem ersten Element des Dokuments eingefügt werden.

Listing 3.3 Ein Dokument mit integrierter DTD

```
<?xml version="1.0"?>
<!DOCTYPE welt
[
<!ELEMENT welt (gruss)>
<!ELEMENT gruss (#PCDATA)>
]>
<welt>
   <gruss>Hallo, Welt!</gruss>
</welt>
```

Hinter `DOCTYPE` folgt durch ein Leerzeichen getrennt der Name der Dokumentinstanz. Dieser Name muss mit dem Namen des innerhalb der XML-Datei verwendeten Wurzelelements übereinstimmen. Im aktuellen Beispiel ist der Name der Dokumentinstanz also `welt`, und der Name des Wurzelelements lautet ebenfalls `welt`.

Die eigentliche Dokumenttypdefinition steht innerhalb einer öffnenden und schließenden eckigen Klammer. Zur genauen Syntax dieser Dokumenttypdefinition dann später mehr. Bei dem gezeigten Beispiel handelt es sich um eine interne Dokumenttypdefinition. Es bietet sich immer dann an, eine Dokumenttypdefinition zu verwenden, wenn es sich um eine

[1] Ein XML-Dokument kann physisch aus mehreren Einheiten bestehen, die an unterschiedlichen Stellen gespeichert sind und erst bei Bedarf zusammengeführt werden.

kleine DTD mit nur wenigen Elementen handelt. Anderenfalls wird die XML-Datei schnell unübersichtlich.

3.2.1 Externe DTDs verwenden

Gerade für Dokumente mit vielen Elementen und Regeln bietet sich die Verwendung einer Dokumenttypdeklaration mit einer externen DTD an. Somit erhalten Sie eine exakte Trennung zwischen den Informationen der XML-Datei und deren Beschreibung mittels DTD. Zu Demonstrationszwecken verwenden wir für die Darstellung der Dokumenttypdeklaration mit einer externen DTD das gleiche Dokument wie im vorherigen Beispiel. Obwohl die Dokumente prinzipiell identisch sind, sehen sie unterschiedlich aus. Zunächst der Inhalt der ausgelagerten DTD.

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

Beachten Sie, dass es verschiedene Varianten für Dokumenttypdeklarationen mit einer externen DTD gibt. Das folgende Beispiel zeigt die Dokumenttypdeklaration, die sich auf die zuvor gezeigte externe DTD bezieht.

Listing 3.4 Hier wurde auf eine externe DTD verwiesen.

```
<?xml version="1.0"?>
<!DOCTYPE hallo SYSTEM "hallo.dtd">
<hallo>
   Hallo, Welt
</hallo>
```

Die Dokumenttypdeklaration wird über `<!DOCTYPE` eingeleitet. Hieran schließt sich, durch ein Leerzeichen getrennt, der Name der Dokumentklasse an. Dieser Name muss mit dem Wurzelelement der XML-Datei übereinstimmen. Es folgt eines der beiden Schlüsselwörter `SYSTEM` oder `PUBLIC`. Welche dieser beiden Varianten Sie einsetzen, hängt davon ab, ob Sie den Speicherort der DTD explizit kennen und angeben können.

`SYSTEM` wird immer dann verwendet, wenn der Speicherort der DTD bekannt ist. Hierbei spielt es keine Rolle, ob sich die DTD auf Ihrem oder einem anderen Rechner befindet. Einen solchen Fall zeigte das vorherige Beispiel. Hier liegt die DTD im gleichen Verzeichnis wie das XML-Dokument.

Ein weiteres Beispiel für die korrekte Verwendung des `SYSTEM`-Schlüsselworts zeigt aber auch die nachstehende Syntax.

```
<!DOCTYPE hallo SYSTEM "http://www.hanser.de/xml/hallo.dtd">
```

Hier befindet sich die DTD *hallo.dtd* auf einem anderen Rechner. Für die Verwendung einer solchen Syntax gelten die gleichen Regeln wie beim Setzen von Verweisen in HTML.

Das Schlüsselwort `PUBLIC` wird immer dann eingesetzt, wenn der Speicherort der DTD nicht explizit, sondern durch einen öffentlichen Bezeichner angegeben werden soll. Wenn Sie diese Variante wählen, folgen hinter dem Schlüsselwort `PUBLIC` mehrere Angaben. Die folgende Syntax vermittelt einen ersten Eindruck dieser Notationsvariante.

```
<!DOCTYPE EMail PUBLIC "-//hanser//DTD hallo 2.0//DE"
 "http://www.hanser.de/xml/hallo.dtd">
```

Drei Anweisungen folgen dem PUBLIC-Schlüsselwort. Zunächst muss der Hersteller der DTD genannt werden. In unserem Beispiel ist dies `hanser`. Bei dieser Anweisung kann es sich also beispielsweise um ein Unternehmen, eine öffentliche Einrichtung oder eine einzelne Person handeln. Im nächsten Schritt muss ein Name für die DTD vergeben werden. Dieser schließt sich hinter dem Schlüsselwort DTD an. Im gezeigten Beispiel wurde `hallo` verwendet. Die Versionsnummer, hier `2.0`, schließt diesen Schritt ab. Die letzte Pflichtangabe befasst sich mit der innerhalb der DTD verwendeten Sprache. Es wird angegeben, in welcher Sprache die verwendeten Attribute, Tags usw. notiert wurden. In unserem Beispiel wurde die DTD in deutscher Sprache verfasst. Sollte deren Inhalt z.B. Englisch sein, müssten Sie hier EN angeben. Fakultativ ist die letzte Angabe. Hier können Sie den vollständigen URI der DTD notieren.

Dank solcher öffentlicher DTDs kennt der XML-Parser die Regeln der DTD und muss auf diese nicht permanent zugreifen. Darüber hinaus können öffentliche DTDs für beliebig viele Dokumentinstanzen verwendet werden. Der Vorteil dieser DTD-Behandlung besteht darin, dass die XML-Dokumente auch dann angesehen werden können, wenn keine Online-Verbindung besteht. Ein klassisches Beispiel für den Einsatz einer öffentlichen DTD zeigt die folgende Syntax.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

Es handelt sich hierbei um die DTD von XHTML 1.0. Trifft ein WWW-Browser auf diese Anweisung, sollte er eigentlich die Regeln der XHTML-DTD auslesen und mit dem vorliegenden Dokument vergleichen. Stimmt das Dokument nicht mit den Regeln überein, sollte entweder eine Fehlermeldung erscheinen, oder die fehlerhaften Angaben sollten zumindest nicht angezeigt werden.

Auf den folgenden Seiten lernen Sie den Umgang mit Element- und Attributtypdeklarationen kennen. Um das anschaulicher zu machen, wird folgende XML-Datei zugrunde gelegt:

Listing 3.5 Das ist das Ausgangsdokument für die folgenden Beispiele.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE bibliothek SYSTEM "bibliothek.dtd">
<bibliothek>
   <buch>
      <autor>Tony Parsons</autor>
      <titel>Als wir unsterblich waren</titel>
      <inhalt>
         <details>
            <stichwort>Punkrock</stichwort>
            <adresse>Oxford Street, London</adresse>
            <kontakt>
               <telefon>0043343434</telefon>
               <email>tony@parsons.co.uk</email>
            </kontakt>
            <grafik/>
         </details>
      </inhalt>
      <erschienen>10.10.2005</erschienen>
      <probe>Siehe www.kamphausen.de</probe>
   </buch>
```

```
   <buch>
      <autor>John King</autor>
      <titel> Der letzte Kick</titel>
      <inhalt>
         <details>
            <stichwort>England</stichwort>
            <adresse>Maystreet, Dublin</adresse>
            <kontakt>
               <email>news@fo2ml.co.uk</email>
            </kontakt>
            <grafik/>
         </details>
      </inhalt>
      <erschienen>4.11.1999</erschienen>
      <probe>Weitere Informationen: www.goldmann.de</probe>
   </buch>
</bibliothek>
```

3.2.2 Bedingte Abschnitte

Mittels zweier spezieller Schlüsselwörter können Sie angeben, ob der Bereich zwischen den eckigen Klammern bei der Weiterverarbeitung des Dokuments berücksichtigt werden soll. Durch diesen Mechanismus haben Sie die Möglichkeit, bestimmte Abschnitt einer Dokumenttypdefinition wahlweise ein- oder auszuschalten. Auch das lässt sich wieder am besten anhand eines Beispiels zeigen.

Listing 3.6 So sehen bedingte Abschnitte aus.

```
<![INCLUDE[
   <!ELEMENT buecher (titel, autor)>
]]>

<![IGNORE[
   <!ELEMENT buecher (titel, autor, cover)>
]]>
```

In dieser Form wird beim Verarbeiten dieser Dokumenttypdefinition der durch `IGNORE` gekennzeichnete Abschnitt so behandelt, als ob er überhaupt nicht existieren würde. Der `INCLUDE`-Abschnitt wiederum wird als ganz normaler Abschnitt angesehen. Soll dann später nur der untere Abschnitt verwendet werden, lässt sich das ganz einfach anpassen. Dafür tauschen Sie einfach `INCLUDE` und `IGNORE` aus.

Listing 3.7 So leicht ist das mit einem Wechsel.

```
<![IGNORE[
   <!ELEMENT buecher (titel, autor)>
]]>

<![INCLUDE[
   <!ELEMENT buecher (titel, autor, cover)>
]]>
```

Wurden in der Dokumenttypdefinition im Vorfeld Parameterentitäten für die Schlüsselwörter `IGNORE` und/oder `INCLUDE` definiert, können Sie bedingte Abschnitt auch über die Parameterentitäten aktivieren oder deaktivieren. Dazu müssen Sie lediglich eine entsprechende Entitätenreferenz im bedingten Abschnitt verwenden.

Listing 3.8 Hier wurde eine Entitätenreferenz angegeben.

```
<!ENTITY % freiwillig "INCLUDE">
<![%freiwillig;[
   <!ELEMENT buecher #PCDATA>
>>]
```

In diesem Beispiel wird der Prozessor erst die Referenz auf die Parameterentität auflösen. Wenn dann im XML-Dokument der Wert der Parameterentität innerhalb der internen Teilmenge der Dokumenttypdefinition überschrieben wird, kann der bedingte Abschnitt bei der Verarbeitung ausgeblendet werden.

```
<!DOCTYPE bibliothek SYSTEM "bedingte_abschnitte.dtd"
[[ENTITY % freiwillig "IGNORE">]>
```

Der bedingte Abschnitt ist jetzt mittels `IGNORE` ausgeblendet.

3.3 Elemente beschreiben: Elementtypdeklarationen

Wie Sie Dokumenttypdefinitionen anlegen und mit XML-Dokumenten verknüpfen bzw. die Dokumenttypdefinitionen innerhalb der XML-Datei notieren, haben Sie auf den vorherigen Seiten gesehen. In diesem Abschnitt geht es um die Syntax der Elementtypdeklarationen.

Die allgemeine Syntax sieht folgendermaßen aus:

```
<!ELEMENT Name Inhaltsmodell>
```

Der Elementname muss mit einem Buchstaben oder einem Unterstrich beginnen. Ein Doppelpunkt wäre zwar erlaubt, sollte aber vermieden werden, da diese Syntax für die Trennung von Namensraumpräfixen vom lokalen Namen verwendet wird. Achten Sie außerdem darauf, dass die Namen case-sensitiv sind, also zwischen Groß- und Kleinschreibung unterschieden wird.

In der Praxis sollten Sie in jedem Fall beschreibende Elementnamen verwenden. Dadurch nutzen Sie einen der großen Vorteile von XML voll aus: Die Daten beschreiben sich größtenteils selbst.

Achten Sie zusätzlich darauf, Elementnamen innerhalb einer DTD nicht mehrfach zu verwenden. Denn kommen in einem Dokument zwei gleichnamige Elementtypdeklarationen vor, ignoriert der XML-Prozessor die zweite Deklaration.

3.3.1 Ein Beispiel für eine DTD

Wie sich Dokumenttypdefinitionen anlegen lassen, lässt sich am besten anhand eines Beispiels zeigen. Gegenstand der folgenden Beispielsyntax ist eine kleine Bibliothek. Diese Bibliothek enthält mehrere Bücher. Zu jedem Buch wird der Autor angegeben, der Titel notiert, eine Inhaltsangabe beschreibt das Buch, und es wird auch noch der Veröffentlichungstermin notiert.

Die DTD zu dem eingangs gezeigten XML-Beispiel könnte folgendermaßen aussehen:

Listing 3.9 Das ist doch schon mal ein Anfang.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!ELEMENT bibliothek (buch+)>
<!ELEMENT buch (autor, titel, inhalt, erschienen, probe)>
<!ELEMENT autor (#PCDATA)>
<!ELEMENT titel (#PCDATA)>
<!ELEMENT inhalt (details+)>
<!ELEMENT details (stichwort, adresse, kontakt?, grafik?)>
<!ELEMENT stichwort (#PCDATA)>
<!ELEMENT adresse (#PCDATA)>
<!ELEMENT kontakt (telefon | email* | (telefon, email*))>
<!ELEMENT telefon (#PCDATA)>
<!ELEMENT email (#PCDATA)>
<!ELEMENT grafik EMPTY>
<!ELEMENT erschienen (#PCDATA)>
<!ELEMENT probe ANY>
```

Perfekt ist die DTD natürlich noch nicht, sie wird aber Stück für Stück ausgebaut und perfektioniert.[2]

3.3.2 Elemente, die weitere Elemente enthalten

Das Element `bibliothek` darf als Inhalt ausschließlich Kindelemente vom Typ `buch` enthalten. Nun wäre eine Bibliothek nicht wirklich eine Bibliothek, wenn sie lediglich ein Buch aufweisen würde. Demzufolge wird die Syntax

```
<!ELEMENT bibliothek (buch+)>
```

verwendet. Das Pluszeichen hinter dem Elementnamen signalisiert, dass die Bibliothek mehr als ein Buch enthalten kann. Dieses Pluszeichen wird als sogenannte Kardinalität verwendet. Es gibt also an, wie oft ein Element an einer bestimmten Stelle vorkommen kann, darf oder muss. Gibt es keinen Operator wird, muss das `buch`-Element genau einmal vorkommen, was für eine Bibliothek die denkbar schlechteste Lösung wäre.

Das Element `buch` wiederum setzt sich aus mehreren Kindelementen zusammen. Welche das sind, steht innerhalb der Klammern.

```
<!ELEMENT buch (autor, titel, inhalt, erschienen, probe)>
```

Die betreffenden Elemente müssen im XML-Dokument in der angegebenen Reihenfolge – als Sequenz – vorkommen. Wenn sie in einer falschen Reihenfolge stehen, ist das Dokument nicht gültig.

Im aktuellen Beispiel wird für jedes Kindelement von `buch` eine entsprechende Elementtypdeklaration in der DTD definiert.

3.3.3 Elemente mit Zeichendaten

Innerhalb der Beispiel-DTD sind vergleichsweise viele Elemente mit dem Datentyp `#PCDATA` ausgestattet. Bei `#PCDATA` handelt es sich um eine Abkürzung für *parsed charac-*

[2] Korrekt ist die DTD allerdings bereits jetzt. Um das zu überprüfen, kann man das Dokument validieren lassen. Dieses Validieren empfiehlt sich ohnehin immer, wenn man DTDs erstellt.

ter data, was reinen Text ohne Markup beschreibt. Trifft ein Parser auf dieses Schlüsselwort, weiß er, dass vorhandene Entitätenreferenzen vor der Weiterverarbeitung aufgelöst werden müssen.

Die Verwendung von `#PCDATA` macht allerdings auch eine der Schwächen klassischer DTDs deutlich: Denn es handelt sich hierbei um den einzigen Datentyp, der für das Element `inhalt` angegeben werden kann. Es besteht also nicht die Möglichkeit, dass Sie beispielsweise angeben, bei dem Elementinhalt handelt es sich um ein Element, in dem ausschließlich Zahlen enthalten sind. Für solche detaillierten Beschreibungen muss auf XML Schema zurückgegriffen werden. Mehr zu diesem Thema dann im nächsten Kapitel.

`#PCDATA` wird benutzt, wenn im Element Fließtext gespeichert werden soll. Die Textlänge spielt dabei keine Rolle.

Durch die genannten Punkte wird deutlich, dass beispielsweise das `autor`-Element selbst keine weiteren Kindelemente umschließt, da es bekanntermaßen ausschließlich aus Zeichendaten besteht.

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

Innerhalb von `#PCDATA` dürfen die folgenden Zeichen nicht verwendet werden:

- `>`
- `<`
- `&`

Diese müssen, wenn Sie sie einsetzen wollen, durch ihre entsprechenden Entitätenreferenzen ersetzt werden.

- Aus `<` wird `<`
- Aus `>` wird `>`
- Aus `&` wird `&`

3.3.4 Containerelemente verwenden

Im Unterschied zu `inhalt` besitzt das Element `bibliothek` mehrere Kindelemente, die teilweise selbst Kindelemente enthalten. Wie Elemente gekennzeichnet werden, die weitere Elemente enthalten, wurde bereits gezeigt.

```
<!ELEMENT bibliothek (buch+)>
```

Dazu wird einfach ein Pluszeichen angehängt. Denn bekanntermaßen sollte eine Bibliothek mindestens ein Buch als Bestand haben. Das Element `buch` besteht wiederum aus mehreren Kindelementen, die jeweils unterschiedlichen Typs sein können. Diese Elemente sind innerhalb der Klammer aufgeführt und jeweils durch ein Komma getrennt.

```
<!ELEMENT buch (autor, titel, inhalt, erschienen, probe)>
```

Zunächst einmal tauchen hier die folgenden Elemente auf, die in jeden Fall vorhanden sein müssen:

- autor
- titel
- inhalt
- erschienen
- probe

Es handelt sich hierbei jeweils um einfache Elemente, die ausschließlich Zeichendaten enthalten dürfen. Interessant ist das Element kontakt, das sich von den anderen Elementen unterscheidet. Denn bei diesem Element handelt es sich um einen Container zusätzlicher Elemente. Allerdings soll dieses Element optional sein. Sind also Kontaktinformationen für einen Autor nicht vorhanden, darf das Element fehlen. Um das zu erreichen, wird dem Element der ?-Operator angehängt. Dieses Element darf somit höchstens ein Mal vorkommen, in einer Instanz des Dokumenttyps kann es aber auch weggelassen werden.

Und der Vollständigkeit halber hier noch einmal der Inhalt des kontakt-Elements:

```
<!ELEMENT kontakt (telefon | email* | (telefon, email*))>
```

3.3.5 Leere Elemente

In der aktuellen DTD wird ein optionales grafik-Element definiert. Dabei handelt es sich um ein leeres Element. Es enthält weder Zeichendaten noch andere Elemente. Verwendet werden leere Elemente hauptsächlich, um mittels Attributen Verweise auf Dateien (Grafiken, Videos etc.) einzubinden.

In der Beispiel-DTD soll ein Element grafik definiert werden, über das Bilder der jeweiligen Autoren eingeblendet werden können.

```
<!ELEMENT grafik EMPTY>
```

Leere Elemente können auf zwei unterschiedliche Arten geschrieben werden, die allerdings das gleiche Ergebnis liefern. Also entweder verwenden Sie diese

```
<grafik/>
```

oder diese

```
<grafik></grafik>
```

Schreibweise. Es wurde bereits darauf hingewiesen, dass leere Elemente sehr oft für das Einbinden von Bildern verwendet werden. So auch in diesem Fall. Die benötigten Informationen für die Integration eines Fotos liegen in Form von Attributen vor. Das wichtigste ist dabei sicherlich src, über das die Quelle des Bildes angegeben wird.

```
<img src="autor.jpg"/>
```

3.3.6 Inhaltsalternativen angeben

Interessant ist im aktuellen DTD-Beispiel auch die Frage, welche Kontaktinformationen für einen Autor hinterlegt werden können. Schwierig ist das vor dem Hintergrund, dass die

jeweiligen Autoren sicherlich ganz unterschiedliche Vorlieben haben. Hier eine kleine Auswahl der Möglichkeiten:

- Per E-Mail
- Per Telefon
- Mehrere E-Mail-Adressen
- Per Telefon und per E-Mail

Diese Liste ließe sich nun z.B. noch um Angaben zu Faxnummer u.Ä. erweitern. Das Prinzip ist aber klar: Es steht im Vorfeld nicht fest, welche Kontaktmöglichkeiten angegeben werden. Dennoch müssen die unterschiedlichen Varianten innerhalb der DTD berücksichtigt werden. So etwas lässt sich innerhalb von DTDs folgendermaßen realisieren:

```
<!ELEMENT element (variante1 | variante2 | variante3)>
```

Innerhalb der Klammern werden die entsprechenden Alternativen aufgelistet. Als Operator wird der senkrechte Strich verwendet. Dadurch wird nur eine der aufgeführten Kontaktvarianten berücksichtigt.

Zusätzlich können Sie festlegen, dass an einer bestimmten Stelle beliebig viele Elemente stehen können. Verwendet wird dafür der `*`.

```
<!ELEMENT element (variante1 | variante2* | variante3)>
```

In diesem Beispiel könnten bei `variante2` also beliebig viele Elemente notiert werden.

Umgemünzt auf die Beispiel-DTD sieht das folgendermaßen aus:

```
<!ELEMENT kontakt (telefon | email* | (telefon, email*))>
```

Was bedeutet das im Einzelnen?

Insgesamt gibt es drei verschiedene Alternativen für die Kontaktdaten.

- Es wird nur eine Telefonnummer angegeben.
- Es werden lediglich E-Mail-Adressen angegeben. Durch die Sternsyntax spielt es keine Rolle, wie viele E-Mail-Adressen notiert werden.
- Durch `(telefon, email*)` wird bestimmt, dass eine Telefonnummer und eine beliebige Anzahl an E-Mail-Adressen angegeben werden können. Dabei muss man immer zuerst die Telefonnummer angeben.

Welche Operatoren zur Verfügung stehen, fasst Tabelle 3.1 zusammen:

Tabelle 3.1: Mögliche Operatoren

Operator	Beschreibung
?	Das vorherige Element bzw. die vorherige Elementgruppe kann einmal vorkommen, darf aber auch fehlen.
*	Das vorherige Element bzw. die vorherige Elementgruppe kann beliebig oft vorkommen oder fehlen.
+	Das vorherige Element bzw. die vorherige Elementgruppe muss mindestens einmal vorkommen, kann aber auch mehrfach vorkommen.

Operator	Beschreibung
|	Das ist das Trennzeichen zwischen den sich gegenseitig ausschließenden Alternativen.
()	Hierüber lassen sich Elementgruppen bilden.
,	Dieser Operator dient als Trennzeichen innerhalb einer Sequenz von Elementen.

3.3.7 Elemente mit beliebigem Inhalt

Wenn Sie noch einmal einen Blick in die Beispiel-DTD werfen, wird Ihnen dort das Element `probe` auffallen.

```
<!ELEMENT probe ANY>
```

Dieses Element ist für eine Leseprobe des entsprechenden Buchs des Autors gedacht. Auf welche Art und Weise aber könnte eine solche Leseprobe existieren? Lassen Sie Ihrer Fantasie freien Lauf, und überlegen Sie, von welchem Inhalt das Element `probe` sein könnte.

- Reine Textdaten
- Leere Elemente
- Überhaupt kein Inhalt
- Zusätzlich deklarierte Kindelemente
- Mischung aus Textdaten und Kindelementen

Die Möglichkeiten sind vielfältig. Um dieser Vielfalt gerecht zu werden, verwendet man das Inhaltsmodell `ANY`.

Interessant ist `ANY` vor allem hinsichtlich der Wiederverwendung von DTD-Teilen in anderen Dokumenten. Schließlich erspart man sich dadurch eine nachträgliche Änderung. Denn wo es keine Beschränkungen des Inhalts gibt, kann natürlich auch beliebiger Inhalt eingesetzt werden, ohne dass es zu Problemen kommt.

Nützlich ist der Einsatz von `ANY` aber auch während der Entwicklungsphase einer DTD. Das gilt vor allem dann, wenn noch unklar ist, wie bestimmte Bereiche im Endeffekt aussehen sollen. Denen weist man dann einfach `ANY` zu. Auf diese Weise lässt sich die DTD trotzdem validieren. Sobald eine endgültige Entscheidung getroffen wurde, ändert man `ANY` entsprechend um.

3.3.8 Elemente mit gemischtem Inhalt

Bei dem sogenannten Mixed Content lassen sich Textdaten und Elemente mischen. Auch hierzu wieder ein Beispiel:

```
<!ELEMENT probe (#PCDATA | verweise | infos)*>
```

Hier sorgt der Operator `*`, der hinter der schließenden Klammer steht, dafür, dass das `probe`-Element aus beliebig vielen Zeichendaten und den angegebenen Elementen bestehen kann. Ein entsprechend gültiges Element könnte folgendermaßen aussehen:

Listing 3.10 Das ist ein gültiges Element.

```
<probe>Weiterführendes
   <infos>Ausführliche Informationen zum
      Buch finden Sie unter:
   <verweis>http://www.hanser.de/</verweis></infos>
</probe>
```

So verlockend diese Mixed-Content-Variante aber auf den ersten Blick auch sein mag, in der Praxis wird üblicherweise versucht, ihren Einsatz zu vermeiden. Zwei Punkte sprechen gegen gemischten Inhalt.

- Die Häufigkeit kann nicht wie üblich über die Operatoren `+`, `?` und `*` festgelegt werden.
- Die Reihenfolge der innerhalb des gemischten Inhalts vorhandenen Kindelemente lässt sich nicht definieren.

Interessant ist Mixed Content aber immer, wenn man solche Datenbestände, die bislang nicht im XML-Format vorliegen, nach und nach in XML konvertieren möchte.

3.3.9 Das Inhaltsmodell und die Reihenfolge

Auf den vorherigen Seiten haben Sie gesehen, welche verschiedenen Inhaltsmodelle für Elementinhalte verfügbar sind. Hier noch einmal die entsprechenden Varianten zusammengefasst:

Tabelle 3.2: Die verschiedenen Inhaltsmodelle

Inhaltsmodell	Beschreibung
`ANY`	Das Element kann beliebigen Inhalt besitzen, bei dem es sich allerdings um wohlgeformtes XML handeln muss.
`EMPTY`	Das Element besitzt keinen Inhalt, es kann allerdings Attribute haben.
`#PCDATA`	Das Element setzt sich aus Zeichendaten zusammen.
Gemischter Inhalt	Der Inhalt des Elements kann aus Zeichendaten und aus Unterelementen bestehen.
Elementinhalt	Das Element enthält ausschließlich Unterelemente.

In welcher Reihenfolge die einzelnen Elemente innerhalb der DTD stehen, spielt normalerweise keine Rolle. Ausnahme davon bilden Fälle, in denen Elemente mehrmals deklariert werden. Aber auch wenn Parameterentitäten verwendet werden, muss man die Deklarationen, auf die man Bezug nehmen will, vorher deklarieren.

Eine „beliebte“ Fehlerquelle ist es übrigens, dass Elemente, die als Unterelemente eines anderen Elements erscheinen sollen, nicht deklariert werden. Hierzu ein kleines Beispiel:

Listing 3.11 Hier hat sich ein Fehler eingeschlichen.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!ELEMENT bibliothek (buch+)>
<!ELEMENT buch (autor, titel, inhalt, erschienen, probe)>
<!ELEMENT autor (#PCDATA)>
<!ELEMENT titel (#PCDATA)>
<!ELEMENT inhalt (details+)>
<!ELEMENT details (stichwort, adresse, kontakt?, grafik?)>
<!ELEMENT stichwort (#PCDATA)>
<!ELEMENT adresse (#PCDATA)>
<!ELEMENT kontakt (telefon | email* | (telefon, email*))>
<!ELEMENT telefon (#PCDATA)>
<!ELEMENT grafik EMPTY>
<!ELEMENT erschienen (#PCDATA)>
<!ELEMENT probe ANY>
```

In diesem Beispiel wurde beim `kontakt`-Element das Element `email` verwendet.

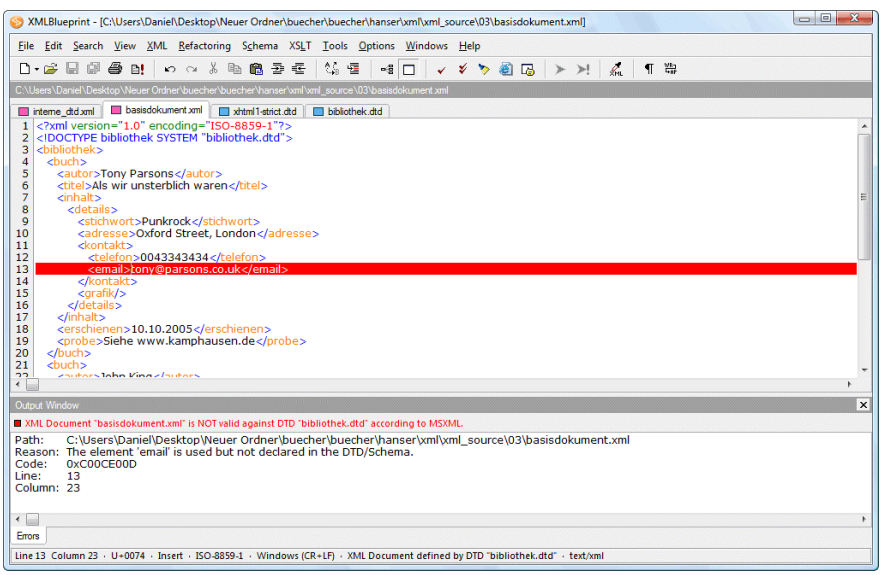

Abbildung 3.1 Hier stimmt etwas nicht.

Allerdings wurde dieses Element nicht deklariert. Korrekterweise müsste die DTD um den folgenden Eintrag erweitert werden:

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

Gute XML-Editoren haben für solche Zwecke einen entsprechenden DTD-Validator mit an Bord, der solche Fehler erkennt. Aber Achtung: Längst nicht alle Validatoren erkennen diese speziellen Fehler!

3.3.10 Kommentare erhöhen die Übersichtlichkeit

Wie in ganz normalen XML- oder HTML-Elementen sind auch innerhalb von DTDs Kommentare erlaubt. Solche Kommentare werden durch `<!--` eingeleitet und durch die Zeichenfolge `-->` abgeschlossen. Zwischen diesen Zeichendarf beliebiger Inhalt stehen,

der sich auch über mehrere Zeilen erstrecken kann. Lediglich die Zeichenfolge -- darf nicht innerhalb des Kommentars stehen.

Listing 3.12 Hier wurde immerhin ein Kommentar eingefügt.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!ELEMENT bibliothek (buch+)>
<!ELEMENT buch (autor, titel, inhalt, erschienen, probe)>
<!ELEMENT autor (#PCDATA)>
<!ELEMENT titel (#PCDATA)>
<!ELEMENT inhalt (details+)>
<!ELEMENT details (stichwort, adresse, kontakt?, grafik?)>
<!ELEMENT stichwort (#PCDATA)>
<!ELEMENT adresse (#PCDATA)>
<!ELEMENT kontakt (telefon | email* | (telefon, email*))>
<!ELEMENT telefon (#PCDATA)>
<!ELEMENT email (#PCDATA)>
<!ELEMENT grafik EMPTY>
<!ELEMENT erschienen (#PCDATA)>

<!--Im Element probe können Leseproben des Buchs gespeichert werden. Da-
bei spielt es keine Rolle, ob die als PDF, TXT oder sonst wie vorliegen--
>

<!ELEMENT probe ANY>
```

Wie viele Kommentare Sie innerhalb einer DTD verwenden, ist zunächst einmal Geschmacksache. Allerdings sollten Sie die DTD nicht mit unnötigen Kommentaren überfrachten. So würde es im gezeigten Beispiel keinen Sinn machen, die einzelnen Elemente immer folgendermaßen zu kommentieren:

```
<!ELEMENT titel (#PCDATA)>
<!--Das ist der Buchtitel -->
```

Normalerweise ergibt sich der Aspekt, dass das `titel`-Element den Buchtitel aufnimmt, von alleine. Wer gute und logische Elementnamen verwendet, kann auf eine umfangreiche Kommentierung der DTD verzichten. Pflicht sollten Kommentare aber überall dort sein, wo Unklarheiten auftreten könnten.

3.4 Attribute beschreiben: Attributlistendeklarationen

Neben der Elementtyp- gibt es auch die Attributlistendeklaration. Denn genauso wie Sie alle Elemente angeben können, die in einem gültigen Dokument vorkommen dürfen, müssen Sie auch alle Attribute definieren, die das Element selbst mitbringt. Beachten Sie, dass Attribute nicht einzeln, sondern innerhalb von sogenannten Attributliste stehen, die einem bestimmten Element zugeordnet werden.

Zunächst ein Blick auf die allgemeine Syntax einer Attributlistendeklaration:

Listing 3.13 So sieht die Attributlistendefinition allgemein aus.

```
<!ATTLIST Elementname
Attributname Attributtyp Vorgabewert
Attributname Attributtyp Vorgabewert

...
>
```

Wie eine solche Definition in der Praxis aussieht, wird anhand der schon bekannten Syntax gezeigt. Dieses Mal wird das `buch`-Element um zwei Attribute erweitert.

Listing 3.14 Zwei neue Attribute kommen hinzu.

```
<buch sachgebiet="roman" fach="5">
     <autor>Nick Hornby</autor>
     <isbn>2342223434</isbn>
     <titel>About a Boy</titel>
</buch>
```

Die beiden Attribute `sachgebiet` und `roman` müssen sich nun natürlich auch noch in der Dokumenttypdeklaration wiederfinden.

Listing 3.15 Auch die DTD muss angepasst werden.

```
<!DOCTYPE bibliothek [
   <!ELEMENT bibliothek (buch+)>
   <!ELEMENT buch (autor, isbn, titel, bestellen)>
   <!ELEMENT autor (#PCDATA)>
   <!ATTLIST buch sachgebiet
   fach
]>
```

Für Attributnamen sind die gleichen Regeln zu beachten, die auch für Elementnamen gelten. Zudem bleibt es Ihnen überlassen, alle Attribute eines Elements in einer Attributliste zu deklarieren oder mehrere Teillisten für dasselbe Element zu verwenden.

3.4.1 Attributtypen und Vorgaberegelungen

Im Unterschied zu Elementen, bei denen bekanntermaßen außer Inhaltsmodellen nur noch nicht weiter typisierte Zeichendaten enthalten sein können, kann man bei Werten, die Attributen zugewiesen werden sollen, exaktere Angaben machen. Zudem lässt sich bei Attributen eine Vorgabe für den Fall definieren, dass innerhalb der Dokumentinstanz ein Wert für ein bestimmtes Attribut fehlt.

Prinzipiell wird zwischen zehn verschiedenen Typen gewählt. Zunächst einmal wäre da ein Typ ohne Struktur.

```
CDATA
```

Dann gibt es drei Listentypen.

- `ENTITIES`
- `IDREFS`
- `NMTOKENS`

Und zu guter Letzt gibt es noch die folgenden atomaren Token-Typen:

- Aufzählung
- `ENTITY`
- `ID`
- `IDREF`

- NMTOKEN
- Notation

In Tabelle 3.3 wird beschrieben, was die einzelnen Typen bewirken:

Tabelle 3.3: Die verfügbaren Attributtypen für DTDs

Attributtyp	Beschreibung
Aufzählung	Die Token-Werte werden in einer Liste notiert, die von Klammern umschlossen ist. Von den angegebenen Werten kann und muss jeweils einer verwendet werden.
CDATA	Es handelt sich um einfache Zeichendaten, in denen kein Markup enthalten ist. Ausnahme bilden Entitätenreferenzen. Die sind erlaubt.
ENTITY	Der Name einer innerhalb der DTD deklarierten nicht geparsten Entität.
ENTITIES	Dabei handelt es sich um eine Liste von Entitäten, die jeweils durch Leerzeichen getrennt notiert werden.
ID	Ein eindeutiger XML-Name, der für die Identifizierung eines Elements verwendet werden kann. Der Name entspricht einem Schlüsselwert innerhalb eines Datensatzes.
IDREF	Das ist der Verweis auf den ID-Identifier. Dabei muss der Wert von IDREF mit dem ID-Wert eines anderen Elements innerhalb des Dokuments übereinstimmen.
IDREFS	Hierbei handelt es sich um eine Liste von ID-Identifizierern, die jeweils durch ein Leerzeichen getrennt werden.
NMTOKEN	Das ist ein Namenssymbol aus beliebigen Zeichen, die innerhalb von XML-Namen erlaubt sind. (Leerzeichen sind nicht möglich.)
NMTOKENS	Dabei handelt es sich um eine Liste von Namens-Tokens, die jeweils durch ein Leerzeichen getrennt notiert werden.
NOTATION	Ein Verweis auf eine Notation, also beispielsweise auf eine Grafikdatei.

3.4.1.1 Vorgabetypen

Bei der Vergabe von Attributwerten lassen sich verschiedene Qualifizierungen vornehmen. Anhand der Qualifizierung werden Verwendungsmöglichkeiten und die Bedingungen des Attributs angegeben. Zunächst eine kurze Zusammenfassung der möglichen Werte. Im Anschluss folgen einige Beispiele.

Tabelle 3.4: Die möglichen Vorgabetypen

Vorgabedeklaration	Beschreibung
Attributwert	Es wird ein Standardwert angegeben. Dieser wird benutzt, wenn vom Benutzer keine anderen Angaben gemacht werden. Die dabei angegebene Zeichenkette muss in Anführungszeichen stehen.

Vorgabedeklaration	Beschreibung
`#FIXED Wert`	Bestimmt. dass in jedem Fall der angegebene Wert verwendet wird.
`#IMPLIED`	Es ist dem Benutzer freigestellt, ob er einen Attributwert verwendet.
`#REQIURED`	Wenn das entsprechende Element vorhanden ist, muss das Attribut in jedem Fall verwendet werden.

Attribute, die in jedem Fall vorkommen müssen, werden mit `#REQUIRED` gekennzeichnet. Ein typisches Beispiel für ein notwendiges Attribut kennen Sie vielleicht aus HTML. Dort muss das `img`-Element das `src`-Attribut besitzen. Genau das kann auch in XML bestimmt werden. Das folgende Beispiel ist korrekt.

Listing 3.16 Eine korrekte Syntax

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE buch [
   <!ELEMENT buch (#PCDATA)>
   <!ATTLIST buch
      ausgabe (tb | gb) #REQUIRED
   >
]>
<buch ausgabe="tb">
   American Psycho
</buch>
```

Hier wurde festgelegt, dass das Attribut `ausgabe` in jedem Fall angegeben werden muss. Und tatsächlich wurde dieses Attribut dem `buch`-Element zugewiesen. Falsch ist hingegen die folgende Syntax:

Listing 3.17 Hier stimmt was nicht.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE buch [
   <!ELEMENT buch (#PCDATA)>
   <!ATTLIST buch
      ausgabe (tb | gb) #REQUIRED
   >
]>
<buch>
   American Psycho
</buch>
```

Denn auch in diesem Beispiel wurde `ausgabe` mit `#REQUIRED` gekennzeichnet, allerdings wurde das Attribut bei `buch` nicht eingesetzt.

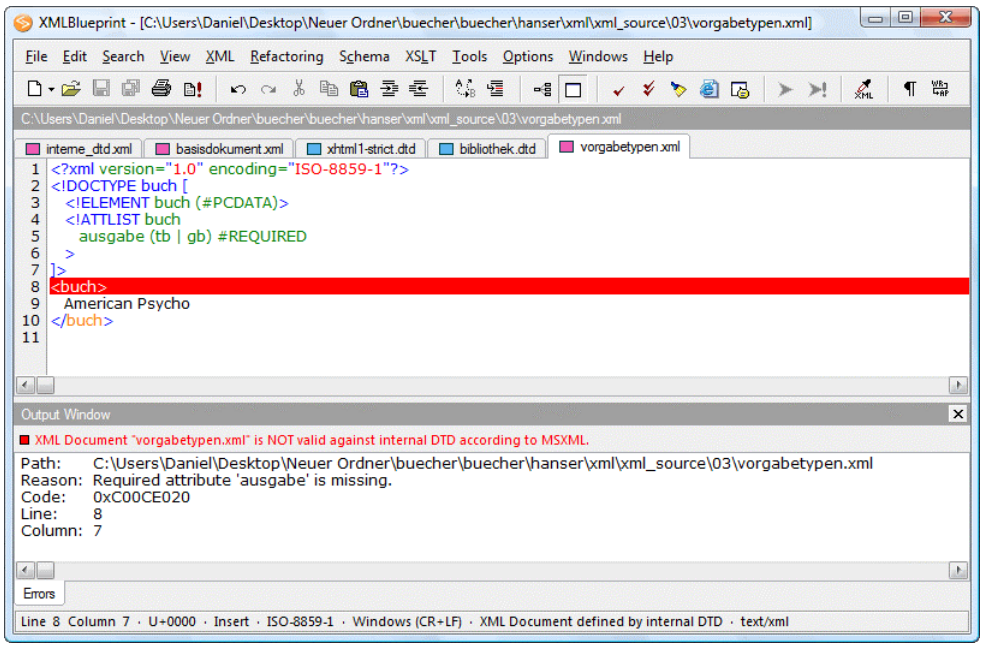

Abbildung 3.2 Der Validator meldet einen Fehler.

Folgerichtig in diesem Fall dann auch die Validator-Ausgabe.

```
Required attribute 'ausgabe' is missing.
```

Durch `#IMPLIED` werden Attribute gekennzeichnet, die zwar angegeben werden können, aber nicht müssen. So ist zum Beispiel die folgende Syntax korrekt:

Listing 3.18 `ausgabe` kann angegeben werden.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE buch [
   <!ELEMENT buch (#PCDATA)>
   <!ATTLIST buch
      ausgabe (tb | gb) #IMPLIED
   >
]>
<buch ausgabe="tb">
     American Psycho
</buch>
```

Das Attribut `ausgabe` wurde mit `#IMPLIED` ausgezeichnet und dann auch tatsächlich verwendet. Korrekt ist allerdings auch die folgende Syntax:

Listing 3.19 Hier wurde es weggelassen.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE buch [
   <!ELEMENT buch (#PCDATA)>
   <!ATTLIST buch
      ausgabe (tb | gb) #IMPLIED
   >
]>
<buch>
```

```
    American Psycho
</buch>
```

Da `ausgabe` mit `#IMPLIED` markiert wurde, steht es Ihnen frei, es im Dokument zu verwenden. Im aktuellen Beispiel wurde es nicht eingesetzt, das Dokument ist trotzdem valide.

Mittels `#FIXED` wird ein Attributwert definiert, der unveränderbar ist. Die entsprechende Software, die das Dokument und die DTD verarbeitet, sollte die Attribute und ihre Werte von sich aus einsetzen. Das funktioniert erfahrungsgemäß allerdings nicht immer. Auch hierzu wieder ein Beispiel:

Listing 3.20 Das sollte eigentlich funktionieren.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE link [
   <!ELEMENT link EMPTY>
   <!ATTLIST link
      href CDATA #REQUIRED
      rel NMTOKEN #FIXED "stylesheet"
      type CDATA #FIXED "text/css"
   >
]>
<link href="styles.css"/>
```

Es handelt sich hierbei um eine normale Syntax, mit der eine CSS-Datei mittels `<link>` eingebunden werden kann. Trifft entsprechende Software auf diesen Code, wird bzw. sollte sie daraus Folgendes machen:

```
<link href="stylest.css" rel="stylesheet" type="text/css"  />
```

In modernen Browsern funktioniert die Ersetzung.

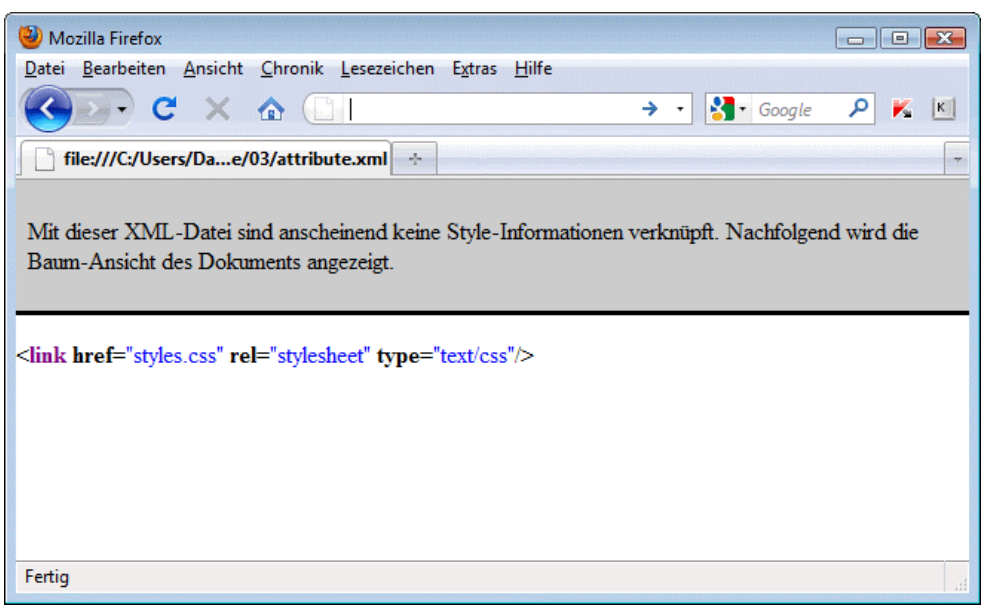

Abbildung 3.3 Der Validator meldet einen Fehler.

Die letzte Variante, auf die hier genauer eingegangen wird, ist der Einsatz von Standardwerten. Bei dem Standardwert handelt es sich um die Qualifizierung, und er wird – ähnlich wie bei `#FIXED` – standardmäßig eingesetzt. Im Gegensatz zu `#FIXED` können allerdings gemäß dem jeweiligen Datentyp des Attributs auch andere Werte eingesetzt werden.

Listing 3.21 Das ist eine Variante.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE buch [
   <!ELEMENT buch (#PCDATA)>
   <!ATTLIST buch
      ausgabe (tb | gb) "gb"
   >
]>
<buch ausgabe="gb">
    About a boy
</buch>
```

Genau das Gleiche macht auch die folgende Syntax:

Listing 3.22 Das funktioniert ebenfalls.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE person [
   <!ELEMENT person (#PCDATA)>
   <!ATTLIST person
      status (single | verheiratet) "verheiratet"
   >
]>
<buch>
    About a boy
</buch>
```

Das Ergebnis beider Varianten sähe folgendermaßen aus:

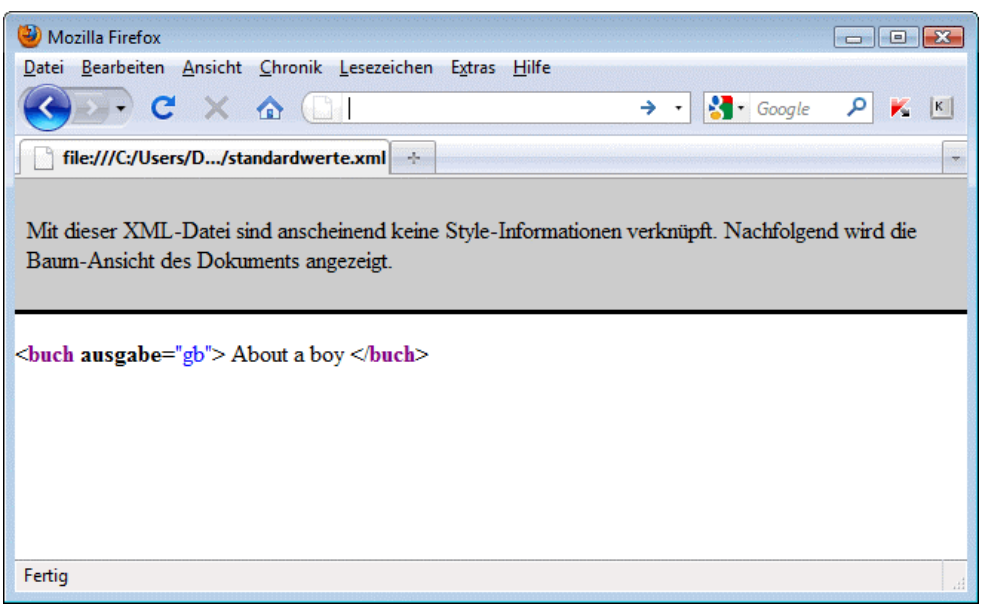

Abbildung 3.4 Das Attribut wurde eingefügt.

3.5 Auf andere Elemente verweisen

Es besteht die Möglichkeit, innerhalb von Dokumenten interne Verweise zu erstellen. Verwendet wird dafür das `IDREF`-Attribut. Damit ein solcher Verweis gesetzt werden kann, müssen Attribute vom Typ `ID` vorhanden sein. Die allgemeine Syntax für einen solchen Verweis sieht folgendermaßen aus:

```
<!ATTLIST Elementname Attributname IDREF [#REQUIRED|#IMPLIED|...]>
```

Durch diese Syntax kann die DTD überprüfen, ob das Attribut optional ist oder angegeben werden muss.

Auch dieser Aspekt lässt sich wieder am besten anhand eines Beispiels zeigen. Stellen Sie sich vor, es soll ein Dokument definiert werden, in dem verschiedene Kurse stehen.

- *XML für Einsteiger*
- *XML für Fortgeschrittene*

Nun ist es aber so, dass man an dem Kurs *XML für Fortgeschrittene* nur teilnehmen kann, wenn man *XML für Einsteiger* bereits absolviert hat. Genau so etwas ist mittels `IDREF` möglich.

Listing 3.23 Hier werden die Voraussetzungen definiert.

```
<!ATLIST kurse
   id ID #REQUIRED
   voraussetzung IDREF #IMPLIED

   ...
>
```

Durch diese Deklaration wird die Verknüpfung mit einem anderen Kurs möglich, sie ist allerdings keine Pflicht. Das Dokument selbst könnte dann folgendermaßen aussehen:

Listing 3.24 Das ist das Dokument bzw. ein Ausschnitt davon.

```
<kurse>
   <kurs id="einsteiger">

      ...

   </kurs>
   <kurs id="fortgeschrittene" voraussetzung="einsteiger">

      ...

   </kurs>
</kurse>
```

3.6 Entitäten – Kürzel verwenden

Auf den folgenden Seiten geht es um die sogenannten Entitäten. Dabei handelt es sich eigentlich um nichts anderes als definierte Kürzel. Sie kennen solche Entitäten möglicherweise von Ihrem Textverarbeitungsprogramm her. Dort laufen Entitäten z.B. unter dem

Namen *AutoText* oder *Schnellbausteine*. Informationen zum Einsatz von Entitäten haben Sie bereits im 2. Kapitel erhalten. Dort ging es allerdings ganz allgemein um XML-Entitäten. Auf den folgenden Seiten stehen die sogenannten Parameterentitäten im Vordergrund. Durch diese ist es möglich, innerhalb einer DTD Verweise auf Teile einer internen oder externen DTD zu setzen.

3.6.1 Interne Entitäten

Sehr oft werden Ihnen die internen Entitäten begegnen. Meistens nimmt man diese dazu, um Kürzel für längere Zeichenketten zu definieren und sich auf diese Weise Tipparbeit zu ersparen. Angenommen, Sie müssen innerhalb Ihres XML-Dokuments mehrmals

```
Hanser Verlag
```

schreiben. Genau für solche Fälle sind Entitäten da. Denn dank einer solchen Entität können Sie eine Abkürzung definieren. Im Fall von `Hanser Verlag` könnte das beispielsweise folgendermaßen aussehen:

Listing 3.25 Eine Abkürzung wurde definiert.

```
<?xml version="1.0" ?>
<!DOCTYPE verlag [
<!ELEMENT verlag (#PCDATA)>
<!ENTITY hv "Hanser Verlag">
]>
<verlag>
   &hv;
</verlag>
```

Wird dieses XML-Dokument nun im Browser aufgerufen, ergibt sich folgendes Bild:

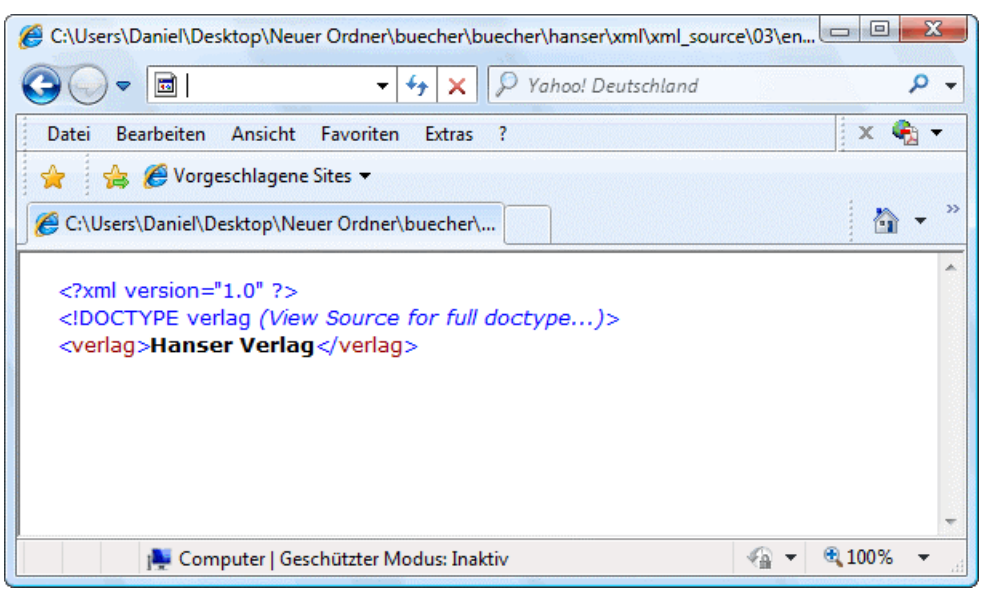

Abbildung 3.5 Die Entität wurde entsprechend ersetzt.

Der Browser wandelt die Entität in die tatsächliche Zeichenkette um. Damit das funktioniert, muss die natürlich entsprechend angegeben werden. Die allgemeine Syntax für Entitäten innerhalb von DTDs sieht folgendermaßen aus:

```
<!ENTITY Kürzel "Der lange Text">
```

Für die Ersetzung im Dokument selbst muss ebenfalls gesorgt werden. Damit der Parser erkennt, dass es sich um eine Entität handelt, wird Entitäten ein `&`-Zeichen vorangestellt, und es wird mit einem Semikolon beendet.

```
&Kürzel;
```

Es sind übrigens auch verschachtelte Entitäten möglich. Man kann also auch Folgendes einsetzen:

Listing 3.26 Eine typische Entitätendefinition

```
<?xml version="1.0" ?>
<!DOCTYPE verlag [
<!ELEMENT verlag (#PCDATA)>
<!ENTITY hv "Hanser Verlag">
<!ENTITY claim "Bücher &amp; Mehr">
<!ENTITY ausgabe "&hv; bietet &claim;">
]>
<verlag>
   &ausgabe;
</verlag>
```

Beachten Sie, dass in dieser Syntax bereits eine eingebaute XML-Entität integriert wurde. Diese wird verwendet, um bei `Bücher & Mehr` das `&`-Zeichen entsprechend auszuzeichnen. Auch hier wieder das Ergebnis im Browser:

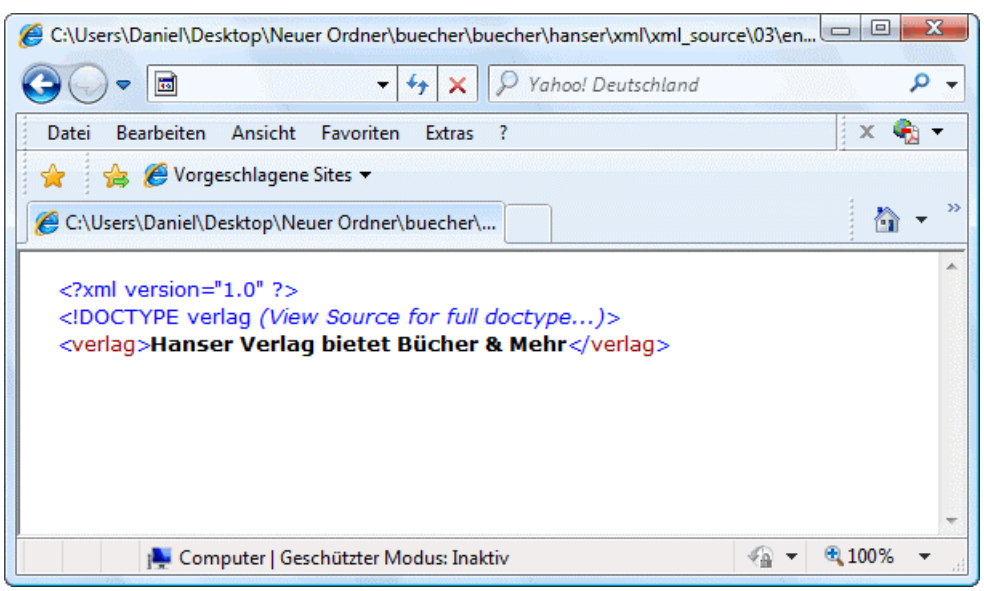

Abbildung 3.6 Die Entität wurde korrekt umgewandelt.

Der Browser erkennt ebenfalls, dass es sich um Entitäten handelt, und wandelt sie korrekt um.

3.6.2 Externe Entitäten

Der Einsatz interner Entitäten ist eigentlich nur ratsam, wenn die Ersetzungstexte vergleichsweise kurz sind. Im vorherigen Beispiel wurde gezeigt, dass sich Firmennamen u.Ä. durchaus mittels interner Entitäten ersetzen lassen. Sobald die Texte umfangreicher werden, sollte man sie in externe Dateien auslagern. Das hält die DTD übersichtlich. Innerhalb der DTD wird dann nämlich lediglich ein Verweis auf die externe Entität definiert. Die allgemeine Syntax sieht folgendermaßen aus:

```
<!ENTITY name SYSTEM uri>
```

In diesem Fall wird ein URI für den Bezug auf die externe Entität verwendet. Sollte es sich um einen Verweis auf eine öffentliche Ressource handeln, wird zusätzlich ein sogenannter *Formal Public Identifier* (FPI) verwendet.

```
<!ENTITY name PUBLIC fpi  uri>
```

Einen solchen FPI kennen Sie möglicherweise von XHTML. Dort sieht der FPI folgendermaßen aus:

```
-//W3C//DTD XHTML 1.0 Transitional//EN
```

Auch die Definition externer Entitäten lässt sich am besten anhand eines Beispiels zeigen. Zunächst die Syntax, über die auf die externe Entität verwiesen wird.

```
<!ENTITY xmlbuch SYSTEM "beschreibung.xml">
```

Die externe Entität liegt in der Datei *beschreibung.xml*, die sich im gleichen Verzeichnis wie die aktuelle XML-Datei befindet. Bei der externen Datei ist darauf zu achten, dass die eigentliche XML-Datei auch noch nach dem Einfügen der Referenz wohlgeformt ist. Die DTD muss daher entsprechend angepasst werden.

Angenommen, es existiert ein Element `buch`, in das der Inhalt eines externen Dokuments eingefügt werden soll, das innerhalb eines `buchtext`-Elements ausführliche Angaben über ein Buch enthält. Die dahingehend angepasste DTD sieht folgendermaßen aus:

```
<!ELEMENT buch (buchtext?)>
<ELEMENT buchtext (#PCDATA)>
```

Insgesamt könnte das XML-Dokument folgendermaßen aussehen:

Listing 3.27 Das ist die XML-Datei.

```
<?xml version="1.0" ?>
<!DOCTYPE bibliothek [
<!ELEMENT bibliothek (buch)>
<!ENTITY xmlbuch SYSTEM "beschreibung.xml">
<!ELEMENT buch (buchtext?)>
<!ELEMENT buchtext (#PCDATA)>
]>
<bibliothek>
   <buch>
      <buchtext>
```

```
            &xmlbuch;
        </buchtext>
    </buch>
</bibliothek>
```

Der Inhalt der *beschreibung.xml* stellt sich so dar:

Listing 3.28 Das ist die Beschreibung für das XML-Buch.

```
Moderne Webanwendungen sind ohne XML nicht mehr denkbar. Ob Portale, Con-
tent-Management-Systeme oder die Anbindung von Webapplikationen an Daten-
banken - an XML führt im Web kein Weg vorbei. Was aber verbirgt sich hin-
ter XML, und wie kann der Webentwickler die verschiedenen XML-basierten
Sprachen effizient einsetzen? Hier setzt dieses Buch an.
```

Und auch wieder das Ergebnis im Browser:

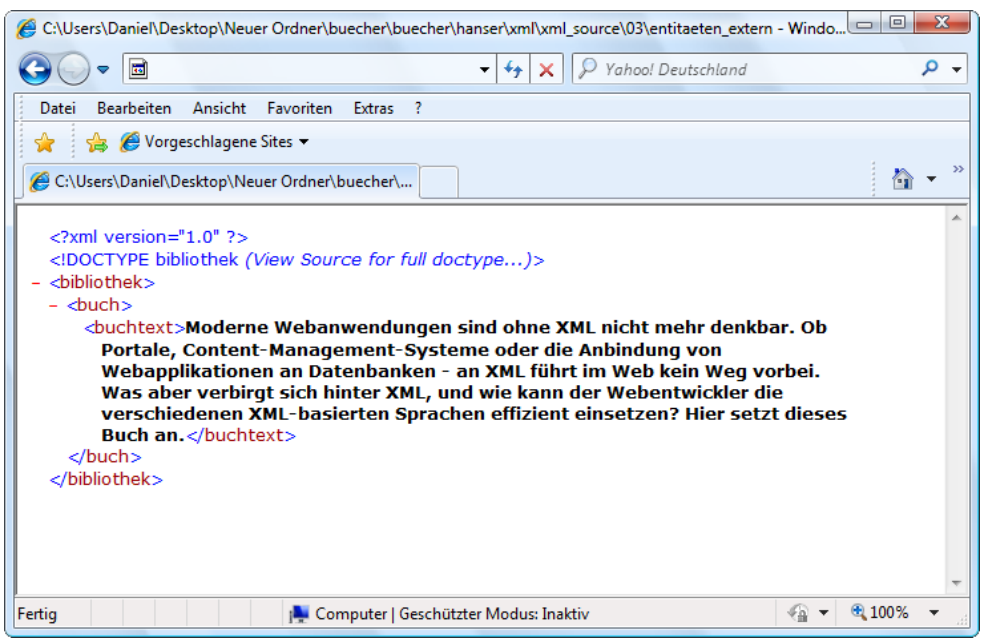

Abbildung 3.7 Der externe Inhalt wurde eingebunden.

Die Ersetzung klappt also. Nun wurde im vorherigen Beispiel lediglich auf eine externe Ressource zurückgegriffen. Wenn Sie sich vorstellen, dass auf diese Weise eine Bibliothek nachgebildet werden sollte, wird schnell klar, dass diese Ein-Datei-Lösung nicht ideal ist. XML bietet aber die Möglichkeit, mehrere Dateien zu verwenden. So könnte man z.B. für jedes Buch eine eigene XML-Datei anlegen.

Listing 3.29 Jedes Buch bekommt seine eigene XML-Datei.

```
<!ENTITY xmlbuch SYSTEM "xmlbuch.xml">
<!ENTITY javabuch SYSTEM "javabuch.xml">
<!ENTITY cssbuch SYSTEM "cssbuch.xml">
```

Die dazu passende Dokumentinstanz sähe folgendermaßen aus:

Listing 3.30 Und das könnte die Dokumentinstanz sein.

```
<bibliothek>
   &xmlbuch;
   &javabuch;
   &cssbuch;
</bibliothek>
```

3.6.3 Notationen und ungeparste Entitäten

Entwickelt wurde das XML-Format für die Zusammenarbeit mit Textdateien. Ebenso kann man in XML-Dokumente aber auch andere Formate einbinden. Denken Sie z.B. an das Einbinden von Grafiken. In diesem Zusammenhang fällt zwangsläufig der Begriff Notationen. Dabei handelt es sich um Hinweise für die Interpretation von externen Daten, die XML-Parser nicht direkt verarbeiten können.

Die allgemeine Syntax für eine solche Notation sieht folgendermaßen aus:

```
<!NOTATION name SYSTEM uri
```

Auch hier gilt wieder: Wenn sich der Verweis auf eine öffentliche Ressource bezieht, muss ein entsprechender Formal Public Identifier (FIP) angegeben werden.

```
<!NOTATION name PUBLIC fpi  uri
```

Trifft der Parser auf eine solche Syntax, erkennt er, dass es sich bei dem Format um etwas anderes als um XML handelt. Der den Elementen zugewiesene Name kann später innerhalb von Attributlisten oder in Entitäten verwendet werden.

Stellt sich nun noch die Frage, wie Entitäten mit Verweisen auf externe Datenformate aussehen, die nicht XML-konform sind. Solche Entitäten, die vom Prozessor nicht geparst werden sollen, werden ähnlich wie allgemeine externe Entitäten deklariert. Einziger Unterschied ist das Schlüsselwort `NDATA`, durch das ein zuvor deklarierter Notationstyp benannt wird.

Am besten lässt sich die Notations-„Problematik“ anhand eines Beispiels demonstrieren. Angenommen, es soll in einem XML-Dokument ein `grafik`-Element aufgenommen werden, über das ein Bild ausgegeben wird. Der entsprechende DTD-Eintrag könnte wie folgt aussehen:

Listing 3.31 Das ist ein Auszug der DTD.

```
<!NOTATION jpeg  SYSTEM "image/jpeg">
<!ENTITY buch SYSTEM "xml.jpg" NDATA jpeg>
<!ELEMENT grafik EMPTY>
<!ATTLIST grafik pfad ENTITY #IMPLIED>
```

Im Dokument selbst könne Folgendes stehen:

```
<grafik pfad="buch"/>
```

Durch diese Syntax wird ein Verweis auf die ungeparste Entität gesetzt. Verwendet wird dafür der Wert des `pfad`-Attributs.

3.6.4 Parameterentitäten einsetzen

Auf den folgenden Seiten geht es um die sogenannten Parameterentitäten. Diese werden – und hier unterscheiden sie sich zu den bislang vorgestellten allgemeinen Entitäten – ausschließlich innerhalb von DTDs verwendet. In Dokumentinstanzen selbst spielen sie keine Rolle. Dabei können Parameterentitäten sowohl innerhalb von internen als auch externen Teilmengen verwendet werden.

3.6.4.1 Interne Parameterentitäten

Innerhalb von internen DTD verwendet man Parameterentitäten, um mehrmals auftretende Elementgruppen oder Attributlisten nur einmal definieren zu müssen. Zum einen hilft das dabei, die Tipparbeit zu ersparen. Ebenso kann dadurch aber auch das Dokumentmodell besser gepflegt werden. Denn schließlich müssen die Änderungen ausschließlich an der Stelle vorgenommen werden, an denen die Parameterentität deklariert wurde.

Eingeleitet wird die interne Parameterentität durch das Schlüsselwort `ENTITY`.

```
<!ENTITY ...>
```

Daran schließt sich das Prozentzeichen an. Erst durch dieses Zeichen können Parameterentitäten von allgemeinen Entitäten unterschieden werden.

```
<!ENTITY % … "'>
```

Im Anschluss wird der Name der Entität angegeben, unter dem sie später referenziert bzw. aufgerufen werden kann.

```
<!ENTITY % autor …>
```

Daran schließt sich in einfache oder doppelte Anführungszeichen gesetzt der DTD-Block an, der beim Aufruf der Entität eingesetzt werden soll. Beachten Sie, dass dieser DTD-Block in sich nicht unbedingt valides oder wohlgeformtes XML sein bzw. eine eigene DTD bilden muss. Erst wenn der DTD-Block an der referenzierten Stelle verwendet wird, muss er zusammen mit dem Dokument valide und wohlgeformt sein und eine schlüssige Gesamt-DTD bilden.

Auch hierzu wieder ein Beispiel:

```
<!ENTITY % koch 'autor CDATA "Michael Mayer"
                          email  CDATA "kontakt@hanser.de"'>
```

Diese Entität definiert einen Block, in dem entsprechende Informationen über einen Autor enthalten sind. Auf diese Entität kann an verschiedenen Stellen innerhalb der DTD folgendermaßen zugegriffen werden:

```
<!ATTLIST buch %koch;>
```

Der Aufruf setzt sich aus dem bekannten Prozentzeichen, dem Namen der Entität und einem Semikolon zusammen. Zwischen diesen drei Elementen darf kein Leerzeichen stehen.

Nun soll die Verwendung einer Parameterentität noch einmal in Aktion gezeigt werden. Wieder bezogen auf eine entsprechende Autoren-DTD könnte sich folgendes Bild ergeben:

Listing 3.32 So sieht die Autoren-DTD aus.

```
<!DOCTYPE bibliothek [
     <!ENTITY % mayer 'autor CDATA "Michael Mayer"
                          email  CDATA  "dunst@hanser.de"'>
     <!ELEMENT buch (kapitel)*>
     <!ATTLIST buch %dunst;>
     <!ELEMENT kapitel (ueberschrift, text)>
     <!ATTLIST kapitel %mayer;>
     <!ELEMENT ueberschrift (#PCDATA)>
     <!ELEMENT text (#PCDATA)>
]>
```

Oft werden Parameterentitäten auch im Zusammenhang mit mehrfach benötigten Attributdefinitionen verwendet. Das spart Tipparbeit, hält die DTD übersichtlich und macht sie pflegeleichter.

Listing 3.33 Das ist pflegeleichter Code.

```
<!ENTITY % id "id ID #REQIRED">
   <!ATTLIST buch
      %id;>
      ...
   >

   <!ATTLIST autor
      %id;>
      ...
   >

   <!ATTLIST verlag
      %id;>
      ...
   >
```

3.6.4.2 Externe Parameterentitäten

Externe Parameterentitäten erfüllen einen ähnlichen Zweck wie ihre internen Verwandten. Durch sie können vordefinierte DTD-Fragmente an einer oder mehreren Stellen in die DTD eingebunden werden. Im Gegensatz zu internen Parameterentitäten befinden sich die externen allerdings nicht innerhalb der DTD, die die Entität aufruft, sondern in speziellen Dateien.

Oftmals ist es nämlich tatsächlich sinnvoll, DTDs in mehrere kleinere Teile zu zerlegen und diese bei Bedarf miteinander zu kombinieren. Die DTD-Blöcke, die in externen Dateien gespeichert wurden, können von allen DTDs, die Zugriff auf diese Dateien haben, eingebunden werden.

Diese Form der DTD-Gestaltung hat den Vorteil, dass die DTD modularisiert wird und sich die einzelnen DTD-Teile sehr schnell ändern lassen. Bei externen Parameterentitäten wird zwischen zwei Varianten unterschieden:

- Direkt lokalisierbare externe Parameterentitäten, die mit einem URI zu finden sind. (`SYSTEM`)
- Externe Parameterentitäten, die systemunabhängig auffindbar sind. (`PUBLIC`)

Allerdings spielen die systemunabhängig auffindbaren Parameterentitäten in der Praxis kaum eine Rolle. Die allgemeine Syntax für eine externe Parameterentität, die über einen URI angesprochen wird, sieht folgendermaßen aus:

Listing 3.34 Das ist die allgemeine Syntax.

```
<!ENTITY % name SYSTEM "definition.dtd">
<!-- Entität-Aufruf -->
%name;
```

Anders als bei internen Parameterentitäten wird bei dieser Variante kein DTD-Block genannt, der verwendet werden soll. Vielmehr wird hier das Schlüsselwort `SYSTEM` eingesetzt, an das sich in Anführungszeichen gesetzt der URI der entsprechenden Datei anschließt.

Der Vollständigkeit halber auch noch die `PUBLIC`-Variante:

```
<!ENTITY % name PUBLIC fip uri>
```

3.7 DTD-Tipps für die Praxis

Sie haben auf den vorherigen Seiten eine ganze Menge rund um das Thema DTD erfahren. Mit diesem Wissen können Sie nun Ihre eigenen DTDs erstellen. Wie aber geht man dabei eigentlich vor? Verwendet man am besten Attribute oder doch lieber Elemente? Und ist es überhaupt sinnvoll, für jedes XML-Dokument eine DTD anzulegen? Diesen Fragen widmen sich die folgenden Abschnitte.

3.7.1 Elemente oder Attribute

Am wichtigsten ist zweifellos die Frage, auf welche Art die Daten wiedergegeben werden sollen. Sie werden nämlich immer vor der Entscheidung stehen, ob Sie Attribute oder Elemente einsetzen sollen. In diesem Kapitel wurden zu diesem Aspekt bereits einige allgemeine Hinweise gegeben. Als Faustregel lässt sich aber festhalten: Alle lesbaren Informationen sollten in Elemente eingefügt werden. Für alles andere wird üblicherweise auf Attribute zurückgegriffen.

3.7.2 Parameterentitäten

Sie haben bereits die Möglichkeiten kennengelernt, wie sich alternative Inhaltsmodelle definieren und im Dokument aktivieren und deaktivieren lassen. Ebenso können Parameterentitäten aber auch dazu genutzt werden, häufig benötigte Attributlisten oder Erweiterungen der DTD aufzunehmen. Da innerhalb von Entitäten ebenfalls Entitätenreferenzen enthalten sein können, lassen sich ineinander verschachtelte Module erstellen. Ein Beispiel zeigt, wie das funktioniert.

Listing 3.35 So lassen sich Parameterentitäten zusammenfassen.

```
<!ENTITY % buch
  " id ID #IMPLIED
   ausgabe CDATA #IMPLIED
   sprache CDATA #IMPLIED
  "
>

...

<!ATTLIST inhalt %buch;
   autor CDATA #REQUIRED
   stichwort CDATA #IMPLIED>
...

<!ATTLIST lager %buch;
   status (ja|nein|bald) #IMPLIED>
...
```

In diesem Beispiel wurden die beiden globalen Attribute `ausgabe` und `sprache` in dem Parameterentität `buch` zusammengefasst. Auf diese Weise können die beiden Attribute in verschiedenen Modulen verwendet werden, die dann außer den Attributdefinitionen innerhalb der Parameterentität zusätzliche Attributdefinitionen enthalten.

3.7.3 Mögliche Gründe für eine DTD

Natürlich stellt sich die Frage, wann denn eigentlich DTDs verwendet werden sollten. (Im nächsten Abschnitt finden Sie übrigens Hinweise dazu, wann Sie besser auf eine DTD verzichten sollten.)

Wer regelmäßig große XML-Dokumente erstellt, die den gleichen Regeln und somit auch der gleichen DTD genügen müssen, muss eine DTD anlegen. Die Dokumente können dann gegen die DTD validiert werden.

Die Struktur eines XML-Dokuments lässt sich oft besser anhand der DTD als am eigentlichen Dokument beschreiben. Somit ist die DTD – vor allem bei umfangreichen Projekten, die dokumentiert werden müssen – ein ideales Dokumentationsmittel.

Es gibt Techniken in XML, die eine DTD zwingend voraussetzen. Dazu gehören z.B. der Einsatz von `INCLUDE` und `IGNORE`, der Einsatz von Attributen des Typs `ID` und `IDREF` sowie von Auflistungstypen.

Eigentlich zwingend nötig wird der Einsatz einer DTD dort, wo solche Daten mehrmals verwendet werden, die an einer anderen Stelle angelegt wurden. Stellen Sie sich vor, Sie bekommen eine XML-Datei per E-Mail zugeschickt, bearbeiten diese weiter und schicken sie dann wieder zurück. Wenn in diesem Fall eine DTD vorhanden ist, können Sie das Dokument zur Kontrolle gegen die DTD validieren. (Gleiches gilt dann natürlich auch für den Empfänger.)

3.7.4 Hier lohnen sich DTDs nicht

Dass DTDs sinnvoll und wichtig sind, ist – sicherlich auch nach der Lektüre dieses Kapitels – deutlich geworden. Allerdings gilt das nicht uneingeschränkt. So gibt es durchaus

Situationen, in denen man mit einer DTD deutlich über das Ziel hinausschießt. Hier also die „ultimative Liste“ dazu, wann sich der Einsatz einer DTD eigentlich nicht lohnt:

- DTDs für ein einzelnes Dokument sind meistens überflüssig. DTDs lohnen sich nur, wenn mehrere Dokumente erstellt werden, die auf dieser DTD basieren sollen.
- Wer ausschließlich XML-Dokumente mittels XSLT transformieren möchte, kann auf den Einsatz einer DTD ebenfalls verzichten. XSLT kann nämlich wohlgeformte XML-Dokumente verarbeiten, eine zusätzliche Validierung durch eine DTD ist dabei nicht nötig. Durch das Fehlen einer DTD kann man dem XML-Dokument einfach zusätzliche Elemente hinzufügen und das Ergebnis der Transformation testen, ohne dass die Validität des XML-Dokuments verloren geht.
- Wenn Sie alleine an einem XML-Dokument arbeiten, dessen Struktur noch nicht feststeht und dem Sie immer wieder neue Elemente hinzufügen, können Sie ebenfalls auf eine DTD verzichten. Zu aufwendig wären die permanenten Änderungen, als dass das noch praktikabel ist.

Sie haben gesehen, dass es nicht immer sinnvoll oder ratsam ist, eine DTD zu erstellen.

4

4 Dokumenttypdefinition reloaded: XML Schema

Im vorherigen Kapitel ging es um die Möglichkeiten, die DTDs hinsichtlich der Dokumenttypbeschreibung zu bieten haben. Um etwas ganz Ähnliches geht es auch auf den folgenden Seiten: XML Schema. Durch diesen neuen Standard sollen die Nachteile behoben werden, die DTDs haben. Um welche Nachteile es sich dabei handelt und wie Sie XML Schema einsetzen können, erfahren Sie auf den folgenden Seiten.

4.1 Die Idee hinter XML Schema

XML Schema ist eine Empfehlung des W3C zum Definieren von Strukturen für XML-Dokumente. Die offizielle Webseite zum Thema finden Sie unter *http://www.w3.org/XML/Schema*.

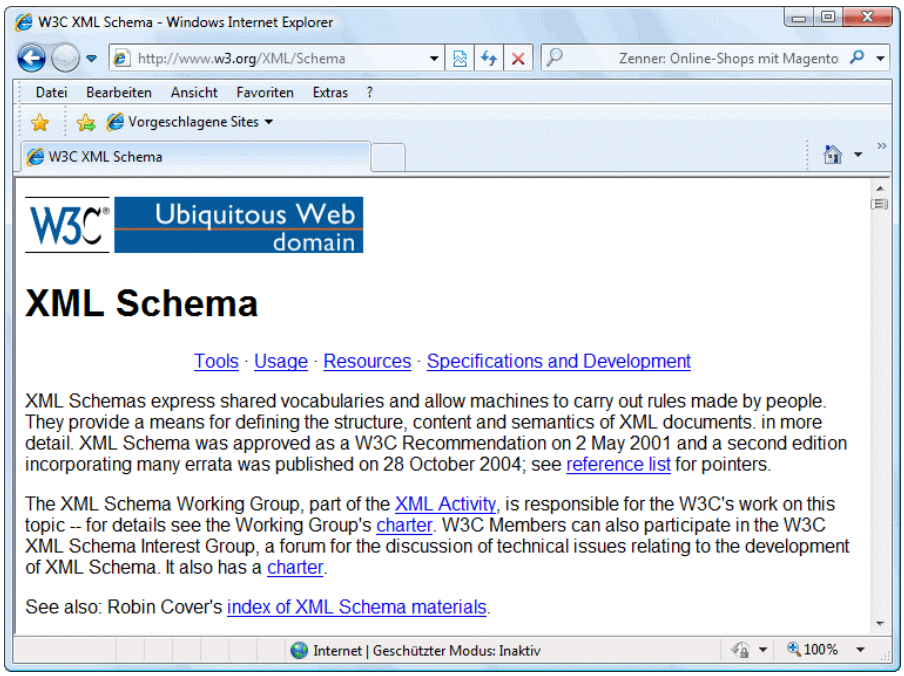

Abbildung 4.1 Hier gibt es ausführliche Informationen zu XML Schema.

Neben allerlei Tools für XML Schema gibt es dort auch alle relevanten Spezifikationen. Denn anders als viele andere Sprachen besteht XML Schema nicht nur aus einer, sondern gleich aus mehreren Spezifikationen.

- XML Schema Part 0: Primer
- XML Schema Part 1: Structures
- XML Schema Part 2: Datatypes
- XML Schema: Component Designators

Durch XML Schema wird eine Schemasprache beschrieben. Bei einer solchen Schemasprache handelt es sich um eine Sprache für die Klassifizierung von XML-Dokumenten und für die syntaktische Beschreibung ihrer Struktur und ihres Inhalts. Nun geht es in diesem Kapitel ausschließlich um XML Schema. Das bedeutet aber nicht, dass das die einzige Schema-Sprache ist. Hier einige andere Schema-Sprachen:

- SOX
- DSD
- RELAX NG
- XML-Data
- DDML
- DCD
- Schematron

- Examplotron
- Assertion Grammars
- TREX

Sie sehen, die Auswahl ist gigantisch. Bevor es mit XML Schema losgeht, noch ein kurzer Blick in die Welt einer anderen Schema-Sprache, ein kleines Beispiel zu RELAX NG. Zunächst das eigentliche XML-Dokument:

Listing 4.1 So sieht das XML-Dokument aus.

```
<addressBook>
   <card>
      <givenName>John</givenName>
      <familyName>Smith</familyName>
      <email>js@example.com</email>
   </card>
   <card>
      <name>Fred Bloggs</name>
      <email>fb@example.net</email>
   </card>
</addressBook>
```

Die DTD könnte folgendermaßen aussehen:

Listing 4.2 Das ist die DTD.

```
<!DOCTYPE addressBook [
   <!ELEMENT addressBook (card*)>
   <!ELEMENT card ((name | (givenName, familyName)), email, note?)>
   <!ELEMENT name (#PCDATA)>
   <!ELEMENT email (#PCDATA)>
   <!ELEMENT givenName (#PCDATA)>
   <!ELEMENT familyName (#PCDATA)>
   <!ELEMENT note (#PCDATA)>
]>
```

Und im direkten Vergleich noch die Syntax, die auf RELAX NG basiert.

Listing 4.3 Eine typische Syntax in RELAX NG

```
element addressBook {
  element card {
    (element name { text }
     | (element givenName { text },
        element familyName { text })),
    element email { text },
    element note { text }?
  }*
}
```

Wenn Sie diese Variante mit der im weiteren Verlauf dieses Kapitels vorgestellten Syntax von XML Schema vergleich, werden Sie feststellen, dass beide Ansätze völlig unterschiedlich sind. Das ändert allerdings nichts daran, dass sich mit beiden Varianten Dokumenttypen beschreiben lassen.

4.1.1 Die Nachteile von DTDs

XML Schema ist ein neuer Ansatz, um XML-Dokumenttypen ohne Einsatz einer DTD zu beschreiben. Dabei stellt sich zunächst die Frage, warum es eigentlich eines solch neuen Ansatzes bedarf. Denn schließlich wurde im vorherigen Kapitel gezeigt, dass sich Dokumenttypen durchaus mit DTDs definieren lassen. Es gibt also offensichtlich Defizite, die beim Einsatz von DTDs zutage treten. Denn eines darf man nicht vergessen: XML wurde zunächst für das Zusammenspiel mit Textdokumenten konzipiert. Je mehr allerdings auch andere Datenformate ins Spiel kamen und kommen, umso deutlicher werden die Schwächen von DTDs.

Die sind die größten Nachteile von DTDs:

- Die DTD-Syntax ist nicht ausdrucksstark genug. Denn zwar ermöglichen DTDs die Kontrolle darüber, welche Elemente in einem Dokument vorkommen dürfen, es sind aber keine differenzierten Einschränkungen hinsichtlich des Elementinhalts möglich.
- Es handelt sich bei ihnen selbst nicht um XML-Dokumente, wodurch sie sich nicht automatisiert mit XML-Tools verarbeiten lassen.
- Eine DTD sieht für XML-Elemente lediglich vier rudimentäre Datentypen vor, bei denen alles fast zwangsläufig darauf hinausläuft, dass ein Element Zeichenketten aufnimmt, die Daten oder Unterelemente darstellen.
- Da der Umfang der möglichen Schlüsselwörter usw. in der XML-Spezifikation 1.0 fest vorgeschrieben ist, kann die DTD nicht erweitert werden.
- Wird ein auf einer DTD basierendes XML-Dokument validiert, gibt es keine Möglichkeit, nicht deklarierte Elemente zu verwenden. Vielmehr muss man entweder auf die Validierung verzichten oder die DTD entsprechend erweitern.
- Auf DTDs kann nicht über Programmierschnittstellen wie das DOM zugegriffen werden.

DTDs haben aber nicht nur Nachteile. Denn in der Tat ist es so, dass sie unter gewissen Umständen auch Vorteile gegenüber XML Schema zu bieten haben.

- In aller Regel sind DTDs einfacher als Schemata.
- Derzeit gibt es deutlich mehr Tools für die Arbeit mit DTDs als für XML Schema. Dieser Vorteil wird sich aber sicherlich bald überholt haben, da immer mehr Softwarehäuser auf XML Schema setzen.
- XML-Dokumente lassen sich schneller über eine DTD als über ein Schema validieren. Interessant ist das vor allem im Zusammenhang mit umfangreichen Dokumenten.

Es kann also auch einiges für den Einsatz von DTDs sprechen. In aller Regel ist XML Schema aber dennoch vorzuziehen.

4.1.2 Anforderungen an XML Schema

Die Nachteile der DTDs wurden im vorherigen Abschnitt genannt. Diese Nachteile trugen dazu bei, dass man sich seitens des W3C für eine neue Variante der Modellierung von Daten-Schemas entschloss. Bereits im Jahr 1999 wurden die XML Schema Requirements verabschiedet, die heute unter *http://www.w3.org/TR/NOTE-xml-schema-req.html* zu finden sind.

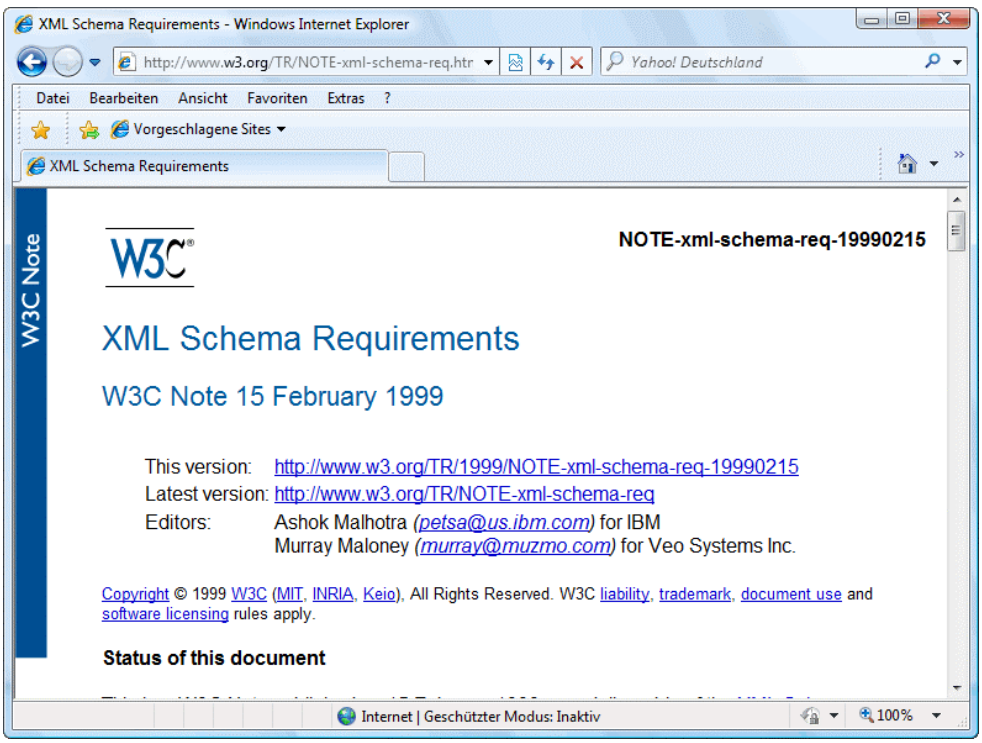

Abbildung 4.2 Hier gibt es ausführliche Informationen.

In diesem Arbeitspapier wurden die Anforderungen spezifiziert, die XML Schema erfüllen sollte. Als wichtigste Punkte galten dabei vor allem die folgenden:

- Es sollte ein Mechanismus für eingebettete Dokumentationen sein.
- Namensräume sollten berücksichtigt werden.
- Integration struktureller Schemas mit einfachen Datentypen sollte möglich sein.
- Einfache Datentypen wie `date`, `integer`, `sequence` usw. sollten unterstützt werden.

Auf der genannten Webseite können Sie noch einmal alle Voraussetzungen nachlesen, die an XML Schema gestellt wurden.

Damit Sie wissen, was Sie in diesem Kapitel erwartet, folgt ein direkter Vergleich zwischen einem XML Schema und einer klassischen DTD. Zunächst die Schema-Definition:

Listing 4.4 So sieht das Schema aus.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="html">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="head"/>
        <xsd:element name="body" type="xsd:string"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="head">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="title" type="xsd:string"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Die DTD sieht folgendermaßen aus:

Listing 4.5 Und das ist die DTD.

```
<!ELEMENT html (head, body)>
<!ELEMENT head (title)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT body (#PCDATA)>
```

Auch wenn die DTD kürzer ist, wirkt das XML Schema doch deutlich übersichtlicher. Hinzu kommt außerdem, dass man mit XML Schema viel flexibler ist. Mehr dazu dann aber im weiteren Verlauf dieses Kapitels.

4.2 Die Grundstruktur

Am besten gelingt der Einstieg in die Welt von XML Schema über ein Beispiel. Als Ausgangspunkt bzw. Basis dient das folgende XML-Dokument.

Listing 4.6 Das ist das Ausgangsdokument.

```
<?xml version="1.0" ?>
<buecher>
<kunde>
   <name>Michael Mayer</name>
   <strasse>Elbgaugasse 12</strasse>
   <plz>22524</plz>
   <ort>Hamburg</ort>
</kunde>
<bestellung>
 <buch nummer="13">
   <kategorie>Roman</kategorie>
   <titel>Die Firma</titel>
   <isbn>3453071174</isbn>
   <preis>14.90</preis>
 </buch>
 <buch nummer="16">
   <kategorie>Roman</kategorie>
   <titel>American Psycho</titel>
   <isbn>33530744474</isbn>
   <preis>12.90</preis>
```

```
</buch>
</bestellung>
</buecher>
```

Es handelt sich um ein einfaches Bestellformular, in dem Kundendaten und Informationen über die einzelnen Bücher hinterlegt wurden. Für dieses Dokument könnte man – und das wurde ausführlich im vorherigen Kapitel gezeigt – eine DTD definieren. Ein zufriedenstellendes Ergebnis würde man damit allerdings nicht erzielen. Denn ein genauer Blick auf das XML-Dokument zeigt, dass dort mit Daten ganz unterschiedlichen Typs gearbeitet wird. Einige Beispiele:

- *Postleitzahl*
- *Preis*
- *ISBN*

Mit einer DTD stößt man schnell an die Grenzen. Denn bekanntermaßen bieten DTDs keinerlei Möglichkeit zu kontrollieren, ob nicht anstelle einer Postleitzahl eine ISBN-Nummer angegeben wurde. Genau hier setzt XML Schema an bzw. hilft dabei, diese Probleme zu umgehen. Das zum gezeigten XML-Dokumente passende XML Schema könnte folgendermaßen aussehen:

Listing 4.7 So sieht ein XML Schema aus.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XML-Schema">
<xsd:element name="buecher" type="formular" />
<xsd:complexType name="formular">
   <xsd:sequence>
     <xsd:element name="kunde" type="kunde" />
     <xsd:element name="bestellung" type="bestellung" />
   </xsd:sequence>
   <xsd:attribute name="bestellnummer" type="xsd:int"
   use="required" />
   <xsd:attribute name="bestelldatum" type="xsd:date"
   use="required" />
</xsd:complexType>
<xsd:complexType name="kunde">
   <xsd:sequence>
      <xsd:element name="name" type="xsd:string" />
      <xsd:element name="strasse" type="xsd:string" />
      <xsd:element name="plz" type="xsd:int" />
      <xsd:element name="ort" type="xsd:string" />
   </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="bestellung">
   <xsd:sequence>
      <xsd:element name="buch">
         <xsd:complexType>
            <xsd:sequence>
               <xsd:element name="kategorie" type="xsd:string" />
               <xsd:element name="titel" type="xsd:string" />
               <xsd:element name="isbn" type="xsd:int" />
               <xsd:element name="preis" type="xsd:decimal" />
            </xsd:sequence>
            <xsd:attribute name="nummer" type="xsd:int"
            use="required" />
         </xsd:complexType>
      </xsd:element>
   </xsd:sequence>
</xsd:complexType>
</xsd:schema>
```

Das XML Schema beginnt genauso wie andere XML-Dokumente, nämlich mit einer XML-Deklaration. Anstelle der üblichen `DOCTYPE`-Deklaration folgt allerdings ein Schema-Element, in dem die Unterelemente mit den Definitionen für den Dokumenttyp enthalten sind. Die einzelnen Elemente des Dokumenttyps werden über entsprechende `element`-Elemente beschrieben.

Auf die genaue Syntax wird im weiteren Verlauf dieses Kapitels noch ausführlich eingegangen. Aber bereits ein erster, früher Blick auf diese Syntax zeigt, dass XML Schema viel mehr als DTDs zu bieten hat. Sehen Sie sich dazu die Syntax des `plz`-Elements an.

```
<xsd:element name="plz" type="xsd:int" />
```

Über `xsd:int` wird angegeben, dass als Werte dieses Elements ausschließlich Integer-Zahlen erlaubt sind.[1] Und eine Postleitzahl ist eben exakt ein solcher Integer-Wert.

4.2.1 XML Schema validieren

Wie das Beispiel zeigte, handelt es sich bei einer solchen Schema-Datei selbst um ein XML-Dokument. Und genau das ist ein riesiger Vorteil. Denn anders als eine DTD kann man ein XML Schema validieren. Endlich hat man also die Möglichkeit, auch die Definition der Dokumenttypen auf syntaktische Korrektheit hin zu überprüfen.

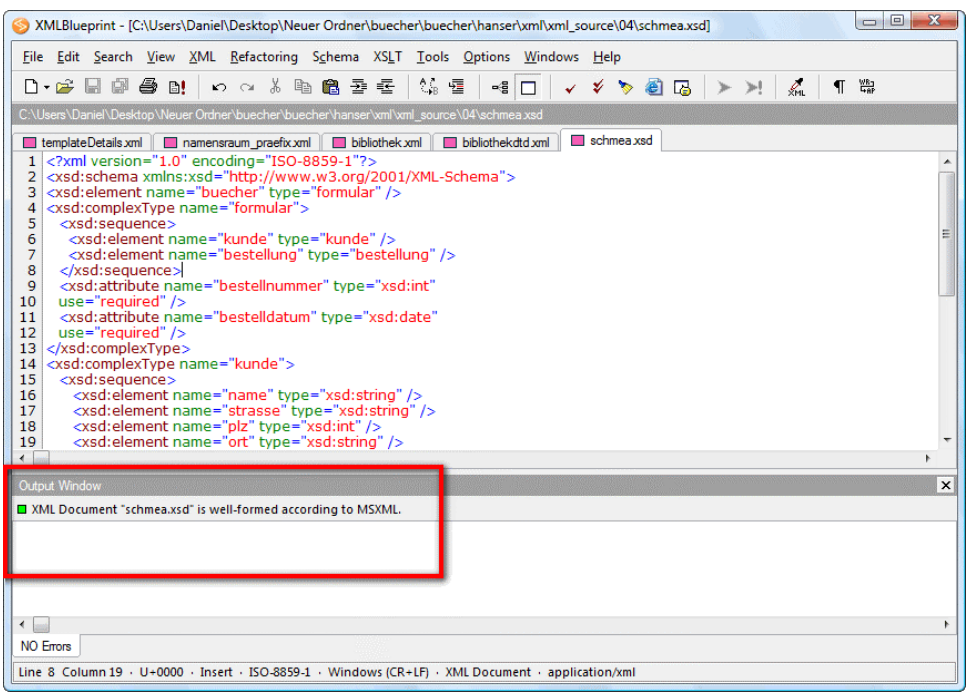

Abbildung 4.3 Das XML Schema ist syntaktisch korrekt.

[1] Wobei ein Integer ein ganzzahliger Wert ist.

Gerade nämlich diese bei DTDs fehlende Überprüfung hat sich als schwerwiegende Fehlerquelle herauskristallisiert. Für die Kontrolle der Syntax wird bei XML Schema zudem keine Spezialsoftware benötigt. Vielmehr kann die Syntax mit jedem XML-Validator überprüft werden.

4.2.2 Schema und Dokument verknüpfen

Nachdem Dokument und Schema erstellt wurden, müssen diese mittels einer passenden Deklaration zusammengefügt werden. Die Angaben dazu werden innerhalb des XML-Dokuments definiert, das dem XML Schema entsprechen soll. Im aktuellen Beispiel könnte das folgendermaßen aussehen:

Listing 4.8 Die Deklaration wird gesetzt.

```
<buecher xmlns="htpp://www.hanser.de/bestellung/"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.hanser.de/bestellung/"
                            file:bestellung.xsd
bestellnummer="10" bestelldatum="01.02.2010">
```

In dieser Syntax wird innerhalb der Dokumentinstanz das Wurzelelement `buecher` mit einem Attribut

```
xsi:schemaLocation
```

aus dem Namensraum `http://www.w3.org/2001/XMLSchema-instance` versehen. Namensräume weisen den XML-Prozessor darauf hin, wo er das Schema für die Validierung findet.

Bei dem Wurzelelement eines XML-Schema-Dokuments handelt es sich immer um `xsd:schema`. Dadurch kann der Parser sofort erkennen, dass es sich um ein Schema für ein Inhaltsmodell handelt. Innerhalb des `xsd:schema`-Wurzelelements gibt es verschiedene Unterelemente, zu denen einige mit dem Namen `xsd:element` und andere mit den Namen `xsd:simpleType` und `xsd:complexType` gehören.

Durch das `xsd`-Präfix wird darauf hingewiesen, dass die verwendeten Namen zum Namensraum von XML Schema gehören. Innerhalb dieses Namensraums sind alle Elemente enthalten, die sich für ein XML Schema eignen. Die Zuordnung zu eben diesem Namensraum erfolgt über die Deklaration

```
xmlns:xsd="http://www.w3.org/2001/XML-Schema"
```

innerhalb von `xsd:schema`. Neben `xsd` können Sie auch einen anderen Namensraum verwenden. Üblicherweise belässt man es allerdings bei `xsd`.

Den Namen der vordefinierten Datentypen, die übrigens auf den folgenden Seiten noch ausführlich vorgestellt werden, stellt man ebenfalls das Präfix voran. Anstelle von `string` verwendet man also `xsd:string`.

Für den Einsatz innerhalb einer Dokumentinstanz, die einem solchen Schema entspricht, gibt es noch einen weiteren Namensraum. Identifiziert wird dieser über den folgenden URI:

```
http://www.w3.org/2001/XMLSchema-Instance
```

Verknüpft wird dieser Namensraum normalerweise durch das Präfix `xsi`.

4.3 Mit Kommentaren arbeiten

Eine gute Kommentierung hilft, den Programmcode übersichtlicher zu gestalten. (Auf diesen Aspekt wurde in diesem Buch bereits mehrfach hingewiesen.)

Da es sich bei XML Schema um ganz normale XML-Syntax handelt, kann man dort zunächst einmal die aus XML bekannte Form der Kommentierung nutzen:

```
<!-- Ich bin ein Kommentar -->
```

Das ist allerdings nur eine Variante, mit der man Kommentare in XML Schema definieren kann. Denn es gibt noch drei spezielle Elemente für die Kommentierung eines XML Schemas.

- `xsd:appinfo` – Liefert Hinweise für Stylesheets oder andere Anwendungen.
- `xsd:annotation` – Definiert eine Anmerkung.
- `xsd:documentation` – Hierüber sollen Personen Hinweise gegeben werden, die das XML Schema lesen.

Ein Beispiel:

Listing 4.9 So lassen sich Kommentare definieren.

```
<xsd:annotation>
   <xsd:documentation xml:lang="de">
      Ich bin ein Kommentar
  </xsd:documentation>
 </xsd:annotation>
```

Über das Attribut `xml:lang` kann man die für die Kommentierung verwendete Sprache angeben. Dieses Attribut ist fakultativ, muss also nicht notiert werden.

4.4 Elementnamen und Elementtypen

In XML Schema sind Elementname und Elementtyp nicht eins zu eins miteinander verbunden, sondern lediglich assoziiert. Auf diese Weise kann man einem Element, je nach seiner Umgebung, andere Attribute und verschiedene Inhaltsmodelle zuordnen.

Elemente eines Zieldokuments werden in XML Schema durch ein Element des Typs `element` beschrieben. Innerhalb dieses Elements wird der Elementname über das Attribut `name` angegeben:

```
<xsd:element name="titel" type="xsd:string" />
```

Um den Elementtyp anzugeben, gibt es zwei Möglichkeiten:

- Als Verweis auf eine benannte Typdefinition.
- Als anonyme Beschreibung eines Inhaltsmodells mit den entsprechenden Attributen.

Beide Varianten haben ihre Vorteile. So können durch globale Typdefinitionen wiederkehrende Inhaltsmodelle wieder verwendet werden. Dem liegt das gleiche Prinzip zugrunde, das bereits im Zusammenhang mit den Parameterentitäten in DTDs vorgestellt wurde. Beispiele zu beiden Varianten folgen im weiteren Verlauf dieses Kapitels.

4.4.1 Elementtypen

Prinzipiell wird in XML Schema zwischen zwei Arten von Elementtypen unterschieden:

- Einfache Typen (*Simple Types*)
- Komplexe Typen (*Complex Types*)

Der Hauptunterschied zwischen beiden Varianten besteht hauptsächlich darin, dass Elemente vom einfachen Typ weder Attribute noch geschachtelte Elemente besitzen dürfen. Elemente mit einem komplexen Typ dürfen aber ebendies enthalten. Attribute sind demnach immer vom einfachen Typ, da für sie exakt die genannten Einschränkungen per XML-Definition gelten.

Der Aufbau für explizite Typdefinitionen und anonyme Definitionen ist identisch. Zunächst wird das Inhaltsmodell durch die entsprechenden Elemente des XML-Schema-Vokabulars beschrieben. Daran schließen sich die Attributdeklarationen an. An dieser Stelle darf nicht der Hinweis fehlen, dass im weiteren Verlauf dieses Kapitels noch ausführlich auf die genannten Aspekte eingegangen wird.

Nachdem das Inhaltsmodell definiert wurde, folgt (optional) die Deklaration der Attribute. Auch dazu im Detail dann später mehr. Attribute können über die in Tabelle 4.1 aufgeführten Attribute genauer spezifiziert werden.

Tabelle 4.1: Attribute, mit denen sich Attribute spezifizieren lassen

Attribut	Wert	Beschreibung
`default`	Ein Wert des angegebenen Typs	Dabei handelt es sich um den Vorgabewert. Dieser wird verwendet, wenn kein anderer Wert für das Attribut in der Dokumentinstanz explizit angegeben wurde.
`name`	Der Name des Attributs	Gibt den Namen des zu deklarierten Attributs an.
`ref`	Deklarierte Attribute	Damit wird auf ein (importiertes) Attribut verwiesen.
`type`	`xsd:string`	Dabei handelt es sich um einen einfachen Typ.

Attribut	Wert	Beschreibung
`use`	`prohibited`, `optional`, `required`	Hierüber kann man angeben, ob ein Attributwert in der Dokumentinstanz verboten (*prohibited*), erforderlich (*required*) oder optional (*optional*) ist.

Der Typ eines Attributs kann nicht nur über `type` bestimmt werden. Auch eine anonyme Variante innerhalb des `attribute`-Elements ist möglich. Dabei sind ausschließlich einfache Typen erlaubt.

Nachdem auf den vorherigen Seiten einige allgemeine Informationen geliefert wurden, geht es jetzt im Detail darum, wie sich Elemente deklarieren lassen. Den Anfang macht dabei die Deklaration komplexer Elemente.

4.4.1.1 Komplexe Elemente

Komplexe Elemente kommen z.B. überall dort zum Einsatz, wo ein Element mehrere Kindelemente umschließen soll. Denn diese komplexen Elemente bieten die Möglichkeit, Wurzelelemente, Kindelemente, Ketten sowie Attribute der Wurzelelemente zusammenhängend zu definieren.

Wie die Deklaration eines komplexen Elements aussehen kann, zeigt die folgende Syntax:

Listing 4.10 So werden komplexe Elemente deklariert.

```
<xsd:complexType name="Briefkopf">
   <xsd:sequence>
      <xsd:element ref="absender"/>
      <xsd:element name="empfaenger" type="Name"/>
      <xsd:element name="betreff" type="xsd:string"
      minOccurs="0" maxOccurs="1"/>
      <xsd:element name="datum" type="xsd:date"
      minOccurs="0" maxOccurs="1"/>
   </xsd:sequence>
</xsd:complexType>
```

Hierbei handelt es sich um die Definition eines Briefkopfs. In der definierten Sequenz werden hintereinander die benötigten Elemente deklariert. Durch `xsd:sequence` wird erreicht, dass jedes Element einmal, keinmal oder mehrfach (Attribute `minOccurs` und `maxOccurs`) vorkommen kann. Die Elemente innerhalb der Sequenz müssen in der angegebenen Reihenfolge definiert werden.

Neben `xsd:sequence` ist auch `xsd:choise` verfügbar. Dabei kann aus einer Liste von Alternativen ein Element ausgewählt werden. Auch dazu wieder ein Beispiel:

Listing 4.11 Ein Beispiel für `xsd:choice`.

```
<xsd:complexType name="Anrede">
   <xsd:choice>
      <xsd:element name="mann" type="mann"/>
      <xsd:element name="frau" type="frau"/>
  </xsd:choice>
</xsd:complexType>
```

Hier wird bestimmt, dass die Anrede sich entweder auf einen Mann oder auf eine Frau bezieht.

Innerhalb dieses Briefkopfs sind verschiedene Elemente enthalten.

- `empfaenger`
- `betreff`
- `datum`

Um dem gerecht zu werden, wird der Briefkopf in Form des oben definierten komplexen Datentyps eingeführt.

```
<xsd:element name="Briefkopf" type="Briefkopf"/>
```

Dieser Datentyp wird dem Element über den Typnamen `Briefkopf` zugewiesen.

Neben den gezeigten Varianten `xsd:choice` und `xsd:sequence` gibt es auch noch `xsd:all`. Darüber lässt sich eine Gruppe von Kindelementen definieren, von denen jedes maximal einmal auftreten darf. Die Reihenfolge der Elemente spielt hier keine Rolle.

4.4.1.2 Einfache Elemente

Es gibt eine Vielzahl an einfachen Typen, die in XML Schema bereits vordefiniert sind. Andere wiederum werden von vordefinierten Typen abgeleitet. Beide Varianten, also sowohl die vordefinierten wie auch die abgeleiteten Typen, können innerhalb von Element- und Attributdeklarationen verwendet werden. Die folgende Übersicht zeigt alle in XML Schema vordefinierten einfachen Typen. Beachten Sie, dass im weiteren Verlauf dieses Kapitels noch einmal auf die Datentypen eingegangen wird.

- `string`
- `normalizedString`
- `token`
- `byte`
- `unsignedByte`
- `base64Binary`
- `hexBinary`
- `integer`
- `positiveInteger`
- `negativeInteger`
- `nonNegativeInteger`
- `nonPositiveInteger`
- `int`
- `unsignedInt`
- `long`
- `unsignedLong`

- short
- unsignedShort
- decimal
- float
- double
- boolean
- time
- dateTime
- duration
- date
- gMonth
- gYear
- gYearMonth
- gDay
- gMonthDay
- Name
- QName
- NCName
- anyURI
- language
- ID
- IDREF
- IDREFS
- ENTITY
- ENTITIES
- NOTATION
- NMTOKEN
- NMTOKENS

Auch zu den einfachen Elementen darf natürlich nicht ein entsprechendes Beispiel fehlen. Einfache XML-Elemente dürfen weder Kindelemente besitzen noch Attribute enthalten. (Beachten Sie, dass ein leeres Element mit einem Attribut ein komplexes Element ist.)

Listing 4.12 So sieht ein einfaches Element aus.

```
<xsd:simpleType name="Name">
   <xsd:restriction base="xsd:string">
      <xsd:length value="10"/>
   </xsd:restriction>
```

```
</xsd:simpleType>
<xsd:element name="absender" type="Name"/>
```

Wie das Beispiel zeigt, wird der im Namensraum von XML Schema vordefinierte einfache Datentyp `string` verwendet. Aus diesem Grund wird in diesem Beispiel auch jedes Mal das `xsd`-Präfix verwendet.

Zusätzlich zur Angabe eines Datentyps kann innerhalb einer Elementdeklaration eine Angabe über einen Standard- oder einen fixen Wert stehen. Auch dazu dann im weiteren Verlauf dieses Kapitels mehr.

4.4.2 Attributdefinitionen

Der Elementtyp muss nicht unbedingt durch das `type`-Attribut angegeben werden. Ebenso ist auch eine anonyme Angabe innerhalb des Elements möglich. Verwendet wird dafür das Element `attribute`. Innerhalb dieses Elements sind ausschließlich einfache Typdefinitionen möglich.

Tabelle 4.2: Attribute und ihre Werte

Attribut	Wert	Beschreibung
`default`	Ein Wert des entsprechenden Typs	Dabei handelt es sich um den Standardwert.
`name`	Name des Attributs	Bezeichnet den Wert des zu deklarierenden Attributs.
`ref`	Deklarierte Attribute	Es wird auf (importierte) Attribute verwiesen.
`type`	xsd:string ...	Ein einfacher Typ
`use`	prohibited, optional, required	Bestimmt, ob die Angabe eines Attributwerts innerhalb der Dokumentinstanz verboten (prohibited), optional (optional) oder erforderlich (required) ist.

Über das `xsd:attribute`-Element kann man jede Schemadefinition durch entsprechende Attribute erweitern. Das Element lässt sich innerhalb komplexer Datentypen verwenden, es muss allerdings immer hinter den Elementdeklarationen notiert werden.

Listing 4.13 So lassen sich Attribute verwenden.

```
<xsd:complexType name="auto">
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name="hersteller" type="xsd:string"/>
    <xsd:element name="gaenge"  type="xsd:string"/>
    <xsd:element name="kw" type="xsd:string"/>
  </xsd:sequence>
  <xsd:attribute name="farbe" type="xsd:integer"/>
</xsd:complexType>
```

Der Attributdeklaration muss ein Name zugewiesen werden. Dazu wird das Attribut `name` verwendet. Über `type` gibt man den entsprechenden Datentyp an. Dabei kann man entweder auf die vorgegebenen einfachen Datentypen zurückgreifen, die XML Schema bietet, oder man verwendet benutzerdefinierte einfache Datentypen.

Attribute lassen sich nicht verschachteln, sie dürfen also keine anderen Elemente enthalten. Die Deklaration eines Attributs kann mittels einer Referenz auf ein bereits vorhandenes Attribut erfolgen. Verwendet wird dafür das `ref`-Attribut.

Listing 4.14 So werden Referenzen eingesetzt.

```
<xsd:complexType name="Text">
   <xsd:complexContent mixed="true">
      <xsd:restriction base="xsd:anyType">
         <xsd:sequence>
            <xsd:any processContents="lax"
            minOccurs="0" maxOccurs="unbounded"/>
         </xsd:sequence>
         <xsd:attribute ref="xml:lang"/>
      </xsd:restriction>
   </xsd:complexContent>
</xsd:complexType>
```

Beachten Sie, dass die Attributnamen innerhalb eines komplexen Datentyps eindeutig sein müssen. Die Reihenfolge der Attributdeklaration zu einem Datentyp spielt hingegen keine Rolle.

4.5 Datentypen

Große Vorteile gegenüber DTDs hat XML Schema vor allem hinsichtlich der Datentypen. In diesem Buch werden die wichtigsten Aspekte der Datentypen in XML Schema betrachtet. Ausführlichere Informationen zu den Datentypen in XML Schema finden Sie z.B. unter *http://www.edition-w3c.de/TR/2001/REC-xmlschema-2-20010502/*.

Das abstrakte Datenmodell, das XML Schema zugrunde liegt, basiert auf einer hierarchischen Anordnung der Datentypen. Diese Datentypen gehen von einer Wurzel aus, von der sie sich auf zwei verschiedene Arten entfalten können.

- Da wäre zunächst das Prinzip der Einschränkung. Diese Einschränkung wird durch das Element `xsd:restriction` realisiert.
- Bei dem anderen Verfahren handelt es sich um eine Erweiterung. Realisiert wird diese mit dem Element `xsd:extension`. Dadurch können in ein Inhaltmodell weitere Komponenten eingefügt werden.

Beide Verfahren beziehen sich explizit auf einen angegebenen Basistyp als Ausgangspunkt.

An der Wurzel des Datentypbaums wird ein sogenannter Urtyp angenommen. Ansprechen lässt der dieser über `anytyp`.

Abbildung 4.4 zeigt das Schema der eingebauten Datentypen, wie es vom W3C vorgesehen ist.

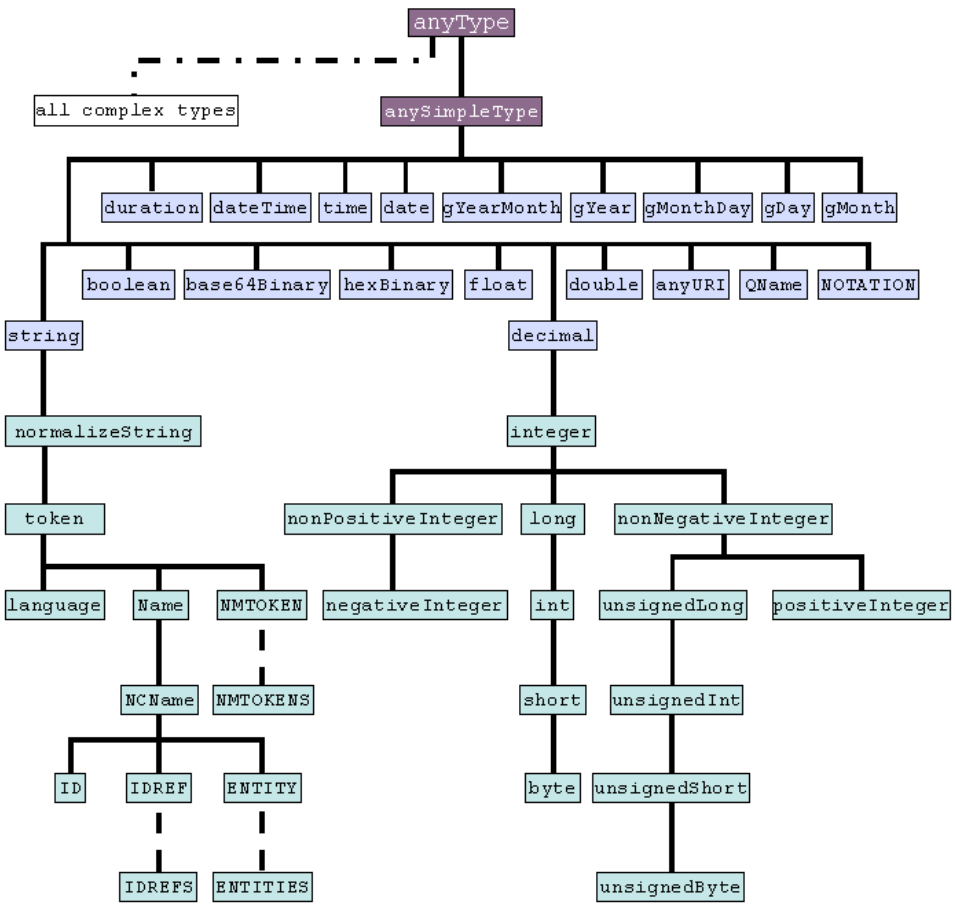

Abbildung 4.4 Bildquelle: http://www.edition-w3c.de/TR/2001/REC-xmlschema-2-20010502/

Insgesamt umfasst XML Schema Teil 2 stolze 44 vordefinierte Datentypen, von denen 19 den Status primitiver Datentypen besitzen.

4.5.1 Alle Datentypen in der Übersicht

Da es fast unmöglich ist, alle in XML Schema verfügbaren Datentypen im Kopf zu haben, folgt an dieser Stelle eine Übersicht, in der die möglichen Datentypen zusammengefasst sind. Eine vollständige Zusammenfassung aller in XML Schema vorhandenen Datentypen finden Sie auf den Seiten des W3C unter *http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/*.

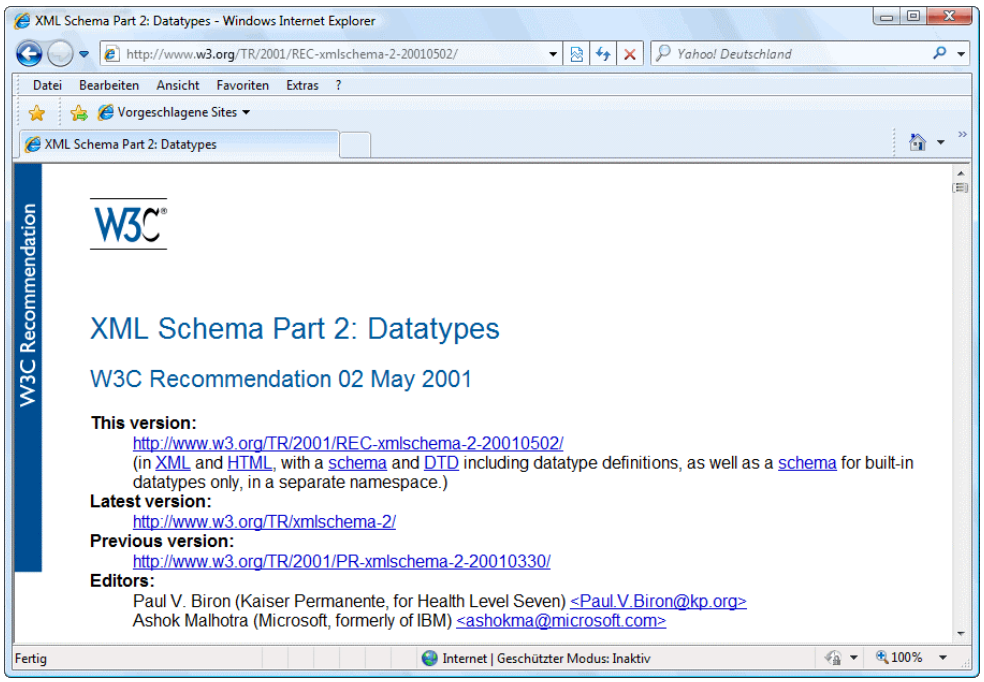

Abbildung 4.5 Dort gibt es ausführliche Informationen zu den Datentypen.

Prinzipiell dürften die folgenden Ausführungen aber ausreichen.

Tabelle 4.3: Logische Datentypen

Datentyp	Beschreibung
`boolean`	Boolescher Wert (`true` oder `false` oder `1` oder `0`)

Tabelle 4.4: Binäre Datentypen

Datentyp	Beschreibung
`base64Binary`	Binäre Daten, die Base64-kodiert sind.
`hexBinary`	Beliebige Daten, die hexadezimal kodiert wurden.

Tabelle 4.5: Zahlentypen

Datentyp	Beschreibung
`byte`	Eine Ganzzahl zwischen `127` und `-128`. Wenn das Vorzeichen fehlt, wird von einer positiven Zahl ausgegangen.
`decimal`	Eine beliebig genaue Dezimalzahl. Wenn das Vorzeichen fehlt, wird von einer positiven Zahl ausgegangen.
`double`	Eine 64-Bit-Gleitkommazahl mit einer doppelten Genauigkeit nach IEEE 754-1985.

Datentyp	Beschreibung
float	Eine 32-Bit-Gleitkommazahl mit einer einfachen Genauigkeit nach IEEE 754-1985.
int	Eine Ganzzahl zwischen 2147483647 und -2147483648.
integer	Eine beliebig große Ganzzahl. Fehlt das Vorzeichen, wird von einer positiven Zahl ausgegangen.
long	Eine Ganzzahl zwischen 9223372036854775807 und -9223372036854775808. Wenn das Vorzeichen fehlt, wird von einer positiven Zahl ausgegangen.
negativeInteger	Eine negative Ganzzahl mit einer beliebigen Größe.
NonNegativeInteger	Eine nichtnegative Ganzzahl mit einer beliebigen Größe. Die Angabe des Plusvorzeichens ist optional; das Vorzeichen - ist verboten.
NonPositiveInteger	Eine nichtpositive Ganzzahl mit einer beliebigen Größe. Die Angabe des Minusvorzeichens ist optional, das Vorzeichen + ist verboten.
positiveInteger	Eine positive Ganzzahl mit einer beliebigen Größe.
short	Eine Ganzzahl zwischen 32767 und -32768. Wenn das Vorzeichen fehlt, wird von einer positiven Zahl ausgegangen.
unsignedByte	Eine Ganzzahl zwischen 0 und 255, die kein Vorzeichen besitzt.
unsignedInt	Eine Ganzzahl zwischen 0 und 4294967295, die kein Vorzeichen besitzt.
unsignedLong	Eine Ganzzahl zwischen 0 und 18446744073709551615, die kein Vorzeichen besitzt.
unsignedShort	Eine Ganzzahl zwischen 0 und 65535, die kein Vorzeichen besitzt.

Tabelle 4.6: Zeichenfolgen

Datentyp	Beschreibung
anyURI	Steht für eine Zeichenfolge, die eine Referenz auf eine Ressource in Form eines URI liefert. Dieser URI kann relativ oder absolut sein und darf einen Fragment-Identifizierer enthalten.
language	Dabei handelt es sich um einen Code für eine natürliche Sprache. Dieser besteht meistens aus einer zweistelligen Zeichenfolge nach RFC 1766. Typische Beispiele sind en für England und de für Deutschland.
normalizedString	Das ist eine Zeichenfolge, in der keine der folgenden Leerräume enthalten sind: #x9 (Tabulator), #xD (Wagenrücklauf), #xA (Zeilenvorschub)
string	Eine Folge von Unicode-Zeichen in XML.
token	Bei dieser Zeichenfolge handelt es sich um eine lexikalische Einheit, aus der sich normalisierte Zeichenfolgen ableiten lassen. Leerräume sind darin nicht enthalten.

Tabelle 4.7: Datum und Uhrzeit

Datentyp	Beschreibung
`date`	Dabei handelt es sich um ein Kalenderdatum wie beispielsweise `2010-02-01`.
`dateTime`	Es wird ein bestimmter Zeitpunkt unter Berücksichtigung der entsprechenden Zeitzone. Verwendet wird dabei das ISO-Format 8601, das folgendermaßen aussieht: `CCYY-MMDDThh:mm:ss`.
`duration`	Ein Wert für die Zeitdauer. Verwendet wird dabei das ISO-Format 8601, das folgendermaßen aussieht: `PnYn MnDTnH nM nS`. Dabei steht `nY` für die Anzahl der Jahre, `nM` für die Monate, `nD` für die Tage, `nH` für die Stunden, `nM` für die Minuten und `nS` für die Sekunden. Durch das Zeichen `T` werden die Datums- und die Zeitangaben voneinander getrennt.
`gDay`	Es handelt sich hierbei um den Tag nach dem gregorianischen Kalender, wie beispielsweise der 10. Tag des Monats.
`gMonth`	Es handelt sich hierbei um den Monat nach dem gregorianischen Kalender. Dabei steht zum Beispiel die `1` für den Januar.
`gMonthDay`	Es handelt sich hierbei um den Tag im Monat nach dem gregorianischen Kalender. Dabei steht zum Beispiel die `05-01` für den 5. Januar.
`GYear`	Es handelt sich hierbei um das Jahr nach dem gregorianischen Kalender.
`gYearMonth`	Es handelt sich hierbei um einen Monat in einem Jahr nach dem gregorianischen Kalender. So steht beispielsweise `2020-01` für den Januar 2010.
`time`	Eine Zeitangabe an einem beliebigen Tag. Der Wert bezieht sich auf die koordinierte Weltzeit.

Tabelle 4.8: XML-Datentypen

Datentyp	Beschreibung
Name	XML-Name
NCName	Ein Name, in dem kein Doppelpunkt enthalten ist. Es handelt sich also um einen lokalen Namen oder das Namensraumpräfix in einem qualifizierten Namen.
NOTATION	Der Name einer Notation.
QName	Dabei handelt es sich um einen qualifizierten XML-Namen.

In Tabelle 4.9 sind weitere Datentypen enthalten. Diese wurden eingeführt, um die Kompatibilität zwischen DTDs und XML Schema aufrechterhalten zu können. Aus diesem Grund sollten sie ausschließlich als Datentypen für Attributwerte verwendet werden.

Tabelle 4.9: Datentypen für die Kompatibilität zwischen DTDs und XML Schema

Datentyp	Beschreibung
ENTITIES	Eine Liste von Entitäten, die jeweils durch ein Leerzeichen getrennt werden.
ENTITY	Der Name einer allgemeinen Entität.
ID	Ein eindeutiger Identifizierer eines Elements.
IDREF	Der Verweis auf eine ID eines Elements.
IDREFS	Eine Liste von Verweisen auf IDs, die jeweils durch ein Leerzeichen getrennt sind.
NMTOKEN	Namens-Token[2]
NMTOKENS	Eine Liste von Namens-Token, die jeweils durch ein Leerzeichen getrennt sind.

4.5.2 Von Werteräumen, lexikalischen Räumen und Facetten

Datentypen werden von der XML-Schema-Spezifikation als Kombination aus einem Werteraum, einem lexikalischen Raum und einem Satz von Eigenschaften betrachtet. Ein Element, dem ein bestimmter Datentyp zugewiesen wurde, kann daher auf unterschiedliche Arten dargestellt werden kann.

Die Spezifikation unterscheidet zunächst einmal zwischen fünf grundlegenden Facetten, über die sich die Eigenschaften von Datentypen benennen lassen.

Tabelle 4.10: Das sind die möglichen Facetten.

Facette	Beschreibung
bounded	Der Datentypen kann auf einen bestimmten Wertebereich beschränkt sein.
cardinality	Dabei handelt es sich um die Anzahl der Werte innerhalb eines Werteraums. Diese Anzahl kann begrenzt oder unbegrenzt sein.
equal	Die Werte innerhalb eines Werteraums lassen sich auf Gleichheit und Ungleichheit zu anderen Werten überprüfen.
numeric	Dadurch wird angegeben, ob es sich um einen numerischen Datentyp handelt oder nicht.
ordered	Werte werden immer dann als geordnet eingestuft, wenn sie zu einem bestimmten Werteraum in einer mathematischen Relation stehen.

Interessant ist vor allem die Tatsache, dass sich aus den Standard-Datentypen bei Bedarf weitere Datentypen ableiten lassen. Hierfür kommet eine Reihe einschränkender Facetten

[2] Ein Wert bei NMTOKEN ist eine Zeichenkette, die mit einem Buchstaben beginnt. Enthalten dürfen darin Zahlen, Buchstaben und bestimmte Interpunktionszeichen sein.

zum Einsatz, durch die sich der Wertebereich eines vorgegebenen Datentyps eingrenzen lässt. Dabei gibt es einige Einschränkungen, die in Tabelle 4.10 deutlich gemacht werden.

Tabelle 4.11: Facetten und ihre Einschränkungen

Facette	Beschreibung
enumeration	Eine ungeordnete Liste erlaubter Werte.
fractionDigits	Die maximal erlaubte Anzahl von Nachkommastellen.
length	Die Anzahl der Längeneinheiten. Was genau eine Längeneinheit ist, hängt dabei vom jeweiligen Typ ab.
maxExclusive	Das ist der obere Grenzwert. Damit ein Wert gültig ist, muss er kleiner sein.
maxInclusive	Das ist der maximale Wert.
maxLength	Die Anzahl der Längeneinheiten. Was genau eine Längeneinheit ist, hängt vom jeweiligen Typ ab.
minExclusive	Das ist der untere Grenzwert. Damit ein Wert gültig ist, muss er größer sein.
minInclusive	Das ist der minimale Wert.
minLength	Die Anzahl der Längeneinheiten. Was genau eine Längeneinheit ist, hängt dabei vom jeweiligen Typ ab.
totalDigits	Die Gesamtzahl der Stellen einer Dezimalzahl.
pattern	Das ist ein regulärer Ausdruck, der angibt, welche Zeichen für die Darstellung des Werts verwenden werden dürfen.
whitespace	Hierüber wird angegeben, wie Leerraum behandelt werden soll. Mögliche Werte sind `preserve`, `collapse` und `replace`.

4.5.3 Ableitungen durch Einschränkungen

In der Spezifikation von XML Schema ist exakt beschrieben, wie eine Ableitung durch Einschränkungen auszusehen hat. Verwendet wird dafür eine `simpleType`-Definition. Dabei bietet jeder Datentyp eine gewisse Anzahl möglicher Facetten. (Ausführliche Informationen dazu finden Sie auch im nächsten Abschnitt.) So kennt der Datentyp `string` beispielsweise die folgenden Einschränkungen:

- `length`
- `minLength`
- `maxLength`
- `pattern`
- `enumeration`
- `whiteSpace`

Will man z.B. erreichen, dass ein Element für eine Postleitzahl mindestens vier (für Österreich) und höchstens fünf (für Deutschland) Zeichen enthält, könnte man das folgendermaßen umsetzen:

Listing 4.15 So lassen sich Postleitzahlen überprüfen.

```
<xsd:simpleType name="plz">
   <xsd:restriction base="xsd:string">
      <xsd:minLength value="4"/>
      <xsd:maxLength value="5"/>
   </xsd:restriction>
</xsd:simpleType>
```

Über das `name`-Attribut wird dem Datentyp zunächst der Name `plz` zugewiesen. Anschließend wird der Datentyp benannt, der als Basis für die weiteren Ableitungen gelten soll. Verwendet wird dafür jeweils folgendes Element:

```
xsd:restriction
```

Diesem Element wird über das `base`-Attribut der entsprechende Datentyp, im Beispiel also `string`, zugewiesen. Im aktuellen Beispiel wurde jeweils die minimale (`minLength`) und die maximale (`maxLength`) Zeichenzahl bestimmt.

Wenn für ein Element oder die Werte von Attributen die Anzahl der Einträge auf ganz bestimmte Werte begrenzt ist, lassen sich diese mittels des `enumeration`-Elements definieren. Auch hierzu wieder ein Beispiel:

Listing 4.16 Hier wurden die Werte definiert.

```
<xsd:simpleType name="monate">
   <xsd:restriction base="xsd:string">
      <xsd:enumeration value="Januar"/>
      <xsd:enumeration value="Februar"/>
      <xsd:enumeration value="März"/>
      <xsd:enumeration value="April"/>
      <xsd:enumeration value="Mai"/>
      <xsd:enumeration value="Juni"/>
      <xsd:enumeration value="Juli"/>
      <xsd:enumeration value="August"/>
      <xsd:enumeration value="September"/>
      <xsd:enumeration value="Oktober"/>
      <xsd:enumeration value="November"/>
      <xsd:enumeration value="Dezember"/>
   </xsd:restriction>
</xsd:simpleType>
```

4.5.4 Facetten verwenden

Es wurde bereits darauf hingewiesen, dass sich die für jeden einfachen Typ erlaubten Werte durch den Einsatz einer oder mehrerer Facetten einschränken lassen. Die beiden Tabellen Tabelle 4.12 und Tabelle 4.13 enthalten alle vordefinierten einfachen Typen von XML Schema mitsamt den darauf anwendbaren Facetten.

Tabelle 4.12: Einfache Typen in XML Schema

einfacher Typ	**Facette**					
	length	min-Length	max-Length	pattern	enumeration	white-Space
string	X	X	X	X	X	X
normalizedString	X	X	X	X	X	X
token	X	X	X	X	X	X
byte				X	X	X
unsignedByte				X	X	X
base64Binary	X	X	X	X	X	X
hexBinary	X	X	X	X	X	X
integer				X	X	X
positiveInteger				X	X	X
negativeInteger				X	X	X
nonNegativeInteger				X	X	X
nonPositiveInteger				X	X	X
int				X	X	X
unsignedInt				X	X	X
long				X	X	X
unsignedLong				X	X	X
short				X	X	X
unsignedShort				X	X	X
decimal				X	X	X
float				X	X	X
double				X	X	X
boolean				X		X
time				X	X	X
dateTime				X	X	X
duration				X	X	X
date				X	X	X
gMonth				X	X	X
gYear				X	X	X
gYearMonth				X	X	X
gDay				X	X	X
gMonthDay				X	X	X

einfacher Typ	Facette					
	length	min-Length	max-Length	pattern	enumeration	white-Space
QName	X	X	X	X	X	X
NCName	X	X	X	X	X	X
anyURI	X	X	X	X	X	X
language	X	X	X	X	X	X
ID	X	X	X	X	X	X
IDREF	X	X	X	X	X	X
IDREFS	X	X	X		X	X
ENTITY	X	X	X	X	X	X
ENTITIES	X	X	X		X	X
NOTATION	X	X	X	X	X	X
NMTOKEN	X	X	X	X	X	X
NMTOKENS	X	X	X		X	X

Die in Tabelle 4.13 aufgeführten Facetten können nur auf geordnete einfache Typen angewendet werden. Aus diesem Grund sind in dieser Tabelle auch nicht alle einfachen Typen aufgeführt. Insgesamt liefert die Tabelle einen Überblick der einfachen Typen und der darauf anwendbaren Facetten.

Tabelle 4.13: Und weitere einfache Typen

einfache Typen	Facetten					
	maxInclusive	maxExclusive	min-Inclusive	min-Exclusive	totalDigits	fractionDigits
byte	X	X	X	X	X	X
unsignedByte	X	X	X	X	X	X
integer	X	X	X	X	X	X
positiveInteger	X	X	X	X	X	X
negativeInteger	X	X	X	X	X	X
nonNegativeInteger	X	X	X	X	X	X
nonPositiveInteger	X	X	X	X	X	X

einfache Typen	Facetten					
	maxIn-clusive	maxEx-clusive	min-Inc-lusive	min-Exclusi-ve	total-Digits	fract-ion-Digits
int	X	X	X	X	X	X
unsignedInt	X	X	X	X	X	X
long	X	X	X	X	X	X
unsigned-Long	X	X	X	X	X	X
short	X	X	X	X	X	X
unsigned-Short	X	X	X	X	X	X
decimal	X	X	X	X	X	X
float	X	X	X	X		
double	X	X	X	X		
time	X	X	X	X		
dateTime	X	X	X	X		
duration	X	X	X	X		
date	X	X	X	X		
gMonth	X	X	X	X		
gYear	X	X	X	X		
gYearMonth	X	X	X	X		
gDay	X	X	X	X		
gMonthDay	X	X	X	X		

4.5.5 Mit regulären Ausdrücken arbeiten

Mittels regulärer Ausdrücke können Zeichenketten in ihrer Struktur beschrieben werden. Sie kennen solche regulären Ausdrücke sicherlich aus verschiedenen Programmiersprachen. Zudem werden sie innerhalb vieler Editoren zum Suchen und Ersetzen verwendet. (Die meisten Anwender nutzen diese Möglichkeit allerdings kaum.)

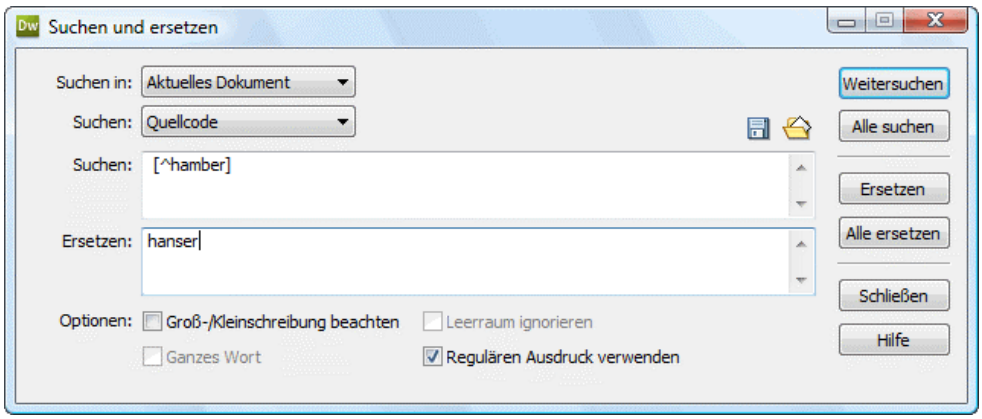

Abbildung 4.6 Suchen mit regulären Ausdrücken.

In XML Schema werden reguläre Ausdrücke bei der Facettenbildung innerhalb des Elements `xsd:pattern value=""` als Wert von `value` eingesetzt, um Zeichenketten genauer zu beschreiben.

Interessant sind reguläre Ausdrücke vor allem, wenn man exakt beschreiben will, welche Daten an einer bestimmten Stelle erwartet werden. Hier einige typische Einsatzgebiete:

- Telefonnummern
- ISBN
- Postleitzahlen

So können Sie beispielsweise bestimmen, dass bei einem Element `postleitzahl` fünf Ziffern erwartet werden. Weder sind dort mehr, noch sind weniger als fünf Ziffern erlaubt. Ideal sind reguläre Ausdrücke aber auch für die Überprüfung von Bestellnummern. Dazu ebenfalls ein Beispiel:

Listing 4.17 Die Bestellnummern werden überprüft.

```
<xsd:simpleType name="bestellnr">
   <xsd:restriction base="xsd:string">
      <xsd:pattern value="[0-9]{10}"/>
      <xsd:length value="5"/>
   </xsd:restriction>
</xsd:simpleType>
```

Durch `[0-9]` wird festgelegt, dass Ziffern zwischen `0` und `9` erlaubt sind. Ebenso kann anhand von regulären Ausdrücken auch das Vorhandensein einer deutschen E-Mail-Adresse überprüft werden.

Listing 4.18 So wird überprüft, ob eine deutsche E-Mail-Adresse vorhanden ist.

```
<xs:simpleType>
   <xs:restriction base="xs:string">
      <xs:pattern value=".*@.*\.de"/>
   </xs:restriction>
</xs:simpleType>
```

Durch diese Syntax wird kontrolliert, ob der String ein @-Zeichen enthält, dem eine beliebige Zeichenkette folgt. Am Ende des Strings muss aber in jedem Fall die Zeichenfolge `.de` stehen. Die angegebene Bedingung ist nur erfüllt, wenn die E-Mail-Adresse folgendem Schema entspricht:

```
zeichenkette@zeichenkette.de
```

Welche regulären Ausdrücke es noch gibt, das zeigt Tabelle 4.14. Ihnen wird möglicherweise auffallen, dass sich XML Schema für reguläre Ausdrücke einer Sprache bedient, die Ihnen von Perl her bekannt ist. Und in der Tat stimmen die regulären Ausdrücke zwischen XML Schema und Perl größtenteils miteinander überein. Die verwendete Sprache orientiert sich an den Unicode Regular Expression Guidelines, die unter *http://www.unicode.org/unicode/reports/tr18/* zu finden sind.

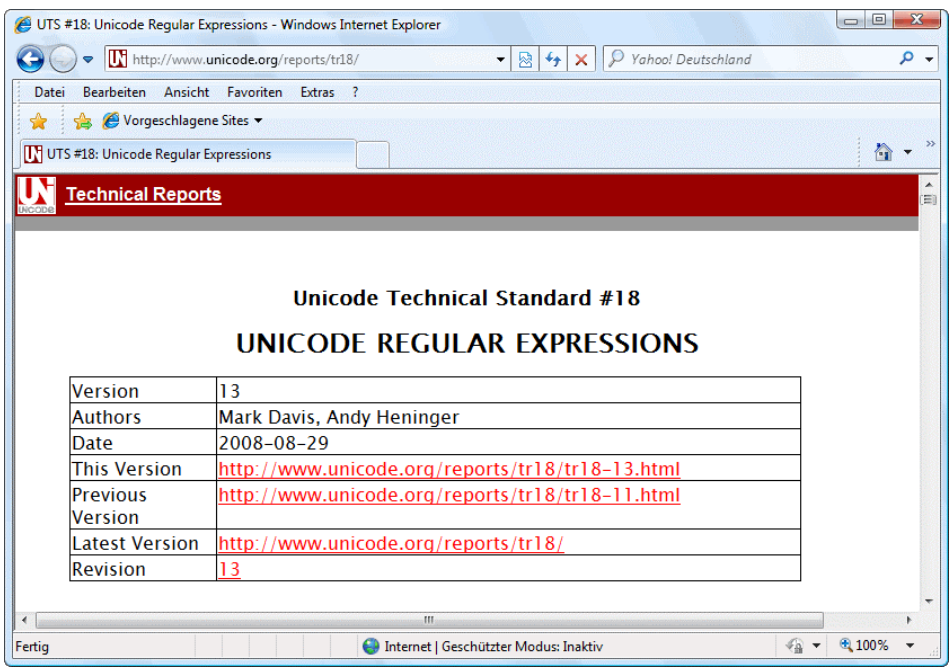

Abbildung 4.7 Hier können Sie sich einen Überblick verschaffen.

Auf dieser Seite finden Sie zu den einzelnen regulären Ausdrücken interessante Beispiele.

Tabelle 4.14 Ein Überblick der regulären Ausdrücke

Regulärer Ausdruck	Beschreibung
*	0 – n Vorkommen des vorherigen Zeichens bzw. der vorherigen Gruppe.
.	Ein beliebiges XML-Zeichen.
?	0 – 1 Vorkommen des vorherigen Zeichens bzw. der vorherigen Gruppe.
+	1 – n Vorkommen des vorherigen Zeichens bzw. der vorherigen Gruppe.

Regulärer Ausdruck	Beschreibung
`()`	Ermöglicht das Gruppieren von Zeichen bzw. Ausdrücken.
`{}`	Wiederholungsanzahl oder Kategoriezeichen
`[]`	Es handelt sich um Zeichenklassen, aus denen jeweils ein Zeichen ausgewählt wird.
`|`	Das ist eine Oder-Bedingung bei Gruppierungen
`\t`	Tabulatorstopp
`\n`	Zeilenumbruch
`\r`	Wagenrücklauf
`\s`	Leerstelle
`\S`	Nicht-Leerstelle
`\w`	Buchstabe
`\W`	Nicht-Buchstabe
`\d`	Ziffer
`\D`	Nicht-Ziffer
`\`	Das ist das Escape-Zeichen, dank dem Sonderzeichen in ihrer eigentlichen Form (also nicht als Sonderzeichen) genutzt werden können.
`\{L}`	Ein Buchstabe jeder beliebigen Sprache.
`\{Lu}`	Ein Großbuchstabe jeder beliebigen Sprache.
`\{P}`	Das Punktzeichen
`\{Sc}`	Ein Währungszeichen jeder beliebigen Sprache.
`\{N}`	Eine römische Zahl oder ein Bruch.
`\{Nd}`	Eine Ziffer aus jeder beliebigen Sprache.

Eine ausführliche Beschreibung der einzelnen regulären Ausdrücke gibt es auf der Seite *http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/#regexs*. Tabelle 4.15 zeigt ein Beispiel für den Einsatz regulärer Ausdrücke.

Tabelle 4.15: Typische Beispiele für reguläre Ausdrücke

Ausdruck	Beispiel	Beschreibung
`*`	`/bald*/`	Findet `bal`, `bald`, `baldd` und `baldddd`.
`+`	`/bald+/`	Findet `bald`, `baldd` und `baldddd`, also das letzte Zeichen vor dem Stern mindestens einmal oder beliebig oft wiederholt.
`?`	`Ba?ld`	Das voranstehende Zeichen ist optional. Findet `alsbald`, aber nicht `abbild`.
`.`	`/.aus/`	Findet `Applaus` und `Geradeaus`. Der Punkt steht also für ein beliebiges Zeichen an einer bestimmten Stelle.
`\d`	`/\d+\B/`	Es werden beliebige Zahlen (`0` bis `9`) gefunden.

Ausdruck	Beispiel	Beschreibung
\D	\D	Ein beliebiges Zeichen, das keine Ziffer ist. Findet S in S04.
\n	/\n/	Findet ein Zeilenvorschubzeichen.
\r	/\r/	Findet ein Wagenrücklaufzeichen.
{n}	e{2}	Genau n Vorkommen des vorangestellten Zeichens. Findet leer und beer, aber nicht bekehre.
\t	/\t/	Findet ein Tabulatorzeichen.
\v	/\v/	Findet ein vertikales Tabulatorzeichen.
\s	/\s/	Findet alle Whitespace-Arten.
\S	/\S+/	Findet ein beliebiges druckbares Zeichen, das kein Whitespace ist.
\w	/\w+/	Findet ein beliebiges alphanumerisches Zeichen, inklusive des Unterstrichs.
\W	/\W/	Findet ein beliebiges Zeichen, dieses darf allerdings nicht alphanumerisch sein.

4.6 Die Dokumentstruktur definieren

Die vorgestellten Typdefinitionen werden zum Erzeugen von Datentypen verwendet. Deklarationen wiederum dienen dazu, Elementen oder Attributen Namen zuzuweisen und mit den entsprechenden Datentypen zu verknüpfen. Darüber hinaus lassen sich auch Standardwerte setzen und eine Anzahl von Einschränkungen hinzufügen. Im Unterschied zu einer DTD ist die Zuweisung von Datentypen zu Elementen und Attributen in XML Schema enorm wichtig.

4.6.1 Elemente deklarieren

Elemente werden über das `xsd:element`-Element deklariert. Die Syntax ist dabei immer die gleiche. Über das `name`-Attribut wird dem Element zunächst ein Name zugewiesen. Bei der Namensvergabe sind die in XML üblichen Regeln zu beachten. In XML Schema wird zwischen einfachen und komplexen Elementen unterschieden.

4.6.1.1 Einfache Elemente

Durch `simpleType` werden einfache Elemente deklariert, die bereits vordefinierte Typen wie `xsd:string` u.ä. enthalten. Ebenso können auch eigene einfache Typen geschaffen werden, die dann allerdings auf einfachen und bereits vordefinierten Typen basieren und diese einschränken.

Listing 4.19 Ein typisches einfaches Element

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

```
...
   <xsd:simpleType name="waehrung">
      <xsd:restriction base="xsd:string">
         <!--Facette -->
      </xsd:restriction>
   </xsd:simpleType>
</xsd:schema>
```

4.6.1.2 Komplexe Elemente

Mittels komplexer Datentypen können die Kindelemente eines Elements beschrieben werden. Im folgenden Beispiel wird ein Element als komplexer Datentyp eingeführt.

```
<xsd:element name="Name" type="name">
```

Das wird erreicht, indem eigens für das betreffende Schema ein komplexer Datentyp entworfen wird, der dem Element über den entsprechenden Typnamen zugewiesen wird. Im vorliegenden Fall handelt es sich dabei um `name`.

Listing 4.20 Ein komplexes Element wird definiert.

```
<xsd:complexType name="name">
   <xsd:sequence>
      <xsd:element name="Vorname" type="xsd:string"/>
      <xsd:element name="Nachname" type="xsd:string"/>
   </xsd:sequence>
</xsd:complexType>
```

Durch die Definition eines komplexen Datentyps wird angegeben, wie die einzelnen Unterelemente gruppiert werden. Im aktuellen Beispiel wird dafür der Kompositor `xsd:sequence` verwendet. Ausführliche Informationen zu Kompositoren finden Sie im weiteren Verlauf dieses Kapitels.

Beim Blick auf das Beispiel fällt auf, dass hintereinander zwei Elemente deklariert werden. Beiden wird neben dem Elementnamen auch der entsprechende Datentyp (`xsd:string`) zugewiesen.

4.6.2 Attribute deklarieren

Schemadefinitionen lassen sich durch die Deklaration von Attributen erweitern. Verwendet wird dafür das Element `xsd:attribute`. Dieses Element kann innerhalb komplexer Datentypen eingesetzt werden. Dabei muss es allerdings immer hinter den Elementdeklarationen stehen. Zunächst ein Beispiel für eine Attributdefinition:

Listing 4.21 So werden Attribute definiert.

```
<xsd:complexType name="artikel">
   <xsd:sequence>
      <xsd:element name="titel" type="xsd:string"/>
      <xsd:element name="teaser" type="xsd:string"/>
      <xsd:element name="inhalt" type="xsd:string"/>
   </xsd:sequence>
   <xsd:attribute name="nr" type="xsd:integer"/>
</xsd:complexType>
```

Ein Attribut wird wie ein einfaches Element definiert. Wie das Beispiel zeigt, geschieht die Definition des Attributs außerhalb von `xsd:sequence`, dafür aber innerhalb von `xsd:complexType`. Die Attributdeklaration benötigt einen Namen für das Attribut sowie die Angabe eines Datentyps, dem der Wert des Attributs entsprechen soll. Verwenden lassen sich dafür entweder die einfachen Datentypen von XML Schema oder benutzerdefinierte einfache Datentypen.

Attribute lassen sich in XML Schema weder verschachteln, noch können sie andere Elemente enthalten.

Eine Attributdeklaration kann auch durch die Referenz auf ein bereits definiertes Attribut erfolgen.

```
<xsd:attribute ref="xml:lang"/>
```

Wie in DTDs dürfen Attribute in XML Schema zusätzliche Eigenschaften besitzen. So lassen sich den Attributen beispielsweise Standardwerte zuweisen.

```
<xsd:attribute name="nr" type="xsd:integer" default="0" />
```

Der bei `default` angegebene Wert wird immer dann verwendet, wenn das Attribut nicht gesetzt wurde.

Ebenso kann man dem Attribut auch einen festen Wert zuweisen.

```
<xsd:attribute name="nr" type="xsd:integer" fixed="20" />
```

In einem solchen Fall kann man diesem Attribut keinen anderen Wert zuweisen.

Ähnlich einfach lassen sich solche Attribute definieren, die man zwar angeben kann, aber nicht muss.

```
<xsd:attribute name="nr" type="xsd:integer" use="optional" />
```

Durch diese Syntax wird das Attribut als optional gekennzeichnet.

Das Gegenstück dazu gibt es auch. Man kann für ein Attribut festlegen, dass es angegeben werden muss.

```
<xsd:attribute name="nr" type="xsd:integer" use="required" />
```

4.6.3 Elementvarianten

Bislang haben sich die komplexen Elementtypen aus einfachen Elementtypen zusammengesetzt. So wird im folgenden Beispiel das Element `preiseuro` dem vordefinierten einfachen Datentyp `xsd:decimal` zugeordnet.

```
<xsd:element name="preiseuro" type="xsd:decimal" />
```

Dieser Wert soll nun mit dem Eurozeichen verknüpft werden. Dazu kann man den einfachen Datentyp `xsd:decimal` erweitern. Wie eine solche Anwendung aussieht, zeigt das folgende Beispiel:

Listing 4.22 So lassen sich einfache Datentypen erweitern.

```
<xsd:element name="preiseuro">
   <xsd:complexType>
      <xsd:simpleContent>
         <xsd:extension base="xsd:decimal">
          <xsd:attribute name="waehrung" type="xsd:string" />
         </xsd:extension>
      </xsd:simpleContent>
   </xsd:complexType>
</xsd:element>
```

In diesem Beispiel wird der einfache Elementtyp, bei dem keinerlei Attribute zulässig sind, als Ausgangsbasis für den komplexen Datentyp verwendet. Innerhalb des Elements `xsd:complexType` ist das Kindelement `xsd:simpleContent` enthalten. Diese Element besitzt wiederum selbst ein Kindelement, und zwar `xsd:extension`. Durch dieses Element wird der einfache Datentyp mit einem Attribut versehen und so zu einem komplexen Datentyp.

4.6.4 Namensräume verwenden

Mit der Einführung von Namensräumen (*http://www.w3.org/TR/REC-xml-names/*) in XML sollten Namenskonflikte vermieden werden. Das gilt insbesondere bei solchen Dokumenttypdefinitionen, die von mehreren Personen erstellt werden. Ebenso wurde durch die Namensräume aber auch nach einer Möglichkeit gesucht, Schemata öffnen und erweitern zu können. Denn in der ersten Version von XML waren Namensräume noch nicht enthalten und wurden der Spezifikation erst später hinzugefügt. Das führt dazu, dass man sie nicht innerhalb einer DTD einsetzen kann. Und eben dies ist ein großer Nachteil der DTDs.

Anders sieht es bei XML Schema aus. Denn hier werden Namensräume unterstützt. Dadurch ist es z.B. möglich, die Gültigkeit von XML-Dokumenten durch Schemata zu überprüfen. Ebenso lassen sich verschiedene Vokabulare mischen, ohne dass es dabei zu Namenskonflikten kommt.

Den Namensraum von XML Schema selbst haben Sie bereits kennengelernt.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

Im gezeigten Beispiel wurde das Präfix `xsd` zugeordnet. Dieses Präfix können Sie selbst wählen. Wichtig ist lediglich, dass es im Schema konsequent verwendet wird. Auf diese Weise werden die im angegebenen Namensraum vorgegebenen Namen für die Elemente des Schemas und die eingebauten Datentypen gekennzeichnet.

Die Definition von Namensräumen in XML Schema unterscheidet sich nur geringfügig von der Definition in Dokumentinstanzen. Man kann einen Standard-Namensraum angeben. Ebenso ist auch der Einsatz beliebig vieler Präfixe möglich.

4.6.4.1 Standard-Namensraum

Neben der zuvor beschriebenen Variante kann man den Namensraum auch zum Standard-Namensraum erklären. In diesem Fall können Sie auf die Angabe des Präfixes verzichten. Erreicht wird dies durch Weglassen des Präfixes bei der Deklaration. Dadurch können Be-

züge auf von XML Schema vorgegebenen Datentypen sowie auf benutzerdefinierte Datentypen nicht mehr direkt unterschieden werden.

Es gibt allerdings Fälle, in denen man auf den Einsatz qualifizierter Namen, die durch ein Präfix mit einem bestimmten Namensraum verbunden sind, verzichten muss. Das ist u.a. der Fall, wenn man ältere XML-Dokumente verwendet, die bislang auf einer DTD aufgebaut haben. Denn Namensräume sind innerhalb von DTDs nicht verfügbar.

4.6.4.2 Unbekannte Elemente einbinden

Durch den Einsatz qualifizierter Namen und Präfixe ist bereits das erste Ziel von Namensräumen, nämlich die Eindeutigkeit von Elementen, erreicht. Allerdings gibt es ein weiteres Ziel, nämlich die Erweiterung. Nun könnte man für ein beliebiges Element die Schema-Definition anpassen und den Namensraum in das Inhaltsmodell aufnehmen. Das Problem dabei: Das Schema wäre in diesem Fall nicht unbedingt für fremde Elemente geöffnet. Stattdessen würde man sich in einem Kreislauf permanenter Anpassung befinden.

Um solche Probleme zu vermeiden, gibt es in der Spezifikation von XML Schema die beiden Elemente `xsd:any` bzw. `xsd:attribute`.

Listing 4.23 Das ist das Ausgangsdokument.

```
<xsd:element name="book">
   <xsd:complexType>
      <xsd:sequence>
         <xsd:element name="isbn" type="xsd:string"/>
         <xsd:element name="informationen">
             <xsd:complexType>
                <xsd:sequence>
                    <xsd:any namespace=
                    http://www.hanser.de/elements/1.1/
                    processContents="skip"/>
                </xsd:sequence>
             </xsd:complexType>
         </xsd:element>
      </xsd:sequence>
   </xsd:complexType>
</xsd:element>
```

`xsd:any` legt fest, dass im Inhaltsmodell des Typs jede Folge von wohlgeformten XML-Elementen verwendet werden kann. Über das `namespace`-Attribut lässt sich steuern, aus welchen Namensräumen Inhalte akzeptiert werden. Im aktuellen Beispiel sind demnach ausschließend Elemente aus dem Namensraum `http://www.hanser.de/elements/1.1/` zulässig.

Bei der folgenden Instanz handelt es sich aus diesem Grund um ein gültiges Element. Denn wie die Syntax zeigt, stammt das `verlag`-Element aus dem geforderten Namensraum.

Listing 4.24 Hier ist alles korrekt, der Namensraum passt also.

```
<book xmlns:dc="http://www.hanser.de/elements/1.1/">
   <isbn>2346175840</isbn>
   <informationen>
      <dc:verlag>
         <dc:name>Michael Mayer</dc:name>
```

```
      </dc:verlag>
    </informationen>

    ...

</book>
```

Neben der expliziten Angabe der Namensräume kann man dem `namespace`-Attribut drei weitere Werte zuweisen.

- `any` – Es handelt sich um wohlgeformtes XML aus einem beliebigen Namensraum. Das ist der Normalfall.
- `local` – Es handelt sich um wohlgeformtes XML, das nicht qualifiziert ist und somit auch keinem Namensraum angehört.
- `other` – Es handelt sich um wohlgeformtes XML, das nicht aus dem Zielnamensraum des gerade definierten Typs stammt.

Über das zweite Attribut `processContents` des `xsd:any`-Elements kann man festlegen, wie ein XML-Prozessor mit fremden Inhalten umgehen soll. Auch diesem Attribut kann man drei Werte zuweisen.

- `lax` – Der XML-Prozessor wird versuchen, die Elemente zu validieren. Werden keine Schema-Informationen gefunden, gibt er allerdings keine Fehler aus.
- `skip` – Der fremde Inhalt wird vom XML-Prozessor ignoriert.
- `strict` – Das ist die Standardeinstellung. Der XML-Prozessor überprüft das mit dem Namensraum verbundene Schema und validiert das Dokument entsprechend.

Um fremde Attribute zu integrieren, wird das Element `anyAttribute` verwendet.

Listing 4.25 Auch fremde Attribute lassen sich einfügen.

```
<xsd:element name="book">
   <xsd:complexType>
      <xsd:sequence>
         <xsd:any namespace="http://www.w3.org/1999/xhtml"
         processContents="skip"
         minOccurs="1" maxOccurs="unbounded"/>
      </xsd:sequence>
      <xsd:anyAttribute namespace="http://www.w3.org/1999/xhtml"
      processContents="lax"/>
   </xsd:complexType>
</xsd:element>
```

Auch dabei sind die beiden Attribute `namespace` und `processContents` verfügbar, die die gleichen Werte wie zuvor beschrieben besitzen können. Innerhalb des `xsd:any`-Elements kann man zudem über `minOccurs` und `maxOccurs` die Anzahl der fremden Elemente angeben, die eingefügt werden dürfen. Bei Attributen ist das hingegen nicht möglich.

4.6.4.3 Den Zielnamensraum angeben

Wird ein Schema für ein aktuelles XML-Dokument entworfen, sollte man in aller Regel auf die Möglichkeiten zurückgreifen, die Namensräume bieten. Hier kommt der sogenannte Zielnamensraum ins Spiel. Gebildet wird der Namensraum, indem man ihn über eine

Deklaration identifiziert. Für diesen Zweck können dem Element `xsd:schema` die entsprechenden Attribute zugewiesen werden.

Listing 4.26 Der Zielnamensraum wird angegeben.

```
<xsd:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
   xmlns:hv="http://www.hanser.de/a"
   targetNamespace="http://www.hanser.de/a"
   elementFormDefault="qualified"
   attributeFormDefault="unqualified">
```

Über das Attribut `targetNamespace` wird in diesem Beispiel der Namensraum definiert, durch den das Schema selbst beschrieben wird. Mit der Namensraumdeklaration `xmlns:hanser` wird das gewünschte Präfix zugewiesen. Dieses Präfix verwendet man für die Kennzeichnung der Komponenten. Diese Komponenten sind

- Attribute,
- Elemente und
- Datentypen.

Verwendet wird das Präfix, um Bezüge innerhalb des Schemas herstellen zu können. In diesem Fall muss man qualifizierte Namen, die sogenannten QNames, einsetzen. Das Präfix muss also angegeben werden, außer der Namensraum wird als Standard-Namensraum definiert.

4.6.5 Mit lokalen Elementen und Attributen arbeiten

Auf den folgenden Seiten geht es um den Umgang mit lokalen Elementen und Attributen. Zentrale Attribute sind dabei `elementFormDefault` und `attributeFormDefault`. Über diese kann man global festlegen, dass bei der Validierung eines dem Schema zugeordneten XML-Dokuments die Namen der lokalen Elemente und Attribute auf Übereinstimmung mit dem angegebenen Namensraum validiert werden.

Die Qualifizierung lokaler Elemente und Attribute kann global über die beiden folgenden Attribute des Elements `schema` bestimmt werden:

- `elementFormDefault`
- `attributeFormDefault`

Ebenso kann die Qualifizierung für jede einzelne Deklaration über das `form`-Attribut angegeben werden. Jedes der genannten Attribute kann einen der beiden Werte

- `qualified` oder
- `unqualified`

annehmen, um zu bestimmen, ob lokal definierte Elemente oder Attribute qualifiziert werden müssen oder nicht.

4.6.5.1 Qualifizierte Elemente einsetzen

Für Elemente und Attribute kann unabhängig voneinander verlangt werden, dass sie qualifiziert angegeben werden müssen. Sollen alle lokalen Elemente qualifiziert werden, setzt man den Wert von `elementFormDefault` auf `qualified`.

Listing 4.27 So werden Elemente qualifiziert.

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
   xmlns:best="http://www.hanser.de/bestellt"
   targetNamespace="http://www.hanser.de/bestellt"
   elementFormDefault="qualified"
   attributeFormDefault="unqualified">

<element name="bestellung" type="best:BestellungTyp" />
<element name="hinweis" type="string"/>

<complexType name="BestellungTyp">
   <!-- etc. -->
</complexType>
<!-- etc. -->

</schema>
```

In der folgenden konformen Dokumentinstanz werden die Elemente explizit qualifiziert:

Listing 4.28 Die Elemente werden qualifiziert.

```
<?xml version="1.0"?>
<best:bestellung xmlns:best="http://www.hanser.de/bestellt"
      datum="2010-02-11">

   <best:lieferadresse land="de">
      <best:name>Michael Mayer</best:name>
      <best:straße>Elmstreet</best:straße>
      <best:plz>10435</best:plz>
      <best:ort>Berlin</best:ort>
   </best:lieferadresse>

   <best:rechnungsadresse land="de">
      <best:name>Fred Kruger</best:name>
      <best:straße>Oxfordstreet</best:straße>
       <best:plz>22523</best:plz>
      <best:ort>Hamburg</best:ort>
   </best:rechnungsadresse>

   <best:rechnungsadresse land="de">
      <best:name>Hannes Bundy</best:name>
      <best:straße>Lakeside</best:straße>
      <best:plz>10345</best:plz>
      <best:ort>Berlin</best:ort>
   </best:rechnungsadresse>

   <best:hinweis>Vorsicht Glas!</best:hinweis>
  <!-- etc. -->

</best:bestellung>
```

In diesem Beispiel wurde jedes Element explizit qualifiziert. Eine Alternative dazu ist die Qualifizierung anhand eines voreingestellten Namensraums. Das sähe dann folgendermaßen aus:

Listing 4.29 Hier wird ein Namensraum verwendet.

```
<?xml version="1.0"?>
<bestellung xmlns:best="http://www.hanser.de/bestellt"
      datum="2010-02-01">


<lieferadresse land="de">
   <name>Michael Mayer</name>
   <straße>Elmstreet</straße>
   <plz>10435</plz>
   <ort>Berlin</ort>
</lieferadresse>

 <rechnungsadresse land="de">
    <name>Fred Kruger</name>
    <straße>Oxfordstreet</straße>
    <plz>22523</plz>
    <ort>Hamburg</ort>
 </rechnungsadresse>

 <rechnungsadresse land="de">
   <name>Hannes Bundy</name>
   <straße>Lakeside</straße>
   <plz>10345</plz>
   <ort>Berlin</ort>
</rechnungsadresse>

<hinweis>Vorsicht Glas!</hinweis>
 <!-- etc. -->

</bestellung>
```

In diesem Beispiel gehören alle Elemente zum gleichen Namensraum. Der voreingestellte Namensraum wird über das `xmlns`-Attribut deklariert. Dadurch gehört dieser Namensraum zu allen Elementen dieses Dokuments. Vor die Elemente muss nun kein Präfix mehr gesetzt werden.

4.6.5.2 Qualifizierte Attribute

Die Qualifizierung von Attributen funktioniert ähnlich wie bei den Elementen. Wird dem `attributeFormDefault`-Attribut der Wert `qualified` zugewiesen oder ein Attribut wurde global deklariert, muss es in Dokumentinstanzen mit einem Präfix ausgestattet werden. Im Gegensatz zu Elementen müssen Attribute, die qualifiziert werden sollen, ein Präfix besitzen. Denn XML Namespace sieht keinen voreingestellten Namensraum für Attribute vor. Attribute, die nicht qualifiziert werden müssen, werden im Dokument ohne Präfix angegeben. Das ist der Normalfall.

Bislang wurde gezeigt, wie sich alle Elemente und Attribute innerhalb eines bestimmten Zielnamensraums qualifizieren lassen. Ebenso ist aber auch die Qualifizierung einer einzelnen Deklaration möglich. Verwendet wird dafür das `form`-Attribut.

Im folgenden Beispiel soll erreicht werden, dass ein lokal deklariertes Attribut innerhalb von Dokumenten qualifiziert werden muss. Das Attribut lautet in diesem Beispiel `publicKey`.

Listing 4.30 Das Schema wurde angelegt …

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
   xmlns:best="http://www.hanser.de/bestellung"
   targetNamespace="http://www.hanser.de/bestellung"
   elementFormDefault="qualified"
   attributeFormDefault="unqualified">

<!-- etc. -->

   <element name="Sicherheit">
      <complexType>
         <sequence>
            <!-- Elementdeklarationen -->
         </sequence>
         <attribute name="publicKey"
         type="base64Binary" form="qualified"/>
      </complexType>
   </element>

</schema>
```

Diesem Schema entspricht dann das folgende Dokument:

Listing 4.31 ... und entspricht diesem Dokument.

```
<?xml version="1.0"?>
<bestellung xmlns="http://www.hanser.de/bestellung"
   xmlns:best= "http://www.hanser.de/bestellung"
   bestellt="2010-02-01">

   <!-- etc. -->

   <sicherheit best:publicKey="GpM7">
      <!-- etc. -->
   </sicherheit>

<bestellung>
```

Das bereits vorgestellte `form`-Attribut lässt sich auch in Verbindung mit Elementen anwenden. Ein Dokument, das diesem Schema entspricht, könnte dann folgendermaßen aussehen:

Listing 4.32 So könnte das Dokument aussehen.

```
<?xml version="1.0"?>
<bestellung xmlns="http://www.hanser.de/bestellung"
   xmlns:best="http://www.hanser.de/bestellung"
   bestellt="2010-02-01">

   <!-- etc. -->
   <sicherheit best:publicKey="GpM7">
      <!-- etc. -->

   </sicherheit>

<bestellung>
```

4.6.6 Globale Elemente und Attribute

Bei den globalen Elementen handelt es sich um direkte Nachkommen des Elements `xsd:schema`. Globale Elemente unterliegen zwei Einschränkungen:

- Sie dürfen kein `ref`-Attribut besitzen.

- Angaben über die Häufigkeit des Vorkommens mittels `minOccurs` und `maxOccurs` sind nicht erlaubt.

Ebenso wie Elemente lassen sich auch Attribute global deklarieren, wenn sie als direkte Kinder von xsd:`schema` eingefügt werden. In diesem Fall können sie in untergeordneten Elementen mit dem Attribut `ref` verwendet werden. Das `use`-Attribut darf innerhalb einer globalen Attributdeklaration jedoch nicht eingesetzt werden.

Die Unterscheidung zwischen lokalen Elementen und Attributen spielt noch in einem anderen Zusammenhang eine wichtige Rolle. Einerseits muss zwar jedes globale Element einen eindeutigen Namen besitzen, ebenso kann man aber auch im Kontext unterschiedlicher komplexer Elementtypen Kindelemente oder Attribute verwenden, die über den gleichen Namen angesprochen werden.

Auch wenn diese Elemente oder Attribute den gleichen Namen besitzen, sind sie nicht identisch. Sie unterscheiden sich durch den entsprechenden Kontext. In diesem Zusammenhang ist wichtig zu verstehen, dass jeder komplexe Datentyp einen eigenen Symbolraum eröffnet.

Ebenso sind die Symbolräume für Typdefinitionen sowie Element- und Attributnamen innerhalb eines gemeinsamen Namensraums voneinander getrennt. So ist zum Beispiel auch Folgendes möglich:

Listing 4.33 So etwas ist auch erlaubt.

```
<xsd:element name="vname" type="vname" />
<xsd:complexType name="vname">
   <xsd:attribute name="vname" type="xsd:string" />
</xsd:complexType>
```

Ob solche Namensgleichheiten sich nicht besser auflösen lassen, ist eine ganz andere Frage.

4.7 Häufigkeitsbestimmungen

Es gibt Fälle, da genügt es für die Beschreibung eines Inhaltsmodells nicht, dass man nur aufzählt, welche Elemente in welcher Reihenfolge vorkommen sollen. Vielmehr muss man angeben, ob ein Element oder Attribut vorkommen muss, wie oft es vorkommen soll oder ob es auch weggelassen werden kann. Interessant sind in diesem Zusammenhang hauptsächlich die beiden Attribute `minOccurs` und `maxOccurs`. Während `maxOccurs` für das maximale Vorkommen eines Elements verantwortlich ist, beschreibt `minOccurs` das Mindestvorkommen. Diese Attribute dürfen in globalen Elementen nicht verwendet werden.

Wird bei `minOccurs` der Wert `0` angegeben, ist das Attribut optional. Ein Element muss vorhanden sein, wenn der Wert von `minOccurs` `1` oder größer ist. Die größtmögliche Anzahl, in der ein Element auftreten darf, wird mit `maxOccurs` beschrieben. Bei dessen Wert kann es sich um eine positive Ganzzahl oder den Begriff `unbounded` handeln. Durch `unbounded` wird festgelegt, dass es keine Obergrenze gibt.

Der Standardwert für beide Attribute ist jeweils `1`. Wird also ein Element ohne die Angabe eines `maxOccurs`-Attributs deklariert, darf dieses Element nicht öfter als einmal vorkommen.

Gibt man für `minOccurs` einen Wert an, muss man darauf achten, dass er kleiner oder gleich dem Vorgabewert des `maxOccurs`-Attributs ist, also `0` oder `1`. Wird nur ein Wert für das `maxOccurs`-Attribut angegeben, muss dieser größer oder gleich dem Vorgabewert für `minOccurs` sein, also 1 oder größer. Gibt man keines der beiden Attribute an, muss das Element genau einmal vorkommen.

Attribute können einmal oder keinmal auftreten, in anderen Häufigkeiten jedoch nicht. Demzufolge weicht die Syntax für die Häufigkeitsbeschränkungen für Attribute auch von der von Elementen ab. So lassen sich Attribute beispielsweise mit einem `use`-Attribut deklarieren. Auf diese Weise kann man festlegen, ob ein Attribut vorgeschrieben (`required`), optional (`optional`) oder verboten (`prohibited`) ist.

Vorgabewerte für Attribute und Elemente werden über das `default`-Attribut angegeben. Die Anwendung dieses Attributs auf Elemente und Attribute unterscheidet sich etwas. Wird ein Attribut mit einem Vorgabewert deklariert, entspricht sein Wert dem in der Dokumentinstanz angegeben. Taucht das Attribut in der Dokumentinstanz nicht auf, fügt der Schema-Prozessor als Wert den Wert des `default`-Attributs ein.

Standardwerte für ein Attribut sind nur sinnvoll, wenn das Attribut selbst optional ist. Daher ist es falsch, wenn man einen Default-Wert spezifiziert, wenn das `use`-Attribut einen anderen Wert als `optional` besitzt.

Wie bereits erwähnt, sieht das Ganze bei Elementen, die einen Default-Wert haben, etwas anders aus. In diesem Fall ist der Wert des Elements nämlich der in der Dokumentinstanz als sein Inhalt angegebene Wert. Taucht das Element in der Dokumentinstanz ohne Inhalt auf, fügt der Schema-Prozessor den Wert des `default`-Attributs ein. Tritt das Element hingegen überhaupt nicht auf, fügt der Schema-Prozessor auch nichts ein.

Um sicherzustellen, dass ein Element oder Attribut einen bestimmten Wert hat, wird innerhalb der Deklaration von Attributen und Elementen das `fixed`-Attribut verwendet. Eine Deklaration darf nicht gleichzeitig ein `fixed`- und ein `use`-Attribut haben.

4.7.1.1 Beispiele zum besseren Verständnis

Um das zuvor Beschriebene anschaulicher zu machen, folgen in diesem Abschnitt einige Beispiele. Zunächst ein Beispiel dafür, wie man innerhalb eines Bestellformulars angeben kann, dass gar keine oder unendlich viele Bestellungen möglich sind.

```
<xsd:element name="bestellung" maxOccurs="unbounded"
minOccurs="0" />
```

Will man hingegen die Angabe einer E-Mail-Adresse erzwingen, sieht das folgendermaßen aus:

```
<xsd:element name="email" minOccurs="1" />
```

Soll die Telefonnummer zwar weggelassen werden können, gleichzeitig aber auch bis zu zwei angegebene Nummern gültig sein, sieht die Syntax folgendermaßen aus:

```
<xsd:element name="telefon" maxOccurs="2" minOccurs="0" />
```

Die Werte der vorgestellten Attribute, die innerhalb von Element- und Attribut-Deklarationen verwendet werden, sind noch einmal in Tabelle 4.16 aufgeführt.

Tabelle 4.16: Mögliche Werte der Attribute

Elemente (minOccurs, maxOccurs) fixed, default	Attribute use, fixed, default	Anmerkungen
(1, 1) -, -	required, -, -	Das Element oder Attribut muss exakt einmal vorkommen. Der Wert ist beliebig.
(1, 1) 40, -	required, 40,	Das Element oder Attribut muss exakt einmal vorkommen. Der Wert muss explizit 40 sein.
(2, unbounded) 40, -	lässt sich nicht anwenden	Das Element oder Attribut muss zweimal oder häufiger vorkommen. Der Wert muss 40 sein. Die Werte von `maxOccurs` und `minOccurs` müssen positive Ganzzahlen sein, wobei der Wert von `maxOccurs` auch `unbounded` sein darf.
(0, 1) -, -	optional, -, -	Das Element oder Attribut darf maximal einmal vorkommen. Sein Wert ist beliebig.
0, 1) 40, -	optional, 40, -	Das Element oder Attribut darf maximal einmal vorkommen. Wenn es auftaucht, muss der Wert `40` sein, ansonsten wird `40` angenommen.
(0, 1) -, 40	optional, -, 40	Das Element oder Attribut darf maximal einmal vorkommen. Wenn es leer ist (Element) oder nicht auftaucht (Attribut), ist der Wert `40`, ansonsten wird der angegebene Wert angenommen.
(0, 2) -, 40	nicht anwendbar	Das Element darf maximal zweimal vorkommen. Tritt es nicht auf, wird auch nichts eingefügt. Sollte es auftauchen, aber leer sein, dann ist sein Wert 40, ansonsten der angegebene Wert. Ganz allgemein dürfen die Werte für `minOccurs` und `maxOccurs` positive Ganzzahlen sein. Zusätzlich kann der Wert von maxOccurs unendlich (*unbounded*) sein.
(0, 0) -, -	prohibited, -, -	Das Element oder Attribut darf nicht vorkommen.

4.8 Kompositoren einsetzen

Es stehen verschiedene Möglichkeiten der Elementgruppierung innerhalb von Inhaltsmodellen zur Verfügung. Kennengelernt haben Sie davon bislang allerdings lediglich die Variante, bei der für komplexe Elementtypen die Informationselemente als Sequenz eingeführt wurden. Dabei musste immer eine bestimmte Reihenfolge eingehalten werden. Was noch alles möglich ist, zeigen die folgenden Abschnitte.

4.8.1 xsd:all

`xsd:all` ist immer dann interessant, wenn aus einer Anzahl von Elementen jedes Element exakt einmal vorkommen darf. So spielt es im folgenden Beispiel keine Rolle, welche Elemente vorhanden sein müssen. Die Elementreihenfolge ist ebenfalls egal.

Listing 4.34 Die Reihenfolge spielt keine Rolle.

```
<xsd:complexType name="einkaufen">
   <xsd:all>
      <xsd:element name="Butter" type="xsd:string"/>
      <xsd:element name="Käse" type="xsd:string"/>
      <xsd:element name="Smothie" type="xsd:string"/>
      <xsd:element name="Wasser" type="xsd:string"/>
   </xsd:all>
</xsd:complexType>
```

Hier handelt es sich offensichtlich um eine Einkaufsliste. Es wurde alles aufgeschrieben, selbst wenn man nicht weiß, ob man alles im Geschäft bekommt. Ebenso wenig spielt es eine Rolle, ob man nun die Butter oder das Wasser zuerst in den Warenkorb legt.

Aufgepasst werden muss allerdings, da `xsd:all`-Modellgruppen bestimmten Einschränkungen unterliegen. So müssen sie zum Beispiel jeweils als einziges Kind eines Elements verwendet werden, sie lassen sich also nicht mit anderen Kindelementen mischen. Folgendes wäre demnach nicht möglich:

Listing 4.35 Das wird nicht funktionieren.

```
   <xsd:sequence>


   <xsd:all>
      <xsd:element ... />
      <xsd:element ... />
      <xsd:element ... />
   </xsd:all>
   <xsd:sequence>


         <xsd:element ... />
         <xsd:element ... />


   <xsd:element ... />
</xsd:sequence>
<xsd:choice>
<xsd:element ... />
   <xsd:element ... />
```

```
   <xsd:element ... />
</xsd:choice>
```

Zudem dürfen die Elemente innerhalb der Modellgruppe maximal einmal auftauchen.

4.8.2 xsd:choice

Mit `xsd:choice` können Sie Alternativen anbieten, aus denen eine gewählt werden kann. Aus den vorgegebenen Elementen darf dann, falls kein anderer Wert über `maxOccurs/minOccurs` gesetzt wurde, ein Element vorkommen. Ansonsten sind aber auch mehrere möglich. Die Reihenfolge der Elemente spielt dabei keine Rolle.

Listing 4.36 `xsd:choice` in Aktion

```
<xsd:complexType name="waehrung">

   <xsd:choice>
      <xsd:element name="dollar" type="xsd:string"/>
      <xsd:element name="euro" type="xsd:string"/>
      <xsd:element name="rubel" type="xsd:string"/>
      <xsd:element name="yen" type="xsd:string"/>
   </xsd:choice>

</xsd:complexType>
```

4.8.3 xsd:sequence

Durch eine Sequenz können Sie die Reihenfolge von Elementen erzwingen. Auch hierzu wieder ein Beispiel:

Listing 4.37 Die Reihenfolge wird erzwungen.

```
<xsd:complexType name="anschrift">

   <xsd:sequence>
      <xsd:element name="wohnort" type="xsd:string"/>
      <xsd:element name="plz" type="xsd:int"/>
      <xsd:element name="strasse" type="xsd:string"/>
      <xsd:element name="nr" type="xsd:int"/>
         <xsd:element name="name" type="xsd:string"/>
   </xsd:sequence>

</xsd:complexType>
```

Es wird angenommen, dass hier verschiedene Elemente für eine Anschrift definiert wurden. Diese Elemente müssen exakt in der angegebenen Reihe gesetzt werden.

4.8.4 Modellgruppen verschachteln

Neben dem Einsatz der drei genannten Gruppen gibt es noch eine weitere Möglichkeit: Modellgruppen lassen sich ineinander verschachteln. Auch hierzu wieder ein kleines Beispiel:

Listing 4.38 Eine verschachtelte Modellgruppe

```
<xsd:element name="bestellung">

<xsd:complexType>
   <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="vorname" type="xsd:string"/>
      <xsd:element name="anschrift" type="xsd:string"/>
      <xsd:element name="kontakt" type="xsd:string"/>

         <xsd:complexType>
            <xsd:all>
               <xsd:element name="telefon" type="xsd:string"/>
               <xsd:element name="email" type="xsd:string"/>
               <xsd:element name="fax" type="xsd:string"/>
            </xsd:all>
         </xsd:complexType>
      </xsd:sequence>
   </xsd:complexType>
</xsd:element>
```

In diesem Beispiel wurde innerhalb einer `xsd:sequence`-Gruppe eine `xsd:all`-Gruppe definiert. Dadurch kann aus verschiedenen Kontaktmöglichkeiten die gewünschte ausgewählt werden.

4.9 Mit benannten Modellgruppen arbeiten

Innerhalb komplexer Elementtypen sind normalerweise Gruppierungen mehrerer Elemente enthalten. Das kann beispielsweise eine Sequenz oder eine Auswahlmöglichkeit sein. Wenn man eine Modellgruppe mehrfach benötigt, bietet sich der Einsatz von `xsd:group` an. Über dieses Element kann man Gruppen zusammenfassen und ihnen einen Namen zuordnen, der Teil des verwendeten Zielnamensraums ist. Anhand von Name und Namensraum muss die Modellgruppe dann innerhalb des Schemas eindeutig identifizierbar sein.

Diese Variante ermöglicht es, die Modellgruppe in mehreren komplexen Datentypen zu verwenden. Ebenso können aber auch benannte komplexe Datentypen in unterschiedlichen Bereichen eines Schemas genutzt werden. Bei einer Modellgruppe handelt es sich also um eine Gruppe innerhalb der Ebene komplexer Elementtypen. Im Gegensatz zu denen können benannte Modellgruppen ihre Eigenschaften nicht vererben. Aus einer Modellgruppe heraus können somit keine neuen Modellgruppen durch Erweiterung oder Einschränkung gebildet werden.

Modellgruppen können Sie für eine Liste bestimmter Zusammenstellungen definieren. Verwendet werden dafür die bereits beschriebenen Kompositoren

- `xsd:all`,
- `xsd:sequence` oder
- `xsd:choice`.

Ebenso lassen sich auch Annotationen einfügen, um auf diese Weise die Gruppe zu kommentieren. Das Verschachteln von Gruppen in sich selbst ist nicht erlaubt. Eine Gruppe `kontakt` kann in sich selbst nicht die Gruppe `kontakt` enthalten.

Im folgenden Beispiel werden Informationen über Online-Kontaktmöglichkeiten in der Gruppe `web` gespeichert. Diese Informationen sollen sowohl bei den privaten wie auch bei den geschäftlichen Kontakten verfügbar sein.

Zunächst die eigentliche Gruppe:

Listing 4.39 So sieht die Gruppe aus.

```
<xsd:group name="web">
   <xsd:sequence>
      <xsd:element name="email" type="xsd:string" minOccurs="0"/>
      <xsd:element name="url" type="xsd:string" minOccurs="0"/>
   </xsd:sequence>
</xsd:group>
```

Nun muss innerhalb der anschließend definierten komplexen Typen die Gruppe jeweils eingefügt werden. Verwendet wird dafür das Attribut `ref`.

Listing 4.40 Und so wird die Gruppe eingefügt.

```
<xsd:complexType name="privat">
   <xsd:sequence>
      <xsd:element name="anschrift" type"xsd:string"/>
      <xsd:group ref="web"/>
   </xsd:sequence>
   <xsd:attribute name="beurteilung" type="xsd:string"/>
</xsd:complexType>

<xsd:complexType name="business">
   <xsd:sequence>
      <xsd:element name="anschrift" type"xsd:string"/>
      <xsd:group ref="web"/>
   </xsd:sequence>
   <xsd:attribute name="beurteilung" type="xsd:string"/>
</xsd:complexType>
```

Bei diesen Gruppen handelt es nicht um vollständige Datentypen. Vielmehr sind es Container für Elemente und Attribute, die für die Beschreibung komplexer Typen verwendet werden.

4.9.1 Attributgruppen definieren

Wie sich Elemente zu Gruppen zusammenfassen lassen, haben Sie bereits gesehen. Gleiches gilt für Attribute. Auch diese kann man zu Gruppen zusammenschließen und sie über einen Namen ansprechen. Angenommen, es wurde das Element `buch` definiert, dem mittels Attributen zahlreiche Eigenschaften zugeordnet werden sollen.

Listing 4.41 So werden Attributgruppen definiert.

```
<xsd:complexType name="buch">
<xsd:sequence>
...

</xsd:sequence>
   <xsd:attribute name="ISBN" type="ISBNTyp" use="required"/>
   <xsd:attribute name="Autor" type="xsd:string"/>
   <xsd:attribute name="Auflage">
```

```
        <xsd:simpleType>
          <xsd:restriction base="xsd:short">
            <xsd:minInclusive value="1"/>
            <xsd:maxInclusive value="40"/>
          </xsd:restriction>

        </xsd:simpleType>
      </xsd:attribute>
  </xsd:complexType>
```

Eine solche Definition ist – und das zeigt die vorherige Syntax eindrucksvoll – recht aufwendig. Benötigt man diese in einem Dokument mehrmals, muss man sie aber nicht immer aufs Neue definieren, sondern kann zu einer Attributgruppe greifen. In der Praxis sähe das dann folgendermaßen aus:

Listing 4.42 So wird die Attributgruppe definiert.

```
<xsd:attributeGroup ref="buch"/>
...
<xsd:attributeGroup name="buch">
    <xsd:attribute name="ISBN" type="ISBNTyp" use="required"/>
    <xsd:attribute name="Autor" type="xsd:string"/>
    <xsd:attribute name="Auflage">
      <xsd:simpleType>
        <xsd:restriction base="xsd:short">
          <xsd:minInclusive value="1"/>
          <xsd:maxInclusive value="40"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:attribute>
</xsd:attributeGroup>
```

Durch den Einsatz von Attributgruppen lässt sich die Lesbarkeit eines Schemas verbessern. Positiv kommt außerdem die leichtere Wartbarkeit des Schemas hinzu. Denn schließlich wird die Attributgruppe nur an einer Stelle definiert. Werden Änderungen nötig, müssen diese nur an einer Stelle vorgenommen werden.

4.10 Schlüsselelemente und deren Bezüge

Wenn Sie bereits Erfahrung mit Datenbanken gemacht haben, wird Ihnen das Prinzip der Schlüsselfelder sicherlich bekannt sein. Über solche Felder kann gezielt auf bestimmte Datensätze zugegriffen werden. Gleichzeitig verhindern Schlüsselfelder, dass Datensätze doppelt vorkommen.

Insgesamt gibt es in XML Schema die folgenden fünf Arten von Schlüsselfeldern:

- `xsd:key`
- `xsd:field`
- `xsd:unique`
- `xsd:refkey`
- `xsd:selector`

Was es mit denen im Einzelnen auf sich hat, wird auf den folgenden Seiten gezeigt.

4.10.1 Die Eindeutigkeit von Elementen

Die in XML Schema verfügbaren bzw. definierbaren eindeutigen Schlüssel sind mit den Primärschlüsseln aus relationalen Datenbanken vergleichbar. Ein solcher Primärschlüssel wird dazu verwendet, die Tupel einer Relation eindeutig identifizieren zu können.

XML Schema unterscheidet hierbei zwischen der Eindeutigkeit und der Schlüsseleigenschaft. Verwendet werden für die Definition einer solchen Eindeutigkeit drei Elemente:

- `xsd:key`
- `xsd:unique`
- `xsd:keyref`

Dabei wird auf eine eingeschränkte Menge von XPath-Ausdrücken zurückgegriffen. Ausführliche Informationen zu dieser Abfragesprache finden Sie in Kapitel 5.

Im folgenden Beispiel wird das Element `buchliste` definiert, das eine Liste verschiedener Elemente enthält.

Listing 4.43 In der Liste sind verschiedene Elemente enthalten.

```
<xsd:element name="Name">

   <xsd:complexType>
      <xsd:sequence>
         <xsd:element name="Vorname" type="xsd:string"/>
         <xsd:element name="Nachname" type="xsd:string"/>
         </xsd:sequence>
      <xsd:attribute name="Nummer" type="xsd:integer"/>
   </xsd:complexType>

</xsd:element>

<xsd:key name="Schlüssel">
   <xsd:selector xpath="Name" />
   <xsd:field xpath="@Nummer" />
</xsd:key>
```

Interessant ist in diesem Beispiel hauptsächlich das `xsd:key`-Element, das im aktuellen Beispiel für einen eindeutigen Autorennamen sorgen soll. Wie das Beispiel zeigt, schließt sich `xsd:key` direkt an die Definition des komplexen Datentyps `buchliste` an. Durch die Platzierung wird gleichzeitig die Reichweite der Einschränkung festgelegt, die innerhalb des `xsd:key`-Elements definiert wird.

Über `xsd:selector` wird das Element angegeben, für das die Einschränkung gelten soll. Im aktuellen Beispiel handelt es sich dabei um `Name`. Als Attributwert wird ein entsprechender XPath-Ausdruck verwendet, durch den das Element `Name` ausgewählt wird. Über

```
<field>
```

gibt man den Teil des Elements an, für das man die Eindeutigkeit definieren will. Dabei kann es sich um ein Kind-, ein Kindeskind-Element oder ein Attribut des Elements handeln.

4.10.1.1 Mit xsd:unique arbeiten

Parallel zu `xsd:key` kann man mit `xsd:unique` arbeiten. Auch hierzu wieder ein Beispiel:

Listing 4.44 Auch so etwas ist möglich.

```
<xsd:unique name="titel">
   <xsd:selector xpath="autor"/>
   <xsd:field xpath="name"/>
   <xsd:field xpath="verlag"/>
</xsd:unique>

<xsd:key name="idKey">
   <xsd:selector xpath="autor"/>
   <xsd:field xpath="@id"/>
</xsd:key>
```

Bei `xsd:unique` sind auch Elemente mit Nullwerten möglich.

4.10.2 Auf Schlüsselelemente Bezug nehmen

Nachdem man mittels `xsd:key` oder `xsd:unique` die Felder definiert hat, kann man auf diese über `xsd:keyref` Bezug nehmen.

Listing 4.45 Der Schlüssel wurde definiert.

```
<xsd:element name="Name">
   <xsd:complexType>
      <xsd:sequence>
         <xsd:element name="Vorname" type="xsd:string"/>
         <xsd:element name="Nachname" type="xsd:string"/>
         </xsd:sequence>
      <xsd:attribute name="Nummer" type="xsd:integer"/>
   </xsd:complexType>
</xsd:element>

<xsd:key name="Schlüssel">
   <xsd:selector xpath="Name" />
   <xsd:field xpath="@Nummer" />
</xsd:key>

<xsd:element name="King">
   <xsd:keyref refer="Schlüssel"/>
      <xsd:selector xpath="Autor/Vorname"/>
      <xsd:field xpath="autorid"/>
   </xsd:keyref>
</xsd:element>
```

Das `xsd:key`-Element definiert den eigentlichen Schlüssel. Das dazu passende XML-Dokument könnte nun folgendermaßen aussehen:

Listing 4.46 So sieht das XML-Dokument aus.

```
<Name Nummer="1">
   <Vorname>Fred</Vorname>
   <Nachname>Kruger</Nachname>
</Name>
```

Gültig wäre nun Folgendes:

```
<King>1</King>
```

Ungültig hingegen dieses hier:

```
<King>12</King>
```

4.11 Komplexe Datentypen ableiten

Die Arbeit an und mit einfachen Datentypen haben Sie kennengelernt. Aber auch die komplexen Datentypen bieten die Möglichkeit, aus einer existierenden Struktur andere Strukturen abzuleiten. Ebenso wie bei den einfachen wird auch bei den komplexen Datentypen mit Einschränkungen oder Erweiterungen gearbeitet.

4.11.1 Komplexe Elemente erweitern

Durch den Einsatz des Elements `xsd:complexType` lassen sich komplexe Elementtypen erweitern. Das folgende Beispiel zeigt, wie das funktioniert:

Listing 4.47 Das ist der Ausgangszustand.

```
<xsd:complexType name="adresse">


<xsd:sequence>
   <xsd:element name="name" type="xsd:string"/>
   <xsd:element name="vorname" type="xsd:string"/>
   <xsd:element name="strasse" type="xsd:string"/>
   <xsd:element name="ort" type="xsd:string"/>
</xsd:sequence>


</xsd:complexType>
```

Innerhalb dieses Beispiels existiert ein komplexer Elementtyp `adresse`. Durch das Anhängen weiterer Elemente oder das Hinzufügen zusätzlicher Attribute soll davon ein Elementtyp abgeleitet werden.

Listing 4.48 Ein Elementtyp wird abgeleitet.

```
<xsd:complexType name="weiterfuehrend">

   <xsd:complexContent>
      <xsd:extension base="adresse">

         <xsd:sequence>
            <xsd:element name="email" type="xsd:string"/>
            <xsd:element name="telefon" type="xsd:string"/>
         </xsd:sequence>
      </xsd:extension>
   </xsd:complexContent>
 </xsd:complexType>
```

In einem solchen Fall, also wenn ein Inhaltsmodell von einem anderen abgeleitet wird, werden beide Modelle nacheinander ausgewertet. Für das obige Beispiel bedeutet dies also, dass die innerhalb `des xsd:extension`-Elements angegebenen Elemente an die Sequenz der Elemente des Basistyps angehängt werden.

Damit innerhalb einer Dokumentinstanz ein abgeleiteter Typ verwendet werden kann, muss man ihn explizit angeben. Verwendet wird dafür das `type`-Attribut, das ein Teil des Namensraums XML-Schema-instance ist. Dieser Namensraum muss unbedingt angegeben werden.

Listing 4.49 Hier wurde der Namensraum korrekt angegeben.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<adressbank xmlns:xsi="http://www.w3.org/2001/XML-Schema-instance">
   [...]
      <adresse xsi:type="weiterfuehrend">
   [...]
</adresssbank>
```

Und der Vollständigkeit halber hier noch der XML-Code des Elements vom Typ `weiterfuehrend`.

Listing 4.50 So sieht das Element aus.

```
<adresse xsi:type="weiterfuehrend">
   <name/>
   <vorname/>
   <strasse/>
   <ort/>
   <email/>
   <telefon/>
</adresse>
```

4.11.2 Komplexe Elemente einschränken

Nachdem man ein komplexes Datenelement definiert hat, kann man davon an anderen Stellen eingeschränkte Varianten nutzen. Vom Prinzip her handelt es sich bei der Einschränkung um das Gleiche wie bei der Erweiterung. Allerdings wird bei komplexen Typen eine Typdeklaration benötigt. Nur die Angabe des zulässigen Wertebereichs genügt hier nicht. Ein durch Einschränkung abgeleiteter Typ ist dem Basistyp sehr ähnlich. Allerdings ist die Deklaration eingeschränkter als die Deklaration des Basistyps. Die durch den neuen Typ repräsentierte Wertemenge ist eine Untermenge der durch den Basistyp beschriebenen Werte.

Listing 4.51 Die Einschränkung komplexer Typen

```
<xsd:complexType name="Bestellung">
  <xsd:complexContent>
   <xsd:restriction base="bestellt:Art">

  <xsd:sequence>
     <xsd:element name="Buch" minOccurs="1" maxOccurs="unbounded">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="Titel" type="string"/>
          <xsd:element name="Anzahl">
            <xsd:simpleType>
              <xsd:restriction base="positiveInteger">
                 <xsd:maxExclusive value="100"/>
            </xsd:restriction>
          </xsd:simpleType>
          </xsd:element>
```

```
            <xsd:element name="PreisEUR" type="decimal"/>
            <xsd:element ref="bestellt:Kommentar" minOccurs="0"/>
            <xsd:element name="Datum" type="date" minOccurs="0"/>
          </xsd:sequence>
          <xsd:attribute name="ISBN"
                type="bestellt:isbn" use="required"/>
        </xsd:complexType>
      </xsd:element>
      </xsd:sequence>
     </xsd:restriction>
   </complexContent>
</complexType>
```

In diesem Beispiel wurde der neue Typ `Bestellung` erstellt.

```
<xsd:restriction base="bestellt:Art">
```

Durch die beiden Attribute `minOccurs` und `maxOccurs` kann außerdem noch die Häufigkeit der Elemente geändert werden. Bei dem abgeleiteten Typ muss man alle Elemente des Basistyps noch einmal angegeben.

4.11.3 Datentypen steuern und ableiten

Je nach Umfang des Schemas kann es sinnvoll sein, die Optionen für die Ableitung bestimmter Datentypen selbst einzuschränken. Auf diese Weise lassen sich sinnfreie Inhaltsmodelle verhindern.

- Für komplexe Datentypen kann eine Ableitung
- durch Einschränkung,
- durch eine Erweiterung oder
- jede andere Art der Ableitung

verhindert werden. Verwendet wird dafür das Attribut `final`. Dieses Attribut kennt die drei möglichen Werte `#all`, `extension` und `restriction`.

Durch die folgende Syntax wird erreicht, dass die Ableitung durch Einschränkung von Wertebereichen einzelner Elemente nicht möglich ist.

Listing 4.52 Die Ableitung durch Einschränkung wird verhindert.

```
<complexType name="Adresse" final="restriction">
   <sequence>
      <element name="Name" type="string"/>
      <element name="Straße"  type="string"/>
      <element name="Ort" type="string"/>
   </sequence>
</complexType>
```

Ähnlich einfach sieht die Sache aus, wenn bestimmte oder alle Ableitungen innerhalb eines XML Schemas verhindert werden sollen. In diesem Fall kann man dem Wurzelelement `xsd:schema` ein entsprechendes `finalDefault`-Attribut zuweisen.

```
<xsd:schema ... finalDefault="restriction">
```

Auch dem `finalDefault`-Attribut können wieder die drei Attribute `#all`, `extension` und `restriction` zugewiesen werden.

Neben der Steuerung der Ableitung von Typen gibt es in XML Schema auch einen Mechanismus, der regelt, welche abgeleiteten Typen und Ersetzungsgruppen innerhalb von Dokumentinstanzen verwendet werden können. Zum Einsatz kommt dabei das `block`- bzw. das `blockDefault`-Attribut.

Listing 4.53 Das `block`-Attribut in Aktion.

```
<complexType name="Adresse" block="restriction">
   <sequence>
      <element name="Name" type="string"/>
      <element name="Straße" type="string"/>
      <element name="Ort" type="string"/>
   </sequence>
</complexType>
```

Durch diese Syntax wird verhindert, dass durch Einschränkung von Adresse abgeleitete Typen in der Instanz Adresse ersetzen. Neben `restriction` können auch hier die Werte `extension` und `#all` eingesetzt werden.

4.11.4 Abstraktionen

Eine weitere interessante Möglichkeit besteht darin, zunächst Datentypen als abstrakt zu definieren, um daraus unterschiedliche konkrete Typen abzuleiten. Dabei kann der abstrakte Datentyp selbst nicht direkt innerhalb einer Dokumentinstanz verwendet werden, sondern nur ein Datentyp seiner Ableitungen. Um einen Datentypen als abstrakt zu definieren, wird das Attribut `abstract` verwendet. Wie das in der Praxis aussieht, zeigt das folgende Beispiel:

Listing 4.54 So sieht Abstraktion aus.

```
<schema xmlsns="http://www.w3.org/2001/XMLSchema"
   targetNamespace="http://buch.hanser.de/schema"
   xmlns:ziel="http://buch.hanser.de/schema">

   <complexType name="Buecher" abstract="true"/>
   <complexType name="Roman">
         <complexContent>
            <extension base="ziel:Buecher"/>
         </complexContent>
   </complexType>

   <complexType name="Sachbuch">
      <complexContent>
         <extension base="ziel:Buecher"/>
      </complexContent>
   </complexType>
   <element name="Kategorie" type="ziel:Buecher"/>

</schema>
```

Das Element `Kategorie` ist nicht abstrakt, es kann also innerhalb einer Dokumentinstanz vorkommen. Da jedoch seine Typdefinition abstrakt ist, kann es nicht in einer Dokumentinstanz auftreten, ohne dass über ein `xsi:type`-Attribut auf einen abgeleiteten Typ verwiesen wird. Mithilfe von `xsi:type` kann man den abstrakten Datentypen durch einen der möglichen konkreten Datentypen ersetzen, der von ihm abgeleitet wurde. Somit wird die Ersetzung also erzwungen.

Folgendes wäre nicht gültig, da der Typ des `Kategorie`-Elements abstrakt ist.

```
<Kategorie xmlns="http://buch.hanser.de/schema"/>
```

Korrekt würde das Ganze aber folgendermaßen aussehen:

Listing 4.55 Diese Syntax ist korrekt.

```
<Kategorie xmlns="http://buch.hanser.de/schema"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:type="Roman"/>
```

In diesem Fall wird ein nicht abstrakter Typ benutzt, der für `Buecher` substituierbar ist.

4.11.5 Gemischte Inhalte

Interessant ist die Möglichkeit, in komplexen Datentypen Kindelemente mit Textinhalten zu mischen. Eine solche Mischung ist bereits fest in der XML-Spezifikation verankert. Allerdings gibt es dort kein Verfahren, mit dem sich die Anzahl und die Reihenfolge der eingefügten Elemente kontrollieren lassen. Mit XML Schema ist so etwas nun möglich.

Hier zunächst ein Beispiel für ein XML-Dokument:

Listing 4.56 Ein typischer Brief

```
<Anschreiben>
   <Anrede>Sehr geehrter Herr <Name>Ullrich</Name>,</Anrede>
   Vielen Dank für die Bestellung vom <Datum>1.2.2010</Datum>.
   Wir schicken Ihnen die <Anzahl>5</Anzahl> Bücher von <Buch>
   About a boy</Buch>zu.
</Anschreiben>
```

Um die Mischung aus Text und Elementen zu ermöglichen, wird dem `xsd:complexType` das Attribut `mixed` mit dem Wert `true` zugewiesen. Insgesamt sieht das Ganze dann folgendermaßen aus:

Listing 4.57 Elemente und Text werden gemischt.

```
<xsd:element name="Anschreiben">
  <xsd:complexType mixed="true">
    <xsd:sequence>
      <xsd:element name="Anrede">
        <xsd:complexType mixed="true">
          <xsd:sequence>
            <xsd:element name="Name" type="xsd:string"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="Datum"type="xsd:date"/>
      <xsd:element name="Anzahl" type="xsd:positiveInteger"/>
      <xsd:element name="Buch" type="xsd:string"/>
      [...]
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

Das `mixed`-Modell aus XML Schema unterscheidet sich deutlich von dem `mixed`-Modell in XML 1.0. So müssen beim gemischten Inhaltsmodell in XML Schema Ordnung und Anzahl der Kindelemente in einer Instanz mit der im Schema angegebenen Ordnung und

Anzahl der Kindelemente übereinstimmen. Anders sieht das beim gemischten Inhaltsmodell von XML 1.0 aus. Dort lassen sich Ordnung und Anzahl der Kindelemente in einer Instanz nicht beschränken.

4.11.6 Leeres Inhaltsmodell

Es gibt auch komplexe Elementtypen, die leer sind. Hierzu ein Beispiel:

```
<preis waehrung="EUR" wert="9.96"/>
```

Das `preis`-Element enthält sowohl die Währung wie auch den Preis. Beides wird ihm mittels Attributwerten zugewiesen. Dieses Element besitzt keinen Inhalt, sein Inhaltsmodell ist also leer. Um einen Typ mit leerem Inhaltsmodell definieren zu können, wird in Wirklichkeit ein Typ definiert, der ausschließlich Elemente als Inhalt erlaubt, aber dann keine Elemente definiert.

Listing 4.58 Ein anonymer Typ wird definiert.

```
<xsd:element name="preis">
  <xsd:complexType>
    <xsd:complexContent>
      <xsd:restriction base="xsd:anyType">
        <xsd:attribute name="waehrung" type="xsd:string"/>
        <xsd:attribute name="wert" type="xsd:decimal"/>
      </xsd:restriction>
    </xsd:complexContent>
  </xsd:complexType>
</xsd:element>
```

In diesem Beispiel wird ein anonymer Typ mit komplexem Inhalt definiert, es sind also ausschließlich Elemente enthalten. Mittels `complexContent` wird angegeben, dass das Inhaltsmodell eines komplexen Typs eingeschränkt oder erweitert werden soll. Im aktuellen Beispiel wird mittels `restriction` eingeschränkt, und zwar `anyType`. Dabei werden zwei Attribute deklariert, allerdings kein Elementinhalt.

Die zuvor gezeigte Syntax war recht komplex. XML Schema stellt daher eine Möglichkeit zur Verfügung, wie dasselbe durch eine deutlich kürzere Syntax erreicht werden kann.

Listing 4.59 Verkürzt geht es auch.

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

Diese Variante funktioniert, da die Definition komplexer Typen, die weder ein `simpleContent`- noch ein `complexContent`-Element enthalten, als Abkürzung für eine Definition, die `anyType` einschränkt, interpretiert wird.

Innerhalb eines XML-Dokuments dürfen Elemente fehlen, da z.B. die entsprechenden Informationen noch nicht verfügbar sind. Sie können explizit das Fehlen von Elementen erlauben. Das gelingt beispielsweise mit der Angabe `minOccurs="0"`. Ebenso können Sie

aber auch bei einem Element ausdrücklich erlauben, dass es als gültig akzeptiert wird, auch wenn es keinen Inhalt besitzt. Dazu muss dem betreffenden Element das Attribut `nillable` zugewiesen werden.

```
<preis waehrung="EUR" wert="9.96" nillable="true"/>
```

Wie dieses Beispiel zeigt, verwendet man also keinen speziellen Nil-Wert als Inhalt des Elements, sondern ein entsprechendes Attribut. So kann man später im XML-Dokument selbst das `nil`-Attribut verwenden.

```
<preis waehrung="EUR" wert="9.96" xsi:nil="true"/>
```

Bei `nil` handelt es sich um ein Attribut, das zum Namensraum von XML-Schema-instance gehört. Dadurch muss das dazugehörende Präfix unbedingt mit angegeben werden. (Im gezeigten Beispiel wurde `xsi` verwendet, Sie können natürlich auch ein anderes nehmen.)

4.12 Die Möglichkeiten der Wiederverwendbarkeit

Benannte Datentypen dürfen innerhalb eines Schemas mehrfach verwenden werden. Dieses Prinzip bzw. so etwas Ähnliches kennen Sie von DTDs. Dort werden für den gleichen Zweck allerdings Parameterentitäten eingesetzt. Im Gegensatz dazu ist XML Schema aber deutlich flexibler.

4.12.1 Benannte Typen

Manchmal ist es sinnvoll, einen komplexen Datentyp zu entwickeln, diesen aber an mehreren Stellen über verschiedene Namen anzusprechen. Das könnte z.B. der Fall sein, wenn man ein Bestellformular entwirft. Auf einem solchen Formular wird für gewöhnlich nach einer Rechnungs- und nach einer Lieferadresse gefragt. Denn bekanntermaßen können beide Adressen voneinander abweichen.

In diesen Fällen ist der Aufbau der Datenblöcke identisch. Daher kann man einen komplexen Datentyp entwerfen und diesen innerhalb der Elementdeklaration über verschiedene Namen ansprechen.

Listing 4.60 Ein Teil eines Bestellformulars

```
<xsd:complexType name="bestellung">
   <xsd:sequence>
      <xsd:element name="produkt" type="lgn"/>
       <xsd:element name="kunde" type="xsd:adresse"/>
   </xsd:sequence>
   <xsd:attribute name="isbn" use="required"/>
</xsd:complexType>

<xsd:complexType name="adresse">
   <xsd:sequence>
      <xsd:element name="vname" type="xsd:string"/>
      <xsd:element name="nname" type="xsd:string"/>
   </xsd:sequence>
</xsd:complexType>
```

4.12.2 Referenzen verwenden

Eine andere Art der Wiederverwendbarkeit stellt der Einsatz einer Referenz dar. Dabei werden die Elemente innerhalb einer Deklaration referenziert. Die Elemente müssen als globale Elemente deklariert sein. Verwendet wird dafür das `ref`-Attribut.

```
<xsd:element ref="adresse"/>
```

Im folgenden Beispiel wird das Element `kommentar` deklariert.

Listing 4.61 Das ist die Deklaration.

```
<xsd:schema ...
   ..
   <xsd:element name="kommentar" type="xsd:string"/
   ..
</xsd:schema>
```

Um nun innerhalb der Deklaration eines komplexen Elementes auf dieses Element zugreifen zu können, wird das `ref`-Attribut verwendet.

Listing 4.62 Hier erfolgt der Zugriff auf das Element.

```
<xsd:complexType name="bestellung"...
   <xsd:sequence->
      ...
      xsd:element ref="xsd:kommentar"/>
   </xsd:sequence--->
</xsd:complexType>
```

4.13 Schmeta inkludieren und importieren

XML Schema ermöglicht die Wiederverwendbarkeit fremder Schemata. Dafür hält die Spezifikation drei verschiedene Elemente parat.

- `xsd:include`
- `xsd:redefine`
- `xsd:import`

Alle drei Varianten werden auf den folgenden Seiten vorgestellt.

4.13.1 Schemata inkludieren

Den Anfang macht die Möglichkeit, Schemata zu inkludieren. Verwendet wird dafür das Element `xsd:include`. Mit diesem Element kann der Zielnamensraum eines Schemas aus mehreren Schema-Dokumenten zusammengefügt werden. Einen vergleichbaren Effekt kennen Sie vielleicht von den DTDs. Dort wird das mithilfe externer Parameterentitäten möglich, die Element- und Attribut-Deklarationen von außen übernehmen.

Ein Beispiel:

Listing 4.63 Die Deklarationen werden inkludiert.

```
<xsd:schema xmlns="http://www.w3.org/2001/XMLSchema"
   xmlns:buecher="http://www.hanser.de/buecher"
   targetNamespace="http://www.hanser.de/buecher">

  ...

   <xsd:include schemaLocation="http://www.hanser.de/schemata/java.xsd"/>
   <xsd:include schemaLocation="http://www.hanser.de/schemata/xml.xsd"/>

  ...

</schema>
```

Integrierten lässt sich das externe Schema über `xsd:include`. Durch das Attribut `schemaLocation` wird angegeben, wo das Schema liegt, das inkludiert werden soll. Für beide Schemata muss derselbe Zielnamensraum angegeben werden. Denn die Komponenten aus dem eingeschlossenen Schema werden in den Namensraum des aufnehmenden Schemas eingefügt. Befindet sich innerhalb des eingeschlossenen Schemas keine Angabe zum Namensraum, werden die Komponenten automatisch in den Namensraum des aufnehmenden Schemas übernommen.

4.13.2 xsd:redefine einsetzen

Eine andere Variante stellt die Verwendung von `redefine` dar. Dabei kann `redefine` anstelle von `xsd:include` verwendet werden. Das `xsd:redefine`-Element ermöglicht, dass einfache und komplexe Typen, Gruppen und aus externen Schema-Dateien abgerufene Attributgruppen im aktuellen Schema neu definiert werden können.

Listing 4.64 Der Name des Typs ändert sich hierbei nicht.

```
<xsd:schema xmlns="http://www.w3.org/2001/XMLSchema"
   xmlns:buecher="http://www.hanser.de/buecher"
   targetNamespace="http://www.hanser.de/buecher">

  ...

  <xsd:redefine schemaLocation="
                http://www.hanser.de/schemata/java.xsd">
    <xsd:complexType name="Autor">
      <xsd:complexContent>

        <xsd:restriction base="buecher:Autor">
          <xsd:sequence>
            <xsd:element name="verlag"
             type="xsd:string" minOccurs="10" maxOccurs="10"/>
          </xsd:sequence>
        </xsd:restriction>
      </xsd:complexContent>
    </xsd:complexType>
  </redefine>

  ...

  <xsd:include schemaLocation="http://www.hanser.de/schemata/xml.xsd"/>

  ...

</xsd:schema>
```

4.13.3 Das xsd:import-Element verwenden

Zu guter Letzt wird das Element `xsd:import` vorgestellt. Der Vorteil dieses Elements liegt darin, dass dabei auch Schemata genutzt werden können, die zu einem anderen Namensraum gehören. Diese werden mit einem Präfix versehen, wodurch sich Schema-Bestandteile aus verschiedenen Namensräumen verwenden lassen.

Listing 4.65 So wird `xsd:import` verwendet.

```
<xsd:schema xmlns="http://www.w3.org/2001/XMLSchema"
   xmlns:pcTeile="http://www.hanser.de/buecher"
   targetNamespace="http://www.hanser.de/verlag">

  ...

  <xsd:import namespace="http://www.hanser.de/buecher"/>

  ...
    <...
      <xsd:attribute name="art" type="buecher:sachbuch"/>
    .../>

  ...

</xsd:schema>
```

Es können ausschließlich Komponenten importieren werden. Dazu gehören Elemente, komplexe oder einfache Datentypen, die in dem ursprünglichen Schema global deklariert wurden, also Kindelemente von `xsd:schema` sind.

Innerhalb des importierten Schemas können die Importe hingegen sowohl innerhalb der Definition wie auch von Deklarationen auf jeder gewünschten Ebene übernommen werden.

4.14 Das Schema dem XML-Dokument zuordnen

Interessant ist zum Abschluss natürlich noch die Frage, wie sich Schemata in XML-Dokumente einbinden lassen. Vom W3C wurde für diesen Zweck ein spezieller Namensraum geschaffen.

```
http://www.w3.org/2001/XMLSchema-instance
```

Diesem Namensraum weist man für gewöhnlich das Präfix `xsi` oder `xs` zu.

```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

Für die Verknüpfung von Schema und XML-Dokument können die beiden Attribute `xsi:schemaLocation` und `xsi:noNamespaceSchemaLocation` verwendet werden. Nötig ist das allerdings nicht unbedingt, da der XML-Prozessor auch auf andere Art ermitteln kann, mit welchem Schema ein Dokument verknüpft ist.

Welche der beiden Attribute man einsetzt, hängt davon ab, ob Namensräume verwendet werden oder nicht. Hat man im Schema einen Zielnamensraum definiert, wird der XML-Prozessor überprüfen, ob der Name mit dem Namensraumnamen in der Dokumentinstanz übereinstimmt. Ist das der Fall, kann das Dokument anhand des Schemas auf Gültigkeit überprüft werden.

Im folgenden Beispiel wird ausgedrückt, dass der Standard-Namensraum *http://www.w3.org/1999/xhtml* ist. Anschließend wird mittels `xsi:schemaLocation` der URI für den Namensraum angegeben. Bei dieser Angabe wird dann auch gleich noch, getrennt durch einen Leerraum, der URI für das Schema als Wert zugeordnet.

Listing 4.66 Namensraumangaben in der Praxis.

```
<katalog xmlns="http://www.w3.org/1999/xhtml"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.hanser.de/katalog/"
                      "http://www.hanser.de/katalog/katalog.xsd"
   nummer="10001" datum="01.02.2010">
```

Im folgenden Beispiel wird kein Namensraum für das Schema selbst verwendet. Stattdessen kommt hier das Attribut `noNamespaceSchemaLocation` zum Einsatz.

Listing 4.67 Ein Namensraum wird nicht eingesetzt.

```
<katalog xmlns="http://www.w3.org/1999/xhtml"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:noNamespaceSchemaLocation="katalog.xsd"
   nummer="10001" datum="01.02.2010">
```

5

5 XPath, XPointer und XLink

Die Struktur von XML-Dokumenten lässt sich in hierarchischen Baumstrukturen darstellen. Dort erscheinen die verschiedenen Informationseinheiten in Form von Knoten.

Interessant ist nun die Frage, wie auf die einzelnen Teile eines XML-Dokuments zugegriffen werden kann. Zu diesem Zweck wurden bereits in den Anfängen von XML spezielle Adressierungssprachen als Ergänzung zu XML definiert. Standardisiert wurden diese unter dem Namen XML Path Language – XPath. Auf der Basis von XPath wiederum wurde mit XML Pointer Language – XPointer – die Möglichkeit geschaffen, auf einzelne Teile externer XML-Dokumente zuzugreifen.

Neben XPath und XPointer gibt es auch noch die XML Linking Language – XLink. Dieser Ansatz definiert die Möglichkeit, Elemente innerhalb von XML-Dokumenten für die Verknüpfung mit anderen Ressourcen zu verwenden, wobei auch dabei XPointer eingesetzt werden kann.

5.1 XPath – alles zum Adressieren

In diesem Kapitel werden die verschiedenen Abfragesprachen vorgestellt. Den Anfang macht dabei XPath. Standardisiert wird diese Sprache vom W3C. Die Spezifikation zu XPath 2.0 ist unter *http://www.w3.org/TR/xpath20/* zu finden.

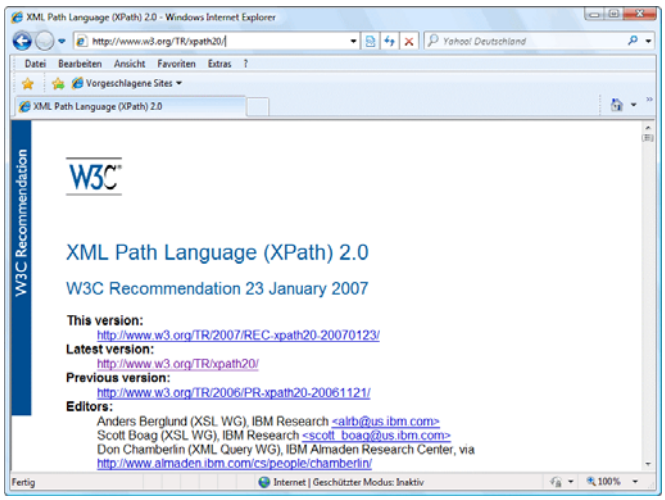

Abbildung 5.1 Die Empfehlung zu XPath 2.0

Bei XPath selbst handelt es sich nicht um eine XML-Anwendung. Vielmehr enthält die Sprache eine eigene Syntax, mit der sich Zeichenketten bilden lassen, die für die Adressierung von Elementen eines Dokuments verwendet werden können.

Im weiteren Verlauf dieses Buches lernen Sie auch den Umgang mit XSLT kennen. Bereits in diesem Kapitel werden Ihnen aber immer wieder Elemente aus dieser Sprache begegnen. Der wichtigste Begriff dabei ist sicherlich Templates. In Templates wird die gesamte Ausgabe des entsprechenden Dokuments definiert.

5.1.1 Die Idee hinter XPath

Es musste eine Sprache gefunden bzw. entwickelt werden, mit der auf die einzelnen Elemente von XML-Dokumenten zugegriffen werden kann. Bevor es ins XPath-Detail geht, noch einige allgemeine Hinweise, die für das Verständnis von XPath wichtig sind. Im Zusammenhang mit XPath werden Ihnen immer wieder die beiden Begriffe Baumstruktur und Knoten begegnen.

XPath arbeitet auf Basis eines Baummodells, in dem die im XML-Dokument enthaltenen Informationseinheiten repräsentiert sind, die in der Infoset-Empfehlung definiert sind.

5.1.1.1 Hintergrundinformationen zu Infoset

Obwohl im XML-Standard selbst und auch in angrenzenden Spezifikationen die korrekte XML-Syntax exakt festgelegt ist, gibt es Unklarheiten dahingehend, worin denn eigentlich die Kerninformationen im Einzelnen bestehen. So kann jeder XML-Parser auf Verstöße hinsichtlich der Wohlgeformtheit mit Fehlermeldungen reagieren. Dabei gehen die jeweiligen XML-Parser von unterschiedlichen Mengen an Kerninformationen aus. Für Sie bedeutet das in der Praxis, dass XML-Parser unterschiedliche Informationsmengen an andere Programme weitergeben.

Um auf diesem Gebiet ebenfalls eine Einigkeit herzustellen, hat das W3C die Empfehlung XML Information Set, kurz XML-Infoset (*http://www.w3.org/TR/xml-infoset/*) herausgegeben. Hierbei wird als Information Set exakt das spezifiziert, was gemeinhin als Informationsmenge bzw. Informationseinheit bezeichnet wird.

Die Infoset-Empfehlung unterscheidet zwischen elf verschiedenen Informationseinheiten, die jeweils exakt definierte Eigenschaften besitzen.

- Document Information Item (Dokument)
- Element Information Items (Elemente)
- Attribute Information Items (Attribute)
- Processing Instruction Information Items (Verarbeitungsanweisungen)
- Unexpanded Entity Reference Information Items
- Character Information Items (Zeichendaten)
- Comment Information Items (Kommentare)
- The Document Type Declaration Information Item
- Unparsed Entity Information Items
- Notation Information Items
- Namespace Information Items (Namensräume)

Weiterführende Informationen zu Infoset finden Sie im entsprechenden Arbeitsentwurf unter *http://www.w3.org/TR/xml-infoset*.

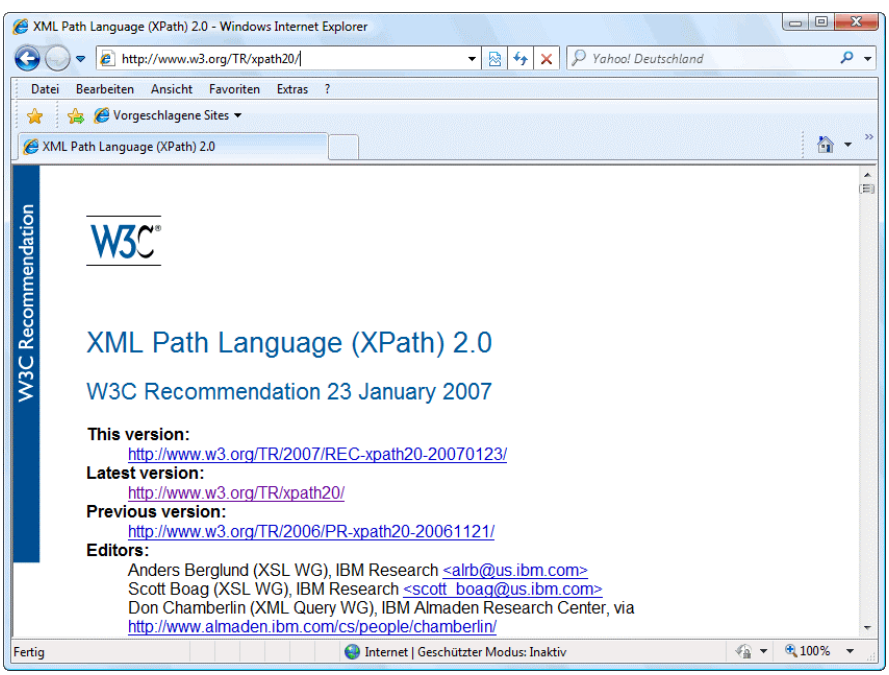

Abbildung 5.2 Die Empfehlung zu XML Information Set

5.1.1.2 Das Baummodell und die XPath-Ausdrücke

XPath arbeitet auf Basis eines sogenannten Baummodells, durch das die im XML-Dokument enthaltenen Informationseinheiten repräsentiert werden. Vergleichbar ist dieses Baummodell mit dem Document Object Model (DOM). Ausführliche Informationen zum DOM dann später im Buch. Aber auch wenn beide Modelle vergleichbar sein mögen, identisch sind sie nicht. Das DOM ist ein Objektmodell mit Knoten, dessen Eigenschaften und Methoden einem entsprechenden Anwendungsprogramm zur Verfügung gestellt werden. Knoten im XPath-Modell sind hingegen verhaltenslos.

Im Zusammenhang mit XPath wird Ihnen immer wieder der Begriff Ausdruck begegnen. Dabei handelt es sich um eine Instanz der XPath-Sprache. Um ein Objekt zu gewinnen, wird der XPath-Ausdruck ausgewertet. Für dieses Objekt sind verschiedene Datentypen verfügbar.

- `node-set` – Dieser Datentyp liefert eine untergeordnete Knotenmenge, die auch leer sein kann.
- `string` – Es handelt sich um eine Zeichenfolge. Auch die kann leer sein.
- `number` – Es handelt sich um eine Zahl im 64-Bit-Format nach IEEE 754.
- `boolean` – Dieser Datentyp besitzt einen der beiden Werte `true` und `false`.

Zwischen den Datentypen sind Umwandlungen möglich. Wie diese aussehen können, ist in der XPath-Empfehlung festgelegt.

XPath übernimmt nicht alle in der Infoset-Spezifikation definierten Informationseinheiten. Entitäten sind in XPath nicht ansprechbar. Der Grund dafür ist eigentlich ganz einfach. XPath-Ausdrücke beziehen sich immer auf das bereits vom Prozessor geparste Dokument. Und dort sind die Entitäten bereits aufgelöst und durch die entsprechenden Zeichenfolgen ersetzt.

Den Einstieg in die XPath-Welt schafft man am leichtesten über ein Beispiel. Werfen Sie zunächst einen Blick auf die folgende XML-Struktur.

Listing 5.1 Ein solches Dokument kennen Sie schon.

```
<?xml version="1.0"?>
<bibliothek>
<buecher>

   <buch id="1" land="de">
      <titel>Fever Pitch</titel>
      <autor>Nick Hornby</autor>
   </buch>

   <buch id="3" land="ene">
      <titel>Global Fish</titel>
      <autor>Rainald Grebe</autor>
   </buch>

</buecher>
</bibliothek>
```

Trifft ein XPath-Prozessor auf dieses Dokument, übersetzt er es in eine Baumstruktur.

Der Baum beginnt beim Wurzelknoten, der nicht dem Dokumentelement entspricht. Vielmehr handelt es sich dabei um ein logisches Konstrukt, von dem zunächst das Dokumentelement und eventuelle Knoten für Kommentare und Verarbeitungsanweisungen abzweigen.

5.1.1.3 Die Dokumentreihenfolge verstehen

Es gibt für die Zuordnung der Knoten des Baums zu den Komponenten des XML-Dokuments feste Zuordnungen. Auch dieser Punkt lässt sich wieder am besten anhand eines praktischen Beispiels zeigen.

Werfen Sie zunächst einen Blick **Abbildung 5.3.**

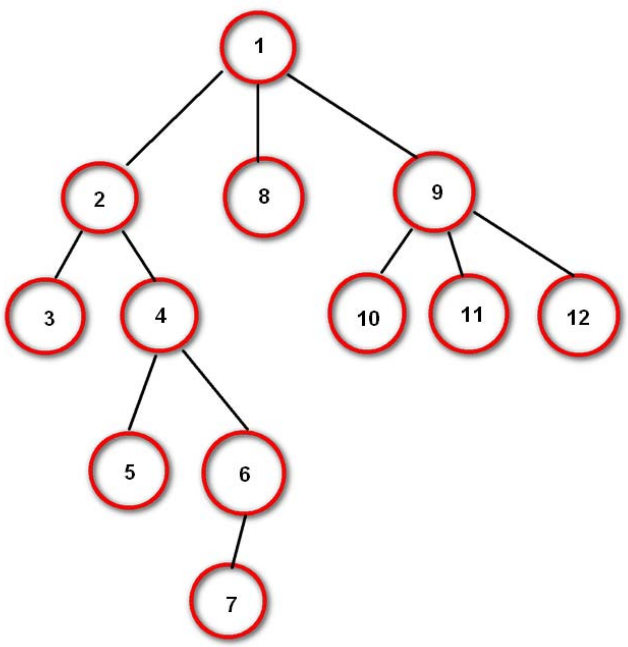

Abbildung 5.3 Die Ziffern beschreiben die Reihenfolge der Knoten, die der Dokumentreihenfolge entspricht.

Es handelt sich hierbei um eine abstrakte Struktur für einen entsprechend angeordneten Baum.

Die innerhalb der Knoten stehenden Zahlen geben die Reihenfolge an, in der ein XML-Prozessor die Knoten ablaufen wird. (Wenn er dazu angehalten ist, alle Knoten des Dokuments abzulaufen.)

Bezeichnet wird diese Anordnung gemeinhin als Dokumentreihenfolge bzw. als Document Ordner. Diese entspricht exakt der Reihenfolge, in der die den Knoten entsprechenden Teile innerhalb des Dokuments auftauchen.

Ein weiterer Aspekt wird deutlich: Innerhalb des Baums findet eine sogenannte Tiefensuche statt. Dabei werden zunächst nicht alle Knoten derselben Ebene durchlaufen. Vielmehr werden immer erst die Kinder und Kindeskinder des aktuellen Knotens abgearbeitet.

Soll aus einem auf diese Weise geordneten Baum wieder ein sequenzielles Dokument erzeugt werden, muss man es zunächst plätten, und die Knoten müssen in der entsprechenden Reihenfolge im Dokument abgelegt werden. Dabei spricht man von der Serialisierung. Von Deserialisierung ist hingegen die Rede, wenn ein Baum aus einem Dokument erzeugt wird.

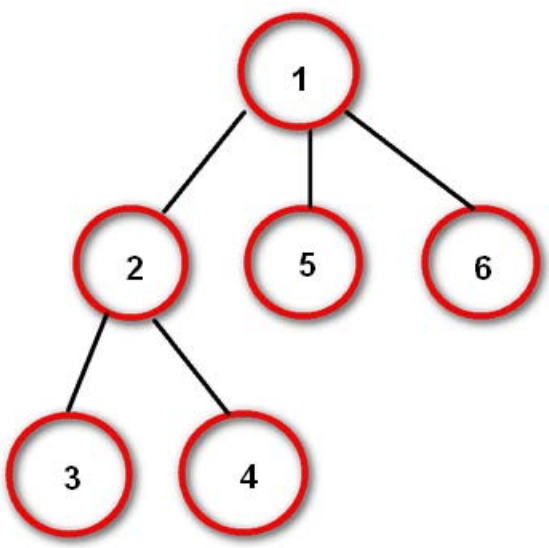

Abbildung 5.4 Die Dokumentreihenfolge entspricht exakt der Reihenfolge der Start-Tags innerhalb des Dokuments.

Im folgenden Beispiel entspricht die Dokumentreihenfolge der Reihenfolge der Start-Elemente im Dokument.

Listing 5.2 So sieht die Dokumentreihenfolge aus.

```
1 <buecher>
2    <autor>
3       <name>Hornby</name>
4       <vorname>Nick</vorname>
     </autor>
5    <titel>Fever Pitch</titel>
6    <ordnung>Roman</ordnung>
  </buecher>
```

5.1.2 Mit Knoten arbeiten

In XPath gibt es verschiedene Arten von Knoten. Genau genommen sind es sieben Knotentypen. Die Knotentypen sind selbst nicht unmittelbarer Bestandteil der Adressierungssyntax von XPath. Allerdings ist es natürlich hilfreich, die Begrifflichkeiten zu kennen.

- Root-Knoten (Root Node) – Die Wurzel des XML-Dokuments. Der Wurzelknoten ist selbst kein Element, sondern lediglich die abstrakte Ursprungswurzel der Baumstruktur des Dokuments.
- Elementknoten (Elements Node) – Jedes Element stellt aus Sicht von XPath einen Elementknoten dar. Um den Zugriff auf ein Element zu erleichtern, kann es mit einer ID eindeutig beschrieben werden. Der erweiterte Name eines Elementknotens ist der Elementname selbst. Der Textwert ist der Textwert des Elements und seiner Kinder.
- Attributknoten (Attribute Node) – Jedes Attribut, das zu einem Element gehört, ist ein Attributknoten. Anders als beim DOM ist ein Attributknoten hier aber kein Kind des Elements. Attribute können dementsprechend auch nicht als Kinder eines Elements angesprochen werden. Der erweiterte Name des Knotens ist der Attributname mit eventuell vorhandenem Präfix. Der Textwert des Knotens ist der Attributwert.
- Kommentarknoten (Comment Node) – Jeder Kommentar ist ein Kommentarknoten. Kommentarknoten besitzen keinen erweiterten Namen. Alles was zwischen `<!--` und `-->` steht, ist der Textwert des Knotens.
- Namensraumknoten (Namespace Node) – Elemente und Attribute können Angaben zum Namensraum enthalten, wenn entsprechende Namenräume in die XML-Datei importiert werden. Namensraumknoten sind alle Attributknoten mit dem Namen `xmlns` oder dem Präfix `xmlns`. Ebenso sind alle Knoten Namensraumknoten, wenn ein Vorfahrenelement das Präfix `xmlns` besitzt. (Dieses darf vorher allerdings nicht aufgehoben worden sein.) Der erweiterte Name eines Namensraumknotens ist der lokale Name des Namensraums. Der Textwert ist der URI, mit dem der Namensraum verbunden wurde.
- Processing-Instruction-Knoten (Processing Instruction Node) – Verarbeitungsanweisungen stehen außerhalb des eigentlichen Dokumentbaums. Verarbeitungsanweisungen beginnen mit `<?` und enden mit `?>`. Aus Sicht von XPath sind sie ein eigener Knotentyp. Die XML-Deklaration `<?xml version="1.0"?>` zu Beginn von XML-Elementen, ist allerdings keine Verarbeitungsanweisung, in diesem Sinne.
- Textknoten – Sämtlicher Inhalt des Dokuments, der nicht zu einem der zuvor genannten Knoten gehört, ist ein Textknoten. Neben dem reinen Text gehören auch `CDATA`-Abschnitte zu den Textknoten. Diese werden in normalen Text umgewandelt und nicht etwa als Elementknoten betrachtet. Textknoten besitzen keinen erweiterten Namen.

Nachdem Sie die verschiedenen Knotentypen kennengelernt haben, soll ein praktisches Beispiel den Einstieg in XPath ebnen. Als Ausgangspunkt wird dabei die folgende XML-Datei angenommen:

Listing 5.3 Das ist das Ausgangsdokument.

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="bibliothek.xsl"?>
<bibliothek>

   <autor id="a1" land="eng">
      <vorname>Nick</vorname>
      <nachname>Hornby</nachname>
   </autor>

   <autor id="a2" land="usa">
      <vorname>Bret Easton</vorname>
      <nachname>Ellis</nachname>
   </autor>

</bibliothek>
```

Für die Ausgabe dieser Datei wird auf ein einfaches XSLT-Stylesheet zurückgegriffen.

Listing 5.4 Diese Syntax sorgt für die Ausgabe.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html" encoding="UTF-8" indent="yes" />
   <xsl:template match="autor">
      <xsl:value-of select="."/><br />
   </xsl:template>
</xsl:stylesheet>
```

Innerhalb dieser Datei wird lediglich das Template für die Autoren aufgerufen. Mittels

```
<xsl:value-of>
```

variiert man dabei die XPath-Aufrufe. Der innerhalb des aktuellen Beispiels verwendete Punkt steht dabei für den aktuellen Knoten, also für `autor`.

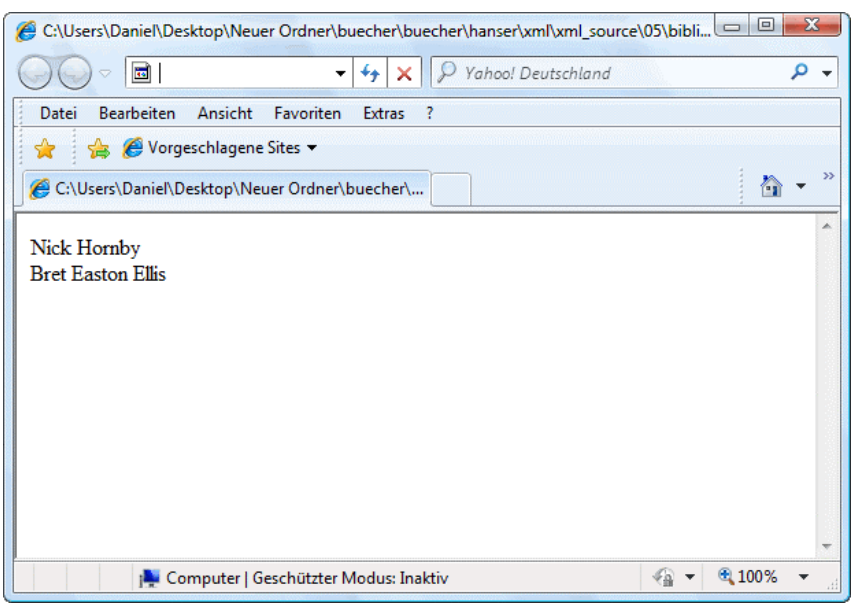

Abbildung 5.5 Die Autorennamen werden ausgegeben.

Ebenso kann man natürlich auf untergeordnete Knoten zugreifen. Dazu muss man lediglich den entsprechenden Knotennamen angeben.

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

In diesem Beispiel wird auf `vorname` zugegriffen.

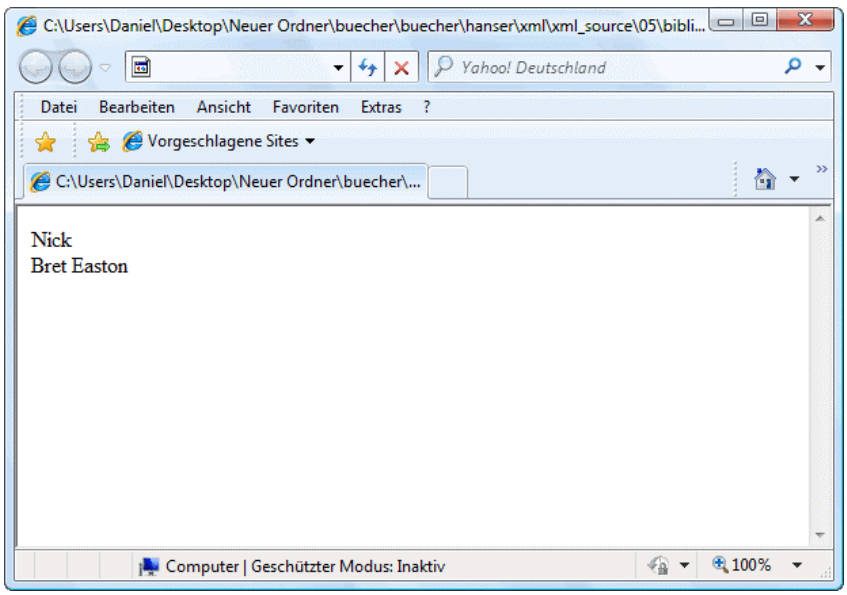

Abbildung 5.6 Jetzt sind nur noch die Vornamen zu sehen.

Ähnlich einfach funktioniert der Zugriff auf Unterknoten. Würde sich im vorherigen Beispiel also unter `vorname` das Element `zweitervorname` befinden, sähe die Syntax folgendermaßen aus:

```
vorname/zweitervorname
```

Die betreffenden Knoten müssen in diesem Fall hintereinander, getrennt durch einen Schrägstrich, notiert werden. Ausführliche Informationen zur Adressierung gibt es im weiteren Verlauf dieses Kapitels. Zunächst aber noch ein paar Worte dazu, in welchen Beziehungen die Knoten zueinander stehen. In XPath werden für die Darstellung der Knotenbeziehungen Verwandtschaftsgrade verwendet.

- Elternknoten (Parent) – Ein übergeordneter Knoten ist der Elternknoten. Alle Elemente besitzen einen Elternknoten. Einzige Ausnahme ist das Wurzelelement.
- Vorfahre (Ancestor) – Vorfahren sind die Elternknoten, Elternknoten von Elterknoten und so weiter.
- Kind (Child) – Untergeordnete Knoten sind Kindknoten.
- Übergeordnetes Kind (Decendants) – Untergeordnete Kindknoten sind Kindknoten.
- Geschwister (Siblings) – Knoten, die den gleichen Elternknoten besitzen, sind Geschwisterknoten.

Mittels spezieller XPath-Ausdrücke kann man auf die einzelnen Knoten zugreifen. Entscheidend dafür ist das Verständnis davon, was Achsen sind und wie die Adressierung in XPath funktioniert.

5.1.3 Achsen – die Richtung der Elementauswahl

Über die Achsen wird festgelegt, in welche Richtung Elemente ausgewählt werden. Wobei Richtung in diesem Fall die Richtung innerhalb des Knotenbaums meint. In XPath gibt es Knoten, die von anderen Knoten abhängig sind, und solche, die sich auf der gleichen Ebene befinden. Dafür existieren Ausdrücke wie Eltern und Kind, die als Achsen (Axes) bezeichnet werden. Das klingt zunächst schrecklich theoretisch, wird anhand eines Beispiels aber schnell klar.

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

Durch diesen Ausdruck werden alle Kindknoten ermittelt, die das Elternelement `vorname` besitzen. Verwendet wird dafür die Achse `child`. Wollte man jetzt z.B. den Elternknoten von `vorname` ermitteln, würde man die `parent`-Achse verwenden.

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

In Tabelle 5.1 sind alle in XPath verfügbaren Achsen aufgeführt. Um das sehr theoretische Thema anschaulicher machen zu können, beziehen sich die in der Tabelle enthaltenen Beschreibungen auf das folgende Beispiel:

Listing 5.5 Das Ausgangsbeispiel

```
<a>
   <b>
      <c />
   </b>
   <b>
      <c />
   </b>
   <b>
      <c />
   </b>
</a>
```

Das ist die Syntax, von der ausgegangen wird.

Tabelle 5.1: Die verfügbaren Achsen

Achse	Beschreibung
Attribut (attribute)	Attribute haben eine Sonderstellung und werden nicht in das Schema child, parent, descendant und ancestor eingefügt. Bei einer angenommen Struktur von `<a b="att">` ist `b` das Attribut von `a`.
Kind (child)	Element `b` ist aus Sicht von `a` ein Kindelement.
Elternknoten (parent)	Element `a` ist aus Sicht von `b` der Elternknoten.
Nachkommen (descendant)	Die Elemente `c` und `b` sind aus Sicht von `a` Nachkommen.
Vorfahr (ancestor)	Aus Sicht von `c` sind die Elemente `a` und `b` Vorfahren.

Achse	Beschreibung
Nachfolgende Knoten (following)	Aus Sicht des ersten `c`-Elements sind das zweite `b`-Element und das zweite `c`-Element nachfolgende Knoten.
Vorherige Geschwisterknoten (preceding-sibling)	Aus Sicht des dritten `c`-Elements ist das zweite `c`-Element ein Geschwisterknoten, nicht aber das erste `c`-Element.
Nachfolgende Geschwisterknoten (following-sibling)	Aus Sicht des ersten `c`-Elements ist das zweite `c`-Element ein nachfolgender Geschwisterknoten, nicht aber das dritte `c`-Element.
Namensraum (namespace)	Bei dem angenommenen Elementnamen `a` ist `name` aus Sicht von `a` der Namensraum.
Der aktuelle Knoten (self)	Wird das Element `a` gerade vom XSLT-Prozessor bearbeitet, besitzt es in diesem Moment den Status *self*.
Nachkomme oder der aktuelle Knoten (descendant-or-self)	Wird das Element `a` gerade vom XSLT-Prozessor bearbeitet, haben `a` und das nachfolgende `b` den Status *descendant-or-self*.
Vorfahre oder der aktuelle Knoten (ancestor-or-self)	Wird das Element `b` gerade vom XSLT-Prozessor bearbeitet, haben `b` und das vorhergehende `a` den Status *descendant-or-self*.

Hinter dem Achsennamen müssen zwei Doppelpunkte stehen. Diese Syntax ist nötig, damit der Parser zwischen einem Elementnamen und einem Achsennamen unterscheiden kann.

5.1.4 Adressierung von Knoten

Um auf die Knoten eines XML-Dokuments zugreifen zu können, muss man sie auf irgendeine Art und Weise adressieren. In XPath bedient man sich dazu eines einfachen Mittels. Es wird vom aktuellen Knoten ausgehend ein bestimmter Pfad definiert. Diesem Pfad wiederum folgt der Interpreter, um zu dem betreffenden Knoten oder der entsprechenden Knotenmenge zu gelangen.

In XPath wird zwischen einer ausführlichen und einer verkürzten Achsenbezeichnung unterschieden. Zunächst ein Beispiel für die vollständige Variante.

```
/child::autor/child::buch/attribute::titel
```

Hier werden die kompletten Achsenbezeichnungen verwendet.

Eine andere Möglichkeit besteht darin, alles „Unnötige“ wegzulassen und sich tatsächlich nur auf die eigentlichen Elemente zu konzentrieren. Das Resultat ist eine deutlich verkürzte Syntax.

```
/autor/buch/@titel
```

Es bleibt letztendlich Ihnen überlassen, welche der beiden Varianten Sie nutzen. Für den Einstieg in die XPath-Welt ist sicherlich die erste Variante besser, da man dadurch besser die Achsenbezeichnungen nachvollziehen kann. Später, wenn etwas Routine eingekehrt ist, kann man aber ruhigen Gewissens zur Kurznotation greifen.

5.1.5 Der Unterschied zwischen absoluten und relativen Pfaden

Sie haben gesehen, dass es bei der Adressierung zwei Varianten gibt, einmal die ausführliche und einmal die verkürzte Achsenbezeichnung. Darüber hinaus existieren auch zwei verschiedene Notationen, wenn es um die Bestimmung der Pfade geht. In XPath wird nämlich zwischen absoluten und relativen Pfadangaben unterschieden.

Bei relativen Pfaden wird von dem Knoten ausgegangen, an dem sich der Interpreter momentan befindet. Ist diese Position innerhalb des Elementbaums, geht der relative Pfad vom aktuellen Knoten aus und zeigt auf einen anderen Knoten innerhalb des Baums.

Anders sieht das bei absoluten Pfadangaben aus. Dort wird der Pfad – Sie ahnen es schon – absolut angegeben. In diesem Fall spielt die Position des Interpreters keine Rolle.

Auch hinsichtlich der unterschiedlichen Varianten von Pfadangaben gilt, dass sich das am besten anhand eines Beispiels zeigen lässt. Dabei dient folgendes Dokument als Ausgangspunkt für weitere Erklärungen:

Listing 5.6 Das ist die Basis.

```
<bibliothek>

   <autoren>

   <autor id="a1" land="eng">
      <vorname>Nick</vorname>
      <nachname>Hornby</nachname>
   </autor>

   <autor id="a2" land="usa">
      <vorname>Bret Easton</vorname>
      <nachname>Ellis</nachname>
   </autor>

   </autoren>

</bibliothek>
```

Im ersten Schritt geht es um die Adressierung der Dokumentwurzel.

Listing 5.7 Es wird auf den Knoten zugegriffen.

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

Hier wird das Wurzelelement `bibliothek` über den Schrägstrich adressiert. Zusätzlich wird ein Template für die XML-Dokumentwurzel über `<xsl:apply-templates />` definiert.

5.1.5.1 Relative Pfadangaben benutzen

Bei dem gezeigten Beispiel handelt sich um eine verschachtelte Elementstruktur. Befindet sich der Interpreter beispielsweise bereits bei dem `autor`-Knoten und man will von dort auf den Knoten `vorname` zugreifen, dann sieht der Zugriff folgendermaßen aus:

Listing 5.8 Auch relative Angaben sind möglich.

```
<xsl:template match="autoren/autor">
   <xsl:value-of select="../vorname" />
</xsl:template>
```

Um das Kindelement `autor` aus Sicht des Großvaterelements `bibliothek` zu adressieren, muss beim `match`-Attribut des `xsl:template`-Elements der Pfad angegeben werden. Will man die verkürzte Syntax verwenden, sieht das so aus:

```
autoren/autor
```

Für die ausführliche Notation ergibt sich hingegen folgendes Bild:

```
child::autoren/child::autor
```

Auch wenn das Prinzip relativer Pfadangaben auf den ersten Blick vielleicht nicht ganz leicht zu verstehen ist, es ist eigentlich ganz einfach. Sie kennen das Prinzip sicherlich aus HTML. Dort werden z.B. für das Setzen von Verweisen oder das Einbinden von Grafiken oft relative Pfadangaben verwendet. Genauso funktioniert das auch in XPath. Am einfachsten versteht man das Prinzip der relativen Pfadangaben, wenn man sich vor Augen führt, dass man lediglich den hierarchiebestimmenden Knotennamen folgen muss.

5.1.5.2 Absolute Pfadangaben einsetzen

Absolute Pfadangaben werden mit einem Schrägstrich eingeleitet. Dieser Schrägstrich repräsentiert die Dokumentwurzel. Die weiteren Hierarchieebenen werden jeweils durch weitere Schrägstriche gekennzeichnet. Bei der ausführlichen Syntax werden die Knoten, über die der Pfad läuft, anhand ihrer Achsen und der dementsprechenden Knotennamen bezeichnet und durch zwei Doppelpunkte voneinander getrennt notiert.

Allerdings verwendet man meistens die verkürzte Syntax, bei der man ohne Achsenbezeichnungen auskommt.

Listing 5.9 Die kurze Schreibweise

```
<xsl:template match="/autoren/autor">
   <xsl:value-of select="/autoren/autor" />
</xsl:template>
```

In diesem Beispiel wird über `xsl:template` ein Template für das `autor`-Element definiert. Bei diesem Element handelt es sich um ein Kindelement von `autoren`. `autoren` wiederum ist ein Kind der Dokumentwurzel.

5.1.5.3 Attribute adressieren

Obwohl XPath die Elementknoten als Eltern ihrer Attributknoten behandelt, werden die Attributknoten nicht als Kindknoten des Elementknotens angesprochen, wie das bei Textknoten der Fall ist. Es muss also einen anderen Weg für den Zugriff geben.

Um auf den Wert eines Attributknotens zuzugreifen, kann man in der Kurzform eine Kombination aus dem `@`-Zeichen und dem Attributnamen verwenden. Angenommen, im XML-Dokument würde

```
<autor name="...">
```

existieren, dann sähe der Zugriff auf das `name`-Attribut folgendermaßen aus:

Listing 5.10 Es wird auf den Namen zugegriffen.

```
<xsl:template match="autor/@name">
   <xsl:text>Name:</xsl:text>
   <xsl:value-of select="." />
</xsl:template>
```

Mit einem Ausdruck wie

```
autor[@name="Mayer"]
```

lassen sich hingegen Elementknoten auswählen, bei denen das Attribut `name` den Wert `Mayer` besitzt.

5.1.5.4 Mit Wildcards arbeiten

Es gibt noch eine spezielle Syntax, mit der man ganz einfach mehrere Knoten auf einmal abfragen kann. Verwendet werden dabei sogenannte Wildcards. Auch hierzu zunächst wieder ein Beispiel.

Listing 5.11 Eine Wildcard im Einsatz.

```
<xsl:template match="bibliothek">
   <xsl:value-of select="*" />
</xsl:template>
```

Durch den Einsatz der Wildcard (*) werden in diesem Beispiel alle Kindelemente von `bibliothek` mit einem Schlag erfasst. Dazu muss innerhalb der Template-Definition lediglich dem `select`-Attribut das Stern- bzw. Wildcard-Zeichen zugewiesen werden.

Beachten Sie, dass durch die Wildcard lediglich die nächstuntere Hierarchieebene ausgewählt wird. Will man auf alle untergeordneten Hierarchieebenen zugreifen, unabhängig von der tatsächlichen Tiefe der Verzweigung des Baums, muss `//` verwendet werden.

5.1.5.5 Positionsangaben und Bedingungen

Sie haben nun die absolute und die relative Adressierung kennengelernt. Ein Problem beider Varianten ist, dass man allein dadurch noch nicht jedes Element ansprechen kann. Werfen Sie zum besseren Verständnis einen Blick auf die folgende Syntax:

Listing 5.12 Das ist das Ausgangsdokument.

```
<bibltiothek>
   <autor>...</autor>
   <autor>...</autor>
   <autor>...</autor>
</bibliothek>
```

Angenommen, Sie würden folgende Adressierung verwenden:

```
bibliothek/autor
```

In diesem Fall greift man lediglich auf das erste `autor`-Element zu. Was aber, wenn man auf das zweite oder dritte Elemente zugreifen möchte? Dieses Problem könnte man mit einer Schleife lösen. Aber auch die hilft nicht in jedem Fall weiter. (An dieser Stelle noch einmal der Hinweis, dass Schleifen usw. im weiteren Verlauf dieses Buches noch ausführlich vorgestellt werden.)

XPath hält genau für solche Zwecke eine ganz besondere Syntax bereit. Bei der kann man auf einzelne Elemente ganz gezielt mittels Positionsangaben und Bedingungen zugreifen. Wie das funktioniert, wird hier anhand einiger Beispiele gezeigt. Beachten Sie, dass dabei bereits einige XPath-Funktionen zum Einsatz kommen, die im weiteren Verlauf dieses Kapitels noch ausführlich vorgestellt werden.

Durch die folgende Syntax wird das aktuelle Element adressiert.

```
self::bibliothek
```

Das gilt allerdings nur für den Fall, dass es vom Typ `bibliothek` ist.

Ebenso einfach kann auf das jeweils letzte Kindelement zurückgegriffen werden.

```
autor[last()]
```

Verwendet wird dafür die XPath-Funktion `last()`. Im aktuellen Beispiel wird auf diese Weise das letzte Kindelement von `autor` ermittelt.

Ganz ähnlich sieht die Syntax aus, durch die der Zugriff auf das vorletzte Element vom Typ `autor` gelingt.

```
autor[last()-1]
```

Auch hier kommt die `last()`-Funktion zum Einsatz, die allerdings noch um den Zusatz `-1` erweitert wird.

Wenn Sie alle `autor`-Elemente adressieren wollen, die sich aus Sicht des aktuellen Knotens unterhalb von Elementen des Typs `bibliothek` befinden, setzen Sie die folgende Syntax ein:

```
bibliothek//autor
```

Um den gewünschten Effekt zu erzielen, trennen Sie beide Elemente durch zwei Schrägstriche.

Es geht sogar noch detaillierter. So wird z.B. durch die folgende Syntax vom dritten Autor aus dem zweiten Buch die fünfte Seite adressiert.

```
/bibliothek/autor[3]/buch[2]/seite[5]
```

Diese Syntax lässt sich beliebig anpassen. Wenn Sie also die sechste Seite adressieren wollen, dann notieren Sie `seite[6]`.

Abschließend wird gezeigt, wie das Kindelement `autor` adressiert wird. Das geschieht allerdings nur unter der Voraussetzung, dass das Attribut `land` den Wert `eng` besitzt.

```
autor[@land=''eng]
```

Sie haben gesehen, wie flexibel XPath bei der Adressierung tatsächlich ist. Einige der vorgestellten Varianten werden Ihnen im weiteren Verlauf dieses Buches noch mehrmals begegnen.

5.1.5.6 Knotentests – die Elementauswahl weiter einschränken

Die durch die gewählte Achse ausgewählte Knotenmenge lässt sich durch den sogenannten Knotentest weiter filtern. Dabei sind die Knotentests als zusätzliche Möglichkeiten zu den bereits beschriebenen Achsendefinitionen zu verstehen. Als Kriterium wird dabei entweder ein bestimmter Knotenname oder ein bestimmter Knotentyp verwendet.

So wählt

```
following::meinknoten
```

z.B. ausgehend vom aktuellen Kontextknoten zunächst alle in der Dokumentreihenfolge folgenden Knoten aus. Gleichzeitig werden dabei alle Knoten mit dem Elementnamen `meinknoten` herausgefiltert. Der Knotentyp des benannten Knotens muss mit dem grundsätzlichen Knotentyp der Achse übereinstimmen. Außer bei den Achsen `attribute` und `namespace` ist das bei allen Achsen der Knotentyp Element. Knoten eines anderen Typs werden ignoriert. Das gilt auch dann, wenn diese den gleichen Namen besitzen. Um Attributknoten zusammenzustellen, wird der Attributname verwendet.

```
attribute::meinattribut
```

Will man Knoten anhand ihres Knotentyps identifizieren, greift man anstelle des Knotennamens auf eine entsprechende Funktion zurück. Die folgende Übersicht zeigt alle verfügbaren Knotentests:

- Knotenname – Innerhalb der Knotenmenge wird nach dem Knoten gesucht, dessen erweiterter Name hier angegeben wurde.
- `*` – Alle Knoten innerhalb der Knotenmenge der Achsendefinition sollen gefunden werden. Der Name spielt dabei keine Rolle.
- `Präfx:*` oder `Präfix:Name`
- `comment()` – Wählt alle Kommentarknoten aus.
- `text()` – Wählt alle Textknoten.
- `node()` – Wählt Knoten eines beliebigen Typs aus.
- `processing-instruction()` – Wählt den Knoten aus, der die Verarbeitungsanweisung enthält.

Will man z.B. alle Textknoten auswählen, die Kinder des Kontextknotens sind, kann

```
child::text()
```

verwendet werden. Durch

```
descendant-or-self::coment()
```

kann man alle Kommentare des Dokuments zusammenfassen. Ebenso ist es aber auch möglich, der Funktion `processing-instruction` einen Parameter zu übergeben, durch den das Ziel bestimmt wird.

5.1.5.7 Mit Prädikaten filtern

Neben den Knotentests gibt es noch eine zusätzliche Möglichkeit zur Filterung der gewünschten Knotenmenge. Dabei setzt man auf die sogenannten Prädikate. Hierbei wird durch einen logischen Ausdruck, der in eckigen Klammern steht, eine Bedingung formuliert. Diese Bedingung muss erfüllt sein, damit bestimmte Knoten ausgewählt werden. Wenn man mehrere Prädikate verwendet, bildet die Knotenmenge, die das Ergebnis des ersten Prädikats ist, den Ausgangspunkt für die Prüfung durch das zweite Prädikat usw.

Als Prädikat kann jeder gültige XPath-Ausdruck verwendet werden. Das Ergebnis eines Prädikats ist immer ein boolescher Wert.

Prädikatausdrücke unterstützen die folgenden Operatoren:

- `<` und `>`
- `<=` und `>=`
- `=` und `!=`

Beachten Sie, dass Operatoren wie `<` oder `>` nicht unmittelbar innerhalb des XML-Dokuments stehen dürfen, sondern durch die Entitätenreferenzen `<` bzw. `>` ersetzt werden müssen.

Neben den genannten Operatoren lassen sich auch mathematische einsetzen. Dadurch wird das Resultat des Ausdrucks von einem String in einen Zahlenwert umgewandelt.

- `div`
- `mod`
- `+`
- `-`
- `*`

Ebenso können auch Klammern verwendet werden.

5.1.6 Verkürzte Syntaxformen verwenden

Bei Ihrer täglichen Arbeit mit XPath kommt einiges an Tipparbeit auf Sie zu. Damit die nicht überhand nimmt, hält XPath einige interessante Syntaxformen bereit, mit denen man sich die Arbeit erleichtern kann.

Wollen Sie beispielsweise auf die Kindelemente des aktuellen Knotens zugreifen, sollte nicht jedes Mal die lange Syntax

```
child::knoten
```

verwendet werden. Bei der verkürzten Syntax lassen Sie einfach `child::` weg. Stattdessen beginnen Sie die Syntax mit dem Knotennamen.

`knoten`

Bei der Auswahl von Attributen kann ebenfalls eine verkürzte Notation verwendet werden. Anstelle von

```
attribute::name
```

kann man das `@`-Zeichen einsetzen.

```
autor[@land
```

Das Zeichen `@` steht dabei als Synonym für das Wort `attribute`.

Elternknoten werden normalerweise mit

```
parent::node()
```

gekennzeichnet. Auch diese Syntax lässt sich verkürzen. Dazu setzt man zwei Punkte ein.

```
../Knoten
```

Diese Syntax bewirkt das Gleiche wie `parent::node()`, ist aber um einiges kürzer.

Durch diese folgende Syntax lässt sich der komplette Teilbaum mit dem aktuellen Kontextknoten als Wurzel referenzieren.

```
/desccendant-or-self::node()
```

Anstelle dieser langen Variante kann man aber auch auf die verkürzte Schreibweise mit zwei Schrägstrichen zurückgreifen. So sorgt die folgende Syntax dafür, dass innerhalb eines HTML-Dokuments alle `a`-Elemente ausgewählt werden, die ein `href`-Attribut besitzen.

```
//a[@href]
```

Überall dort, wo es möglich ist, sollten Sie auch tatsächlich die verkürzte Notation verwenden. Denn so lassen sich die Dokumente nicht nur schneller schreiben, sie sind für erfahrene Entwickler oft sogar besser zu lesen.[1]

5.1.6.1 Operatoren für echtes Programmier-Feeling

Genauso wie in „echten“ Programmiersprachen gibt es auch in XPath Operatoren. Bevor die einzelnen Operatoren vorgestellt werden, einige Worte zu den Datentypen. Die folgenden stehen in XPath zur Verfügung:

- Zahlen
- Text
- Knoten und Knotensammlungen
- Boolesche Werte `true` (wahr) und `false` (falsch)

Bevor die Operatoren im Einzelnen vorgestellt werden, zunächst ein Beispiel, wie man sie einsetzt. Gezeigt wird das anhand der beiden logischen Operatoren `or` und `and`.

```
autor[(buch or zeitschrift) and publiziert]
```

[1] Einsteiger tun sich erfahrungsgemäß allerdings mit der verkürzten Syntax etwas schwer.

Die Angabe bezieht sich auf alle `autor`-Elemente mit mindestens einem `buch`- oder `zeitschrift`-Element und mindestens einem `publiziert`-Element.

Bei einigen Parsern müssen die beiden Operatoren `<` und `>` mit ihren Entitäten angegeben werden. Dort muss man anstelle von `<` die Zeichenfolge `<` und anstelle von `>` die Zeichenfolge `>` notieren.

Tabelle 5.2 liefert eine Übersicht der in XPath vorhandenen Datentypen. Sortiert sind diese nach ihren jeweiligen Einsatzgebieten.

Tabelle 5.2: Alle Operatoren im Überblick

Operator	Beschreibung
Arithmetische Operatoren	
`*`	Multiplikation
`+`	Addition
`-`	Subtraktion
`mod`	Gibt den Rest einer ganzzahligen Division zurück.
`div`	Führt eine Gleitkommadivision nach IEEE 754 aus.
Vergleichsoperatoren	
`=`	Vergleich zweier Werte
`!=`	Prüft zwei Werte auf Ungleichheit
`<` bzw. `<`	Prüft, ob der erste Wert kleiner als der zweite ist.
`>` bzw. `>`	Prüft, ob der erste Wert größer als der zweite ist.
`<=` bzw. `<=`	Prüft, ob der erste Wert kleiner oder gleich dem zweiten ist.
`>=` bzw. `>=`	Prüft, ob der erste Wert größer oder gleich dem zweiten ist.
Logische Operatoren	
`and`	Zwei Ausdrücke werden mit UND (beide Ausdrücke) verknüpft.
`or`	Zwei Ausdrücke werden mit ODER (einer von beiden Ausdrücken) verknüpft.
Sonstige	
`()`	Klammern kennzeichnen einen logischen Zusammenhang. Sie werden oft in mathematischen Berechnungen oder bei logischen Verknüpfungen eingesetzt.
`|`	Zwei Knoten oder Knotenmengen werden zu einer Knotenmenge verbunden.
`/`	Verbindet einen Ausdruck zu einem relativen Pfad.

5.1.7 Variablen einsetzen

Neben den zuvor gezeigten Operatoren gibt es in XPath noch ein weiteres Element, das Sie so vielleicht auch aus anderen Programmiersprachen kennen: Variablen. Nun lassen sich in XPath selbst keine Variablen deklarieren. Das macht man hingegen mittels XSLT. Wie dieses Deklarieren funktioniert, wird später noch ausführlich beschrieben. An dieser Stelle geht es vielmehr darum, wie mittels XPath auf den Wert einer Variablen zugegriffen werden kann. Auch hierzu wieder ein Beispiel. Angenommen, innerhalb einer XSLT-Datei wurde die folgende Variable deklariert:

Listing 5.13 So einfach lassen sich Variablen deklarieren.

```
<xsl:variable name="aname">
   Nick Hornby
</xsl:variable>
```

Mittels XPath können Sie auf die Variable zugreifen. Dazu müssen Sie lediglich ein Dollarzeichen vor den Variablennamen setzen. Durch

```
$aname
```

wird auf die Variable `aname` zugegriffen und der Wert `Nick Hornby` ermittelt.

5.1.8 Auch Funktionen gibt es

Lauf der W3C-Empfehlung müssen XPath-Prozessoren auch eine Bibliothek an Funktionen unterstützen. Über diese Funktionen lässt sich die Transformation der XML-Ausgangsdaten in den Ergebnisbaum steuern. Ebenso interessant sind diese Funktionen aber auch für die Formulierung ausdrucksstarker Prädikate.

Gedacht sind die Funktionen für den Einsatz innerhalb von XSLT-Stylesheets.

Einteilen lassen sich die Funktionen entsprechend den von XPath-Ausdrücken gelieferten Datentypen in vier Gruppen.

- `node-set`
- String
- Boolean
- numerisch

Die Syntax der einzelnen Funktionen ist immer gleich. Hinter dem Funktionsnamen stehen zwei Klammern. Innerhalb dieser Klammern können dann entsprechende Argumente angegeben werden.

Bevor die einzelnen Funktionen vorgestellt werden, als Einstieg ein typisches Beispiel.

Listing 5.14 Ein Beispiel für den Anfang

```
<xsl:template match="autor:*">
   <div>
      <p>Name des Knotens:
         <b>
            <xsl:value-of select="name(.)" />
```

```
            </b>
         </p>
         <p>local-name:
            <b>
               <xsl:value-of select="local-name(.)" />
            </b>
         </p>
         <p>Inhalt:<b>
               <xsl:apply-templates />
            </b>
         </p>

      </div>
   </xsl:template>
```

In dieser Syntax kommen die beiden Funktionen `name()` und `local-name()` zum Einsatz.

- `name()` – Ermittelt den Namen des Knotensets.
- `local-name()` – Liest den lokalen Namen des Knotensets aus.

Viele XPath-Funktionen erwarten Argumente. Die möglichen Argumente werden in den folgenden Tabellen innerhalb der Klammern hinter dem Funktionsnamen aufgeführt. Ein Beispiel dazu:

```
count(Knotenmenge)
```

Diese Angabe bedeutet, dass die Funktion `count()` als Argument eine Knotenmenge erwartet. Praktisch umgesetzt sähe das dann folgendermaßen aus:

```
count(//autor)
```

Die Funktionen liefern einen Wert zurück. Daher werden sie in XSLT hauptsächlich dort verwendet, wo Attributen Werte zugewiesen werden. Das XSLT-Element `value-of` könnte somit folgendermaßen notiert werden:

```
<xsl:value-of select="last()" />
```

Durch diese Syntax wird dem `select`-Attribut ein Wert zugewiesen. Bei diesem handelt es sich allerdings nicht um einen festen, sondern um einen dynamisch ermittelten Wert. Konkret wird hier durch `last()` die Positionsnummer des letzten Knotens ermittelt.

An dieser Stelle darf der Hinweis nicht fehlen, dass nicht jeder XSLT-Prozessor alle Funktionen unterstützt. Sie müssen die gewünschten Funktionen also vor dem Einsatz in „Ihrem“ Prozessor testen.

5.1.8.1 Funktionen für Knotenmengen

Am meisten hat man es normalerweise mit den Knotenmengenfunktionen zu tun. Diese beziehen sich auf Knotenmengen, und zwar entweder auf die aktuelle oder auf eine explizit angegebene Knotenmenge.

Über

```
count(//autor)
```

könnte man z.B. die Anzahl der verschiedenen Autoren ermitteln.

Durch eine Kombination der beiden Funktionen `last()` und `position()` lässt sich der letzte Knoten innerhalb einer Knotenmenge auswählen.

```
child::autor[position() = last()]
```

Ebenso einfach kann man alle Elementknoten ermitteln, die einem ganz bestimmten Namensraum angehören.

```
/descendent-or-self::*[namespace-uri() = "*"]
```

Und nun die möglichen Knotenmengenfunktionen:

Tabelle 5.3: Die Knotenfunktionen

Funktion	Rückgabewert	Beschreibung
`count(Knotenmenge)`	Zahl	Ermittelt die Anzahl der auf der Ebene unterhalb eines Knotensets enthaltenen Knoten.
`id(Objekt)`	Knotenmenge	Ergibt die Menge aller Knoten, die die Elemente des Dokuments enthält, deren Attribut in einer DTD als Attributtyp `ID` definiert wurden.
`last()`	Zahl	Ergibt die Anzahl der Knoten in der Kontextknotenliste.
`local-name(Knotenmenge)`	String	Aus einem Knotennamen mit Namensraumangabe wird der lokale Namensteil ermittelt. Aus `xhtml:cite` wird so `cite`.
`name(Knotenmenge)`	String	Ermittelt den vollständigen Namen eines Knotensets, ggf. mit Angabe des XML-Namensraums.
`namespace-uri(Knotenmenge)`	String	Ermittelt den URI für die DTD zum verwendeten Namensraum.

5.1.8.2 Funktionen für Zahlen

Es gibt auch einige numerische Funktionen. Sehr oft kommt z.B. die `number()`-Funktion zum Einsatz, mit der sich ein Objekt in eine Zahl umwandeln lässt. Bei dieser Umwandlung liefern logische Objekte die Werte `0` für falsch oder `1` für wahr. String-Werte liefern eine Zahl, wenn sie als Zahl ausgewertet werden können.

Interessant ist auch die `sum()`-Funktion. Mit der lässt sich die Summe der Werte einer Knotenliste berechnen. So liefert

```
sum(//gesamt)
```

die Summe aller Werte des `gesamt`-Elements.

Tabelle 5.4: Die Funktionen für Zahlen

Funktion	Rückgabewert	Beschreibung
`ceiling(Zahl)`	Zahl	Angewandt auf `X`, liefert die Funktion die kleinste Zahl, die größer oder gleich `X` ist.
`floor(Zahl)`	Zahl	Rundet eine Bruchzahl zur basierenden Ganzzahl ab.

Funktion	Rückgabewert	Beschreibung
`number(Objekt)`	Zahl	Konvertiert ein Objekt in eine Zahl. Strings, die mit einer Zahl beginnen, werden in diese Zahl konvertiert. Andere Strings werden in `NaN` (Not a Number) konvertiert. `true` wird in `1`, `false` in `0` umgewandelt. Bei Knotenmengen wird zunächst der String-Wert des ersten Knotens der Menge ermittelt und dieser dann in eine Zahl umgewandelt.
`round(Zahl)`	Zahl	Rundet eine Bruchzahl zur nächstgelegenen Ganzzahl auf.
`sum(Knotenmenge)`	Zahl	Ermittelt die Gesamtsumme der Zahlenwerte des Ausgangsknotens.

5.1.8.3 Logische Funktionen

Auch eine Anzahl von logischen Funktionen hat XPath zu bieten. Am häufigsten wird dabei sicherlich die `boolean()`-Funktion eingesetzt. Mit der kann man testen, ob eine Knotenmenge tatsächlich Knoten enthält oder leer ist. Ebenso lässt sich anhand dieser Funktion überprüfen, ob ein String-Wert eine Zeichenkette liefert oder leer ist.

Tabelle 5.5: Die logischen Funktionen

Funktion	Rückgabewert	Beschreibung
`boolean(Objekt)`	Boolean	Ermittelt, ob ein Ausdruck wahr (`true`) oder falsch (`false`) ist. Eine Zahl ist immer wahr, wenn sie weder `0` noch `NaN` ist. Eine Knotenmenge ist immer wahr, wenn sie nicht leer ist. Ein String ist immer wahr, wenn er nicht leer ist.
`false()`	Boolean	Verneint einen Ausdruck.
`lang(String)`	Boolean	Kontrolliert, ob in einem Element ein bestimmter Sprachcode verwendet wird. Setzt voraus, dass im Element das XML-Attribut `xml:lang` definiert ist.
`not(Boolean)`	Boolean	Gibt den Wert aus, der ungleich dem Parameter ist. Für `true` wird `false`, für `false` wird `true` ausgegeben.
`true()`	Boolean	Bejaht einen Ausdruck.

5.1.8.4 Funktionen für Strings

In XPath gibt es einige String-Funktionen. So kann man z.B. mittels der `string`-Funktion ein Objekt in eine Zeichenkette umwandeln.

Einige der String-Funktionen werden zum Zerlegen und Verketten von Zeichenketten innerhalb von Lokalisierungspfaden verwendet. Auf diese Weise lassen sich Knotenmengen anhand von String-Werten filtern. Wollen Sie beispielsweise alle Autoren markieren, deren Name mit `Ho` anfängt, verwenden Sie die folgende Syntax:

```
//child::bibliothek[starts-with(name, "Ho")]
```

Interessant ist auch die Funktion `translate`. Mit der kann man zeichenweise Ersetzungen innerhalb von Zeichenketten durchführen. So ergibt

```
translate("openxml", "oxml", "OXML")
```

die folgende Zeichenkette:

```
OpenXML
```

Und hier alle String-Funktionen in der Übersicht:

Tabelle 5.6: Alle String-Funktionen in der Übersicht

Funktion	Rückgabewert	Beschreibung
`concat(String, String...)`	String	Hängt mehrere Zeichenketten aneinander.
`contains(String, String...)`	Boolean	Überprüft, ob in einer Zeichenkette eine bestimmte Teilzeichenkette enthalten ist.
`normalize-space (String)`	String	Führende oder abschließende Leerzeichen werden entfernt. Mehrere aufeinanderfolgende Leerzeichen ersetzt die Funktion durch ein Leerzeichen. Das bedeutet, dass zum Beispiel auch Zeilenumbrüche durch Leerzeichen ersetzt werden.
`starts-with(String, String)`	Boolean	Ermittelt, ob am Anfang einer Zeichenkette eine bestimmte Teilzeichenkette enthalten ist.
`string(Objekt)`	String	Wandelt das übergebene Objekt in einen String um. Handelt es sich um einen Knoten, wird der Textwert des Knotens ausgegeben. Bei einer Zahl wird die Zahl als Text ausgegeben. Ungültige Zahlen liefern dabei `NaN`, positive unendliche liefern `Infinity` und negative unendliche liefern `–Infinity`. Boolesche Werte werden mit `true` und `false` ausgegeben.
`string-length(String)`	Zahl	Ermittelt die Länge der Zeichenkette.
`substring(String, Zahl, String)`	String	Aus einer Zeichenkette wird ein Teil ab einer bestimmten Zeichenposition extrahiert. Zusätzlich kann die zu extrahierende Zeichenanzahl angegeben werden.
`substring-after(String, String)`	String	Liefert den Teil des ersten Strings, der sich nach der Position des zweiten Strings befindet. So gibt `substring-after("abcdef", "ab")` den Wert `cdef` aus.

Funktion	Rückgabewert	Beschreibung
`substring-before(String, String)`	String	Gibt den ersten Teil des Strings aus, der sich vor der Position des zweiten Strings befindet. So gibt `substring-before("abcdef", "cd")` den Wert `ab` aus.
`translate(String, String, String)`	String	Ersetzt alle Zeichen des zweiten Strings, die sich im ersten String befinden, durch die Zeichen des dritten Strings, die die gleiche Position besitzen, und gibt diesen aus. So gibt `translate("abcdef","ace", "123")` den Wert `"1b2d3f"` aus.

5.2 Neuerungen in XPath 2.0

Bereits seit Januar 2007 gibt es eine neue Empfehlung für XPath.

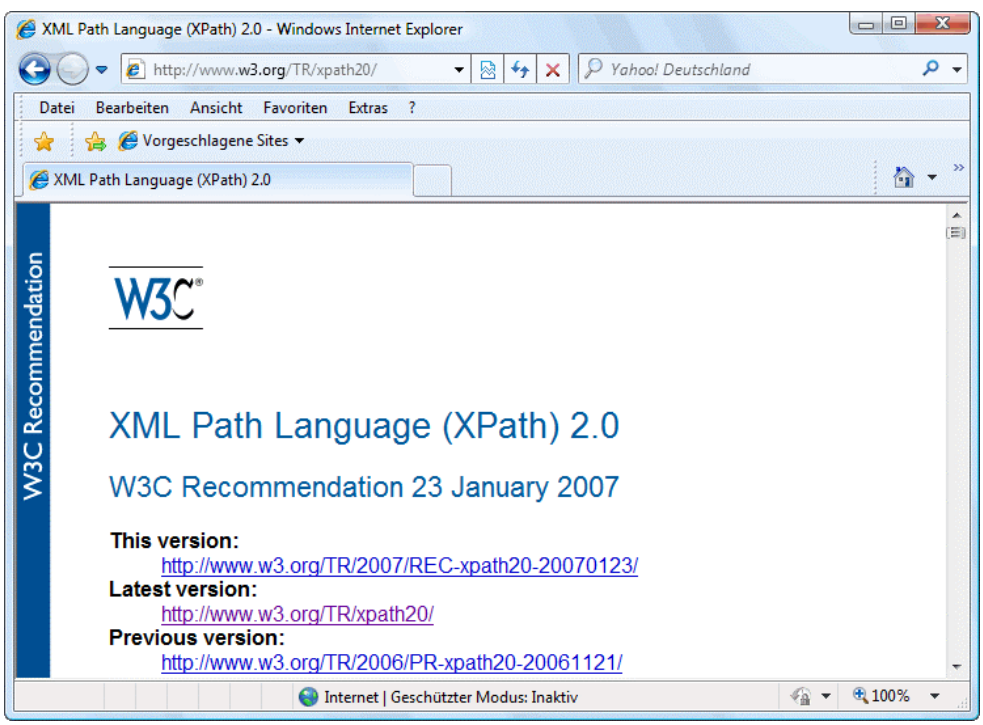

Abbildung 5.7 Die Spezifikation zu XPath 2.0

Unter *http://www.w3.org/TR/xpath20/* finden Sie die Empfehlung zu XPath 2.0. Auf den folgenden Seiten werden die wichtigsten Neuerungen vorgestellt, die dieser neue Standard zu bieten hat. An dieser Stelle allerdings gleich der Hinweis, dass längst noch nicht alle

Prozessoren die neuen Features umsetzen können, die XPath 2.0 mitbringt. Sie sollten daher im Vorfeld prüfen, ob „Ihr“ Prozessor in der Lage ist, die von Ihnen verwendeten Features umzusetzen.

XPath 2.0 wird von XSLT 2.0 und XQuery 1.0 gleichermaßen verwendet. Alle drei Empfehlungen wurden gleichzeitig veröffentlicht. XQuery soll vornehmlich als Standard für die Abfrage von XML-Dokumenten verwendet werden. XSLT ist hingegen hauptsächlich für das Transformieren von XML-Daten da.

Bei XQuery 1.0 handelt es sich eigentlich um eine Erweiterung von XPath 2.0. Aufgrund der Mächtigkeit von XQuery gehen viele Experten mittlerweile davon aus, dass diese Sprache in Zukunft die zentrale Abfragesprache werden könnte.

5.2.1 Die Rückwärtskompatibilität zu XPath 1.0

Bevor man sich mit XPath 2.0 befasst, will man natürlich wissen, wie sich das Neue eigentlich mit dem Alten verträgt. Um es vorwegzunehmen: Aufgrund des neuen Datenmodells und der zusätzlichen Sprachmittel gibt es zwischen XPath 1.0 und XPath 2.0 Inkompatibilitäten. Damit sich diese nicht allzu sehr auswirken, hat XPath einen sogenannten Kompatibilitätsmodus integriert, der allerdings auch nicht alle Unterschiede ausgleichen kann.

Probleme gibt es bei den Ergebnissen von XPath-Ausdrücken vor allem dann, wenn eine Schemaspezifikation der Auswertung zugrunde liegt.

- XPath 1.0 – Alle Attribut- und Textknoten werden für Vergleiche in Strings umgewandelt.
- XPath 2.0 – Vergleiche werden hier unter Berücksichtigung der Schema-Datentypen durchgeführt.

Darüber hinaus gibt es auch syntaktische Unterschiede zwischen beiden Versionen:

- XPath 1.0 – Werte mit positivem Vorzeichen, Gleitkommawerte sowie positiv und negativ unendliche Werte werden auf den nicht numerischen Wert `NAN` abgebildet.
- XPath 2.0 – Werte mit positivem Vorzeichen, Gleitkommawerte sowie positiv und negativ unendliche Werte werden akzeptiert.

Und auch hinsichtlich der Klammerung von Teilausdrücken gibt es Unterschiede. Während sie in der 1.0er-Version noch optional war, ist sie nun Pflicht.

5.2.2 Das erweiterte Datenmodell

Die größten Unterschiede zwischen XPath 2.0 und XPath 1.0 gibt es zweifellos hinsichtlich des Datenmodells. Denn gegenüber seiner Vorgängerversion hat XPath 2.0 vor allem eine striktere Typisierung von vorgegebenen und mittels XML Schema importierten Datentypen. Neben dem bereits beschriebenen Baummodell, durch das die im XML-Dokument vorhandenen Informationseinheiten repräsentiert werden, werden in XPath 2.0

auch einzelne Werte (*Atomic Values*) berücksichtigt. Bei denen kann es sich um Daten ganz unterschiedlichen Typs handeln. Möglich sind:

- Zeit- und Datumswerte
- Qualifizierte Namen
- Zahlen
- Logische Werte
- Zeichenfolgen
- URIs

Darüber hinaus gibt es Sequenzen oder Listen. Diese können aus Einzelwerten sowie aus Bezügen auf Knoten in einem XML-Dokument bestehen. Dabei handelt es sich dann um einfache Sequenzen, bei denen keine Schachtelung möglich ist.

Wie bereits bei XPath 1.0 wird ein XML-Dokument beim Einlesen grundsätzlich in ein XML Information Set verwandelt. Dieses XML Information Set gibt keinerlei Informationen über die Dokumenttypdefinition oder ein entsprechendes Schema preis. Wenn allerdings eine Schema-Definition vorliegt, kann das Dokument anhand dieses Schemas validiert werden. Dabei erzeugt der validierende Parser das Post-Schema Validation Infoset (PSVI). Dieses PSVI enthält zusätzliche Informationen über das Dokument. Innerhalb von XPath 2.0 lassen sich diese Angaben für die Annotation des geparsten Dokumentbaums verwenden. Das führt dazu, dass ein solcher PSVI rückwärtskompatibel zu dem Dokumentbaum von XPath 1.0 ist. (Allerdings enthält ein PSVI zusätzliche Typinformationen für die Auswertung von Pfadausdrücken.) Liegen hingegen keine Typinformationen vor, werden Attribute und Elemente als ungetypt bezeichnet. Deren Verarbeitung erfolgt dann wie in XPath 1.0, mit dem Unterschied, dass die Syntax für typsichere Vergleiche verwendet wird.

5.2.2.1 Neue Datentypen

Das neue Datenmodell unterscheidet sich von der Vorgängerversion aber vor allem darin, dass nun eine striktere Typisierung von vorgegebenen und mittels XML Schema importierten Datentypen vorgegeben ist.

Interessant ist XPath 2.0 auch und gerade hinsichtlich der neuen Datentypen. Denn jetzt werden die durch XML Schema angebotenen Datentypen unterstützt. **Abbildung 5.8** zeigt eindrucksvoll die entsprechenden Erweiterungen.

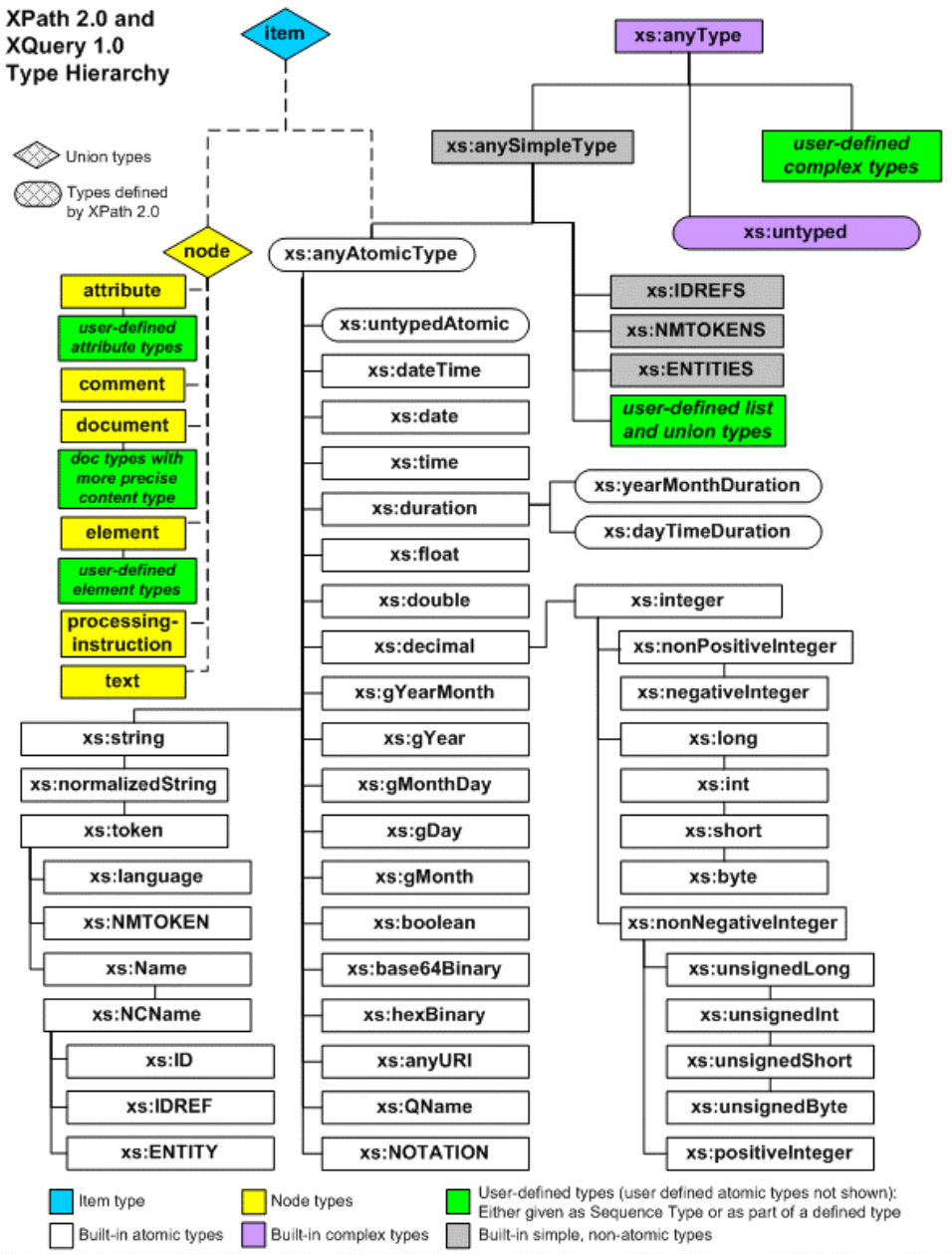

Abbildung 5.8 Quelle: *http://www.w3.org/TR/2007/REC-xpath-datamodel-20070123/#types*

Wenn Sie sich bereits mit XPath 1.0 befasst haben, wissen Sie, dass dort lediglich ein einziger numerischer Datentyp unterstützt wird. In XPath 2.0 wurden zusätzliche zur Verfügung gestellt:

- `decimal`
- `integer`
- `single-precision`

Darüber hinaus gibt es auch zahlreiche Datentypen für Datum- und Zeitangaben sowie für Zeichenketten. Schlussendlich wird die Palette der möglichen Datentypen dann auch noch um die Möglichkeit erweitert, benutzerdefinierte Datentypen einzusetzen.

5.2.2.2 Neue Operatoren

Aufgrund des neuen Datenmodells mussten auch neue Operatoren eingeführt werden. Genau dafür hat man bei XPath 2.0 gesorgt. Die folgende Liste bietet einen Überblick der wichtigsten neuen Operatoren:

- `is` – Dieser Operator überprüft, ob zwei Ausdrücke denselben Knoten liefern.
- `>>` – Überprüft, ob der erste Knoten früher in der Dokumentreihenfolge als der zweite Knoten steht.
- `<<` – Überprüft, ob der erste Knoten später in der Dokumentreihenfolge als der zweite Knoten steht.
- `intersect` – Aus zwei Sequenzen wird eine Sequenz erzeugt, in der die Knoten enthalten sind, die in beiden Sequenzen vorkommen.
- `union` – Es können zwei Knotensequenzen zu einer Sequenz zusammengefügt werden, die innerhalb der einen oder der anderen Sequenz stehen.
- `except` – Eine Sequenz wird aus zwei Knotensequenzen erzeugt, die nur solche Knoten enthält, die zwar in der ersten, nicht aber in der zweiten Sequenz vorkommen.
- `idiv` – Hiermit lassen sich Ganzzahlen dividieren.
- `?` – Dieser Operator bezeichnet das optionale Vorkommen des vorangehenden Terms.
- `*` – Ermöglicht das beliebig häufige Auftreten des Terms.
- `+` – Der Term muss mindestens einmal vorkommen.

So lassen sich z.B. alle Vorkommen von Publikationen in einem Dokument, mit Ausnahme von Groschenromanen, folgendermaßen beschreiben:

```
element(*,Publikationen) except element(*,Groschenroman)
```

5.2.3 Kommentare

XPath-Ausdrücke können nun auch mit Kommentaren ausgestattet werden. Kommentare beginnen mit der Zeichenfolge `(:` und enden auf `:)`. Zudem ist eine Verschachtelung von Kommentaren möglich. Dabei ist darauf zu achten, dass die Kommentarbegrenzer jeweils paarweise balanciert auftreten. Nur so kann der Parser die Begrenzer korrekt zuordnen.

Aus semantischer Sicht haben Kommentare keinerlei Bedeutung, da sie direkt beim Parsen entfernt werden.

```
for (: Es werden alle Autoren ausgelesen :)
...
```

5.2.4 Knotentests

Knotentests spielen in XPath eine entscheidende Rolle. Denn mit ihnen lassen sich die Eigenschaften von Knoten bestimmen. Gab es in XPath 1.0 bereits diverse Varianten, kamen in XPath 2.0 noch einmal einige hinzu. Mit den neuen Funktionen kann man nun auch auf die erweiterten Typangaben aus der Schemadefinition zugreifen.

In XPath 2.0 werden Typangaben innerhalb von Testausdrücken unterstützt. Vor allem wurden die Knotentests `ATTRIBUTE` und `ELEMENT` um einen Parameter erweitert. Dieser spezifiziert den gewünschten Datentyp. Dabei wird ein Knoten auch dann in die Ergebnismenge aufgenommen, wenn es sich um eine Spezialisierung des angegebenen Typs handelt.

Des Weiteren wurde mit `ITEM()` ein zusätzlicher Testausdruck eingefügt, über den Knotentypen und atomare Datentypen zusammengefasst werden. Im Gegensatz zu `NODE()` liefert `ITEM` neben Attribut- und Elementknoten oder zusätzlich definierten Knotentypen auch Zeichenketten und Zahlenwerte.

Mit

- `SCHEMA-ATTRIBUTE` und
- `SCHEMA-ELEMENT`

wurden zweite weitere Tests eingeführt. Beiden Testausdrücken wird als Parameter ein qualifizierter Name zugewiesen. Als Ergebnis liefern beide die Attribut- oder Elementknoten, die eine entsprechende Schema-Definition besitzen.

Interessant in XPath 2.0 ist auch die Möglichkeit, Sequenzen in Knotentests durch sogenannte Occurrence Indicators zu beschreiben. Das sind Indikatoren für das Auftreten bestimmter Knoten. Unterschieden wird dabei zwischen den drei folgenden Operatoren:

- `?` – Der vorangehende Term ist optional.
- `+` – Der Term muss mindestens einmal vorkommen.
- `*` – Der Term kann beliebig oft vorhanden sein.

Schlussendlich können Sie Sequenzen auch noch miteinander kombinieren. Dafür gibt es die folgenden Operatoren:

- `INTERSECT` = Durchschnitt
- `EXCEPT` = Differenz
- `UNION` = Vereinigung

Dieser Abschnitt hat gezeigt, dass XPath 2.0 auch auf dem Gebiet der Knotentests interessante Neuerungen mitbringt.

5.2.5 Schleifenausdrücke

XPath 2.0 bietet für den Zugriff auf geordnete Datensätze einen einfachen Schleifenausdruck. Vergleichbar ist der mit der SELECT-Klausel aus SQL. Die allgemeine Syntax für solche Schleifenausdrücke sieht folgendermaßen aus:

```
for $VAR in Wert return Wert
for $VAR1 in Wert, $VAR2 in Wert, ... return Wert
```

Jeder Schleifenausdruck wird – und auch hier gibt es Ähnlichkeiten zu anderen Programmiersprachen – mit `for` eingeleitet. Daran schließt sich die Bindung eines Bereichs an eine Variable an. Dabei wird der Laufvariablen ein Dollarzeichen vorangestellt. Die eigentliche Bindung an den Bereich erfolgt über das Schlüsselwort `in`, an das sich ein beliebiger XPath-Ausdruck anschließt. Dieser Ausdruck wiederum wird zu einer Sequenz ausgewertet.

Der eigentliche Schleifenköper setzt sich aus dem Schlüsselwort `return` und einem Ausdruck zusammen, der für jeden Schleifendurchlauf exakt einmal ausgewertet wird. Bei jedem dieser Schleifendurchläufe wird jedes Vorkommen einer Laufvariablen aus dem Schleifenkopf durch den aktuellen Wert dieser Variablen ersetzt. Als Ergebnis bekommt man eine Sequenz mit den Ergebnissen eines Schleifenköpers in der Reihenfolge des Auswertungsbereichs.

Zum besseren Verständnis dieser – zumindest aus theoretischer Sicht – nicht ganz einfachen Thematik ein kleines Beispiel:

```
for $zaehler in (1 to 10)[. mod 2 eq 1]
    return ($zaehler * $ zaehler)
```

In diesem Beispiel wird zunächst die Laufvariable `$zaehler` nacheinander an alle ungeraden Zahlen zwischen `1` und `10` gebunden.

```
1, 3, 5, 7, 9
```

Jeder Schleifenkörper wird für jeden dieser Werte exakt einmal evaluiert. Daher handelt es sich bei dem Ergebnis für diesen Schleifenausdruck für eine Sequenz aus den Quadraten der entsprechenden Ziffern.

```
1,9,25,49,81
```

Eignet sich diese Syntax hauptsächlich für einfache Schleifen, muss man bei komplexeren Ausdrücken mehrere Bereiche kombinieren. Dabei wird jede Laufvariable in einem Schleifendurchlauf an den aktuellen Wert aus der entsprechenden Sequenz gebunden und ist im Schleifenkörper verfügbar. Auch hierzu wieder ein Beispiel:

Listing 5.15 Eine einfache Schleife

```
for $bibliothek in buecher,
    $quelle in lager[artikel/@nr = $buch/@nummer]
  return ($buch, $quelle[artikel/@nr = $buch/@nummer]/name)
```

In diesem Beispiel werden aus einer angenommenen Datenbank Werte ausgelesen. Bei diesen Werten handelt es sich um eine Kombination aus Büchern und den Lagern, in denen

sie verfügbar sind. Das Ergebnis ist eine Sequenz aus den Büchern und den entsprechenden Lagern. Dabei ist diese Sequenz sehr flach.

Die gezeigte Syntax macht einen Aspekt deutlich: Mit Schleifenausdrücken lassen sich komplexe Objekte kaum erzeugen. Denn das Ergebnis ist jeweils eine Sequenz, die keinerlei Unterstruktur besitzt. Für das gezeigte Beispiel sähe das folgendermaßen aus:

Listing 5.16 So sieht das Ergebnis aus.

```
<buecher nummer="1223">One for my baby</buecher>
<name>Tony Parsons</name>
<buecher nummer="2342">About a boy</buecher>
<name>Nick Hornby</name>
```

5.2.6 Bedingungen definieren

Es besteht die Möglichkeit, wie Sie das auch von anderen Programmiersprachen her gewohnt sind, Bedingungen zu definieren. Bedingte Knotenausdrücke werden dabei auch in XPath über das Schlüsselwort `if` gebildet, an das sich ein boolescher Ausdruck anschließt. Die allgemeine Syntax sieht folgendermaßen aus:

Listing 5.17 Das ist die allgemeine Syntax.

```
if Bedingung
   then Bedingung
else Bedingung
```

Wird die Bedingung erfüllt, bestimmt der mit `then` eingeleitete Zweig das Ergebnis. Wenn sie nicht erfüllt werden kann, greift `else`. Beachten Sie, dass der `else`-Zweig immer angegeben werden muss.

Im folgenden Beispiel wird versucht, den Preis für ein Buch zu ermitteln:

Listing 5.18 Ein typisches Beispiel für eine Schleife.

```
if ($buch/preis)
     then $buch/Preis
else "Preis konnte nicht ermittelt werden"
```

Lässt sich der Preis ermitteln, wird er ausgegeben. Ist (noch) kein Preis abrufbar, wird `Preis konnte nicht ermittelt werden` ausgegeben.

5.2.7 Die erweiterte Funktionsbibliothek

Mit XPath 2.0 wurden zahlreiche zusätzliche Funktionen eingeführt. So gibt es nun endlich eine große Auswahl an Funktionen für die Behandlung von Zeichenketten. Die folgende Übersicht zeigt die entsprechenden Funktionen nach Gruppen geordnet und dient in erster Linie zum Nachschlagen.

An dieser Stelle darf nicht der Hinweis auf die offizielle XPath-Spezifikation fehlen, die unter *http://www.w3.org/TR/2007/REC-xpath-functions-20070123/* veröffentlicht wurde. Dort finden Sie alle XPath-Funktionen noch einmal übersichtlich aufgereiht.

5.2.7.1 Zugriffsfunktionen

Tabelle 5.7: Die Zugriffsfunktionen aus XPath 2.0

Funktion	Beschreibung
`fn:base-uri`	Liefert den Basis-URI des Arguments
`fn:data`	Liefert eine Sequenz atomarer Werte
`fn:document-uri`	Liefert den Dokument-URI des Arguments
`fn:nilled`	Liefert einen `xs:boolean`-Wert, der anzeigt, ob für den Knoten das Attribut `xs:nil="tru"` gilt
`fn:node-name`	Liefert den vollständigen `xs:QName`

5.2.7.2 Trace-Funktion

Tabelle 5.8: Die Trace-Funktionen aus XPath 2.0

Funktion	Beschreibung
`fn:trace`	Es wird die Ausführungsreihenfolge für das Debuggen geliefert.

5.2.7.3 Fehlerfunktion

Tabelle 5.9: Die Fehlerfunktionen aus XPath 2.0

Funktion	Beschreibung
`fn:error`	Ruft einen Fehler auf.

5.2.7.4 Numerische Werte

Tabelle 5.10: Die numerischen Funktionen aus XPath 2.0

Funktion	Beschreibung
`fn:abs`	Es wird der absolute Wert des Arguments geliefert.
`fn:celling`	Liefert die kleinste Ganzzahl, die größer oder gleich dem Argument ist.
`fn:floor`	Liefert die größte Ganzzahl, die größer oder gleich dem Argument ist.
`fn:round`	Rundet den Wert auf die nächstliegende Ganzzahl.
`fn:round-half-to-even`	Es wird die Zahl mit der angegebenen Präzision geliefert.

5.2.7.5 String-Funktionen

Tabelle 5.11: Die String-Funktionen aus XPath 2.0.

Funktion	Beschreibung
`fn:codepoint-equal`	Es wird `true` geliefert, wenn die beiden Argumente nach der Unicode-Kollation gleich sind.
`fn:codepoints-to-string`	Aus einer Sequenz von Unicode-Codes wird ein `xs:string` gebildet.
`fn:contains`	Hierüber wird angezeigt, ob ein `xs:string` in einem anderen enthalten ist.
`fn:concat`	Es werden zwei oder mehr `xs:anyAtomicType`-Argumente zu einem `xs:string` verknüpft.
`fn:compare`	Es wird entweder `-1`, `0` oder `1` geliefert. Was tatsächlich geliefert wird, hängt davon ab, ob der erste Wert kleiner, gleich oder größer als der zweite ist.
`fn:encode-for-uri`	Es wird das `xs:string`-Argument mit maskierten Zeichen geliefert. Somit kann das für den Einsatz innerhalb von URIs verwendet werden.
`fn:escape-html-uri`	Liefert das `xs:string`-Argument mit den entsprechend maskierten Zeichen. Dadurch kann der Ergebnisstring als URI bzw. als URI-Teil verwendet werden.
`fn:ends-with`	Zeigt an, ob der `xs:string`-Wert auf einen anderen `xs:string`-Wert endet.
`fn:iri-to-uri`	Es wird das `xs:string`-Argument mit maskierten Zeichen geliefert.
`fn:lower-case`	Gibt den Wert des Arguments in Kleinbuchstaben wieder.
`fn:normalize-space`	Es wird der `whitespace-normalized`-Wert des Strings geliefert.
`fn:normalize-unicode`	Liefert den normalisierten Wert. Die entsprechende Form wird über das zweite Argument bestimmt.
`fn:matches`	Es wird angezeigt, ob ein Wert zu dem über das zweite Argument angegebenen regulären Ausdruck passt.
`fn:replace`	Es wird der Wert des ersten Arguments geliefert. Dabei wird jeder Teilstring, der zu dem regulären Ausdruck des zweiten Arguments passt, mittels des im dritten String angegebenen Strings ersetzt.
`fn:string-to-codepoints`	Liefert die Sequenz der Unicode-Codes, die dem `xs:string` entsprechen.
`fn:string-join`	Durch die Verknüpfung einer Sequenz von Zeichenfolgen wird ein `xs:string` geliefert. Dabei kann auch ein Separator verwendet werden.
`fn:substring`	Es wird der `xs:string` geliefert, der an der angegebenen Position im Argument zu finden ist.

Funktion	Beschreibung
fn:string-length	Hierüber wird die Länge des Arguments geliefert.
fn:starts-with	Zeigt an, ob der xs:string-Wert mit einem anderen xs:string-Wert beginnt.
fn:substring-after	Es wird der nachfolgende Teilstring geliefert.
fn:substring-before	Es wird der vorhergehende Teilstring geliefert.
fn:translate	Es werden die im ersten Argument angegebenen Zeichen durch die an der betreffenden Stelle im zweiten Argument stehenden Zeichen ersetzt.
fn:tokenize	Es wird eine Sequenz von Strings geliefert, deren Werte Teilstrings des ersten Arguments sind. Die Strings sind jeweils durch Separatoren getrennt, die zu dem regulären Ausdruck im zweiten Argument passen.
fn:upper-case	Gibt den Wert des Arguments in Großbuchstaben wieder.

5.2.7.6 QName-Funktionen

Tabelle 5.12: Die QName-Funktionen aus XPath 2.0.

Funktion	Beschreibung
fn:in-scope-prefixes	Liefert für das gegebene Element das Präfix.
fn:local-name-from-QName	Liefert den xs:NCName mit dem lokalen Namen.
fn:Qname	Es wird der xs:Qname mit dem Namensraum des ersten Elements geliefert. Zusätzlich bekommt man den lokalen Namen und das Präfix des zweiten Arguments.
fn:resolve-QName	Es wird der xs:QName geliefert.
fn:prefix-from-QName	Für das angegebene Präfix wird der xs:NCName geliefert.
fn:namespace-uri-for-prefix	Für das Element wird der URI des Namensraums geliefert.

5.2.7.7 Knotenfunktionen

Tabelle 5.13: Die Knotenfunktionen aus XPath 2.0

Funktion	Beschreibung
fn:lang	Es wird angegeben, ob die Sprache des Knotens mit dem Wert von xml:lang der angegebenen Sprache übereinstimmt.
fn:local-name	Es wird der lokale Name des betreffenden Knotens als xs:NCName geliefert.
fn:name	Liefert den Knotennamen als xs:string.

Funktion	Beschreibung
fn:namespace-uri	Es wird der Namensraum-URI des Knotens als xs:anyURI geliefert.
fn:number	Liefert den Wert des Arguments.
fn:root	Es wird der Wurzelknoten geliefert.

5.2.7.8 Sequenzfunktionen

Tabelle 5.14: Die Sequenz-Funktionen aus XPath 2.0.

Funktion	Beschreibung
fn:avg	Der Durchschnitt einer Wertesequenz wird geliefert.
fn: boolean	Der logische Wert der Sequenz wird geliefert.
fn:count	Es wird die Zahl der Datenelemente einer Sequenz geliefert.
fn:collection	Über den angegebenen URI oder die Knoten der vorgegebenen Kollektion wird eine Knotensequenz geliefert.
fn:deep-equal	Wenn die beiden Argumente Datenelemente enthalten, die an den entsprechenden Positionen übereinstimmen, wird true geliefert.
fn:doc	Es wird ein document-Knoten über den betreffenden URI geliefert.
fn:doc-available	Gibt true zurück, wenn der document-Knoten über den URI gefunden werden kann.
fn:distinct-values	Sollten in einer Sequenz Duplikate enthalten sein, werden diese entfernt.
fn:empty	Zeigt, ob die betreffende Sequenz leer ist.
fn:exists	Zeigt, ob die betreffende Sequenz nicht leer ist.
fn:exactly-one	Wenn die Sequenz genau ein Datenelement enthält, wird sie geliefert.
fn:id	Es wird die Sequenz der Elementknoten geliefert, die einen ID-Wert besitzen, der zu einem IDREF-Wert passt.
fn:idref	Es wird die Sequenz der Element- oder Attributknoten geliefert, die einen ID-Wert besitzen, der zu einem IDREF-Wert passt.
fn:index-of	Es wird eine Sequenz von Integer-Werten geliefert. Jeder dieser Wert ist der Index eines Mitglieds der Sequenz des ersten Arguments, das mit dem Wert des zweiten Arguments übereinstimmt.
fn:insert-before	Es werden an der gleichen Stelle einer Sequenz entweder eine Sequenz oder ein Datenelement eingefügt.
fn:max	Liefert den höchsten Wert einer Wertesequenz.
fn:min	Liefert den niedrigsten Wert einer Wertesequenz.
fn:one-or-more	Wenn die Sequenz eins ist oder ein Datenelement enthält, wird sie geliefert.
fn:remove	An der angegebenen Position einer Sequenz wird ein Datenelement entfernt.

Funktion	Beschreibung
fn:reverse	Es wird die Reihenfolge der Datenelemente innerhalb einer Sequenz umgekehrt.
fn:subsequence	Liefert die Teilsequenz einer angegebenen Sequenz. Identifiziert wird die über ihre Position.
fn:sum	Liefert die Summe einer Wertesequenz.
fn: unordered	Je nach Art der Implementierung werden die Datenelemente in einer bestimmten Reihenfolge geliefert.
fn:zero-or-none	Wenn die Sequenz null ist oder ein Datenelement enthält, wird sie geliefert.

5.2.7.9 URI-Auflösung

Tabelle 5.15: Die Funktionen zur URI-Auflösung aus XPath 2.0

Funktion	Beschreibung
fn:resolve-uri	Liefert einen absoluten URI aus einem Basis-URI und einem relativen URI.

5.2.7.10 Logische Funktionen

Tabelle 5.16: Die logischen Funktionen aus XPath 2.0

Funktion	Beschreibung
fn:false	Liefert den xs:boolean-Wert false
fn:true	Liefert den xs:boolean-Wert true
fn:not	Kehrt den xs:boolean-Wert um

5.2.7.11 Uhrzeit, Datum und Zeitzonen

Tabelle 5.17: Funktionen für Uhrzeit, Datum und Zeitzonen.

Funktion	Beschreibung
fn:adjust-date-to-timezone	Passt den Wert xs:Date an die entsprechende Zeitzone an.
fn:adjust-dateTime-to-timezone	Passt den Wert xd:dateTime an die entsprechende Zeitzone an
fn:seconds-from-duration	Liefert die Sekunden von xs:duration.
fn:minutes-from-duration	Liefert die Minuten von xs:duration.
fn:hours-from-duration	Liefert die Stunden von xs:duration.

Funktion	Beschreibung
fn:days-from-duration	Liefert die Tage von xs:duration.
fn:month-from-duration	Liefert die Monate von xs:duration.
fn:years-from-duration	Liefert die Jahre von xs:duration.
fn:seconds-from-dateTime	Liefert die Sekunden von xs:dateTime.
fn:minutes-from-dateTime	Liefert die Minuten von xs:dateTime.
fn:hours-from-dateTime	Liefert die Stunden von xs:dateTime.
fn:day-from-dateTime	Liefert die Tage von xs:dateTime.
fn:month-from-dateTime	Liefert die Monate von xs:dateTime.
fn:year-from-dateTime	Liefert die Jahre von xs:dateTime.
fn:timezone-from-dateTime	Liefert die Zeitzone von xs:dateTime.
fn:day-from-date	Liefert die Tage von xs:date.
fn:month-from-date	Liefert die Monate von xs:date.
fn:year-from-date	Liefert die Jahre von xs:date.
fn:timezone-from-date	Liefert die Zeitzone von xs:date.
fn:seconds-from-time	Liefert die Sekunden von xs:time.
fn:minutes-from-time	Liefert die Minuten von xs:time.
fn:hours-from-time	Liefert die Stunden von xs:time.
fn:timezone-from-time	Liefert die Zeitzone von xs:time.

5.2.7.12 Dynamische Inhalte

Tabelle 5.18: Funktionen für dynamische Inhalte

Funktion	Beschreibung
fn:current-date	Liefert das aktuelle xs:date
fn:current-dateTime	Liefert das aktuelle xs:dateTime
fn:current-time	Liefert die aktuelle xs:time
fn:default-collation	Liefert den Wert der vordefinierten collation-Eigenschaft des statischen Kontextes.
fn:implicit-timezone	Liefert die aktuelle Zeitzone
fn:last	Es wird von der aktuell verarbeiteten Sequenz die Zahl der Datenelemente geliefert.
fn:position	Es wird die Position des Kontext-Datenelements in der Sequenz geliefert.
fn:static-base-uri	Es wird der Wert der Eigenschaft BASE URI des statischen Kontextes geliefert.

5.3 XPointer – mit Zeigern arbeiten

Bei XPointer handelt es sich um eine Nicht-XML-Syntax, mit der sich einzelne Punkte oder Bereiche innerhalb von XML-Dokumenten ansprechen lassen. Auch wenn XPointer recht unbekannt ist, wird es doch bereits in den unterschiedlichsten Bereichen eingesetzt. So findet man XPointer z.B. in XLink und SOAP.

Erfreulich an der XPointer-Spezifikation, die vom W3C im März 2003 veröffentlicht wurde, ist ihre Kürze. XPointer lässt sich somit also verhältnismäßig leicht erlernen.

Bei XPointer handelt es sich um eine Erweiterung von XPath, mit der sich URI-Referenzen auf XML-Dokumente mit Zeigern und Fragmenten des betreffenden Dokuments ausstatten lassen. Und eben diese Zeiger laufen unter der Bezeichnung XPointer.

5.3.1 URIs und Fragmentbezeichner

XPath haben Sie bereits kennengelernt. Dabei ist deutlich geworden, dass diese Sprache dazu dient, auf bestimmte Knoten und Knotenmengen innerhalb eines Dokuments zuzugreifen. Bei XPointer sieht das etwas anders aus. Denn XPointer führt über den Rand des einzelnen Dokuments hinaus und ermöglicht es vor allem, Teile eines XML-Dokuments als Verweisziele oder für andere Zwecke auszuwählen. So können Sie mittels XPointer z.B. Verknüpfungen mit XLink exakt auf die benötigten Informationen zuschneiden.

Aus HTML ist Ihnen sicherlich Folgendes bekannt:

```
http://www.hanser.de/news.html#books
```

Auch in HTML können bei einem URI hinter der Referenz, die auf das gesamte Dokument verweist, sogenannte Fragmentbezeichner eingefügt werden. Hier dient das Rautezeichen als Trennsymbol. In HTML lässt sich so etwas allerdings nur umsetzen, wenn innerhalb des Zieldokuments tatsächlich ein entsprechender Anker definiert wurde.

```
<h2><a name="kapitel1" href="#kapitel1">Kapitel 1</a></h2>
```

Genau das ist einer der Vorteile von XPointer. Denn bei dieser Sprache muss das Zieldokument nicht extra präpariert werden. Stattdessen lassen sich solche Fragmentbezeichner aus XPath-Ausdrücken aufbauen, durch die der Zugriff auf die gewünschten Informationen möglich wird.

Dabei lassen sich XPointer für alle Ressourcen anwenden, die einen der folgenden Medientypen besitzen:

- *text/xml*
- *allication/xml*
- *application/xml-external-parsed-entity*
- *xml-external-parsed-entity*

5.3.2 Die XPointer-Syntax

Damit XPointer korrekt eingesetzt werden kann, muss das Dokument, in dem man XPointer verwenden will, XML-konform sein. Als XPointer-Ausdruck kann entweder ein Shorthand Pointer oder schemabasierter Pointer verwendet werden, der sich auch aus mehreren sogenannter Pointer Parts zusammensetzen kann.

Bei der einteiligen Shorthand-Variante eines Pointers wird der Name des Elements, auf das zugegriffen werden soll, direkt verwendet. Als Name dient dabei der Name, der einem Element über ein Attribut oder eine ID zugewiesen wurde. Anstelle von

```
xpointer(*id("news"))
```

können Sie also auch

```
news
```

angeben.

Die sogenannten Pointer Parts schemabasierter Pointer werden im Gegensatz dazu mit einer Schemabezeichnung eingeleitet. Daran schließt sich ein in Klammern gesetzter Ausdruck mit Daten an, die dem Schema entsprechen. Derzeit gibt es die beiden Schemas `element()` und `xmlns()`. Mehr dazu auf den folgenden Seiten. Aber auch wenn bislang lediglich diese beiden Schemas vom W3C spezifiziert wurden, sind damit die Möglichkeiten noch nicht ausgeschöpft. Vielmehr ist XPoint so ausgelegt, dass man als Entwickler eigene Schemas erstellen kann.

Damit eine Zeichenkette in XPointer als Zeiger fungieren kann, muss sie einige Voraussetzungen erfüllen. Bei Kurzbeschreibungen ist das nur eine:

- Die Zeichenkette muss mit einem Buchstaben oder dem Unterstrich beginnen.

Bei der ausführlichen Notation müssen die folgenden Eigenschaften erfüllt sein:

- Die Zeichenkette muss mit dem Schemabezeichner beginnen.
- An den Schemabezeichner folgen in Klammern gesetzt die Schemadaten.
- Der Schemabezeichner kann ein Präfix besitzen, er muss aber in jedem Fall einen festen Namen haben. Die Syntax ist also entweder `Präfix:Name` oder `Name`.
- Innerhalb der Schemadaten können alle Unicode-Zeichen verwendet werden.

Abschließend noch einige Worte zu den Sonderzeichen: Wenn innerhalb von Zeichenketten in XPath Sonderzeichen eingesetzt werden, dann sind diese nicht mehr standardkonform. Aus diesem Grund sollten Sonderzeichen stets maskiert bzw. verschlüsselt werden. Als typisches Beispiel für eine solche Maskierung sei hier die ausführliche Schreibweise genannt, bei der runde Klammern verwendet werden. Denn kommen dabei weitere Klammern zum Einsatz, kann das zu Problemen führen. Wenn diese zusätzlichen Klammern nicht zu einer bestimmten Syntax gehören, müssen diese maskiert werden. Das sieht dann folgendermaßen aus:

```
MeinSchame( MeineD^(aten )
```

Der betreffenden Klammer muss also das Zeichen ^ vorangestellt werden. Soll hingegen das Zeichen ^ selbst verwendet werden, muss man diesem ebenfalls ein ^ voranstellen.

5.3.3 Der XPointer-Sprachschatz

Auf den folgenden Seiten werden die Möglichkeiten vorgestellt, die XPointer hinsichtlich des Zugriffs auf bestimmte Teile von XML-Dokumenten bietet. Dabei umfasst der XPointer-Sprachschatz einige Zeigerdefinitionen, die XML-Datensätze durchsuchen und darin auf einen bestimmten Bereich verweisen.

5.3.3.1 Zeiger in Kurzschreibweise

Beim Einsatz von Zeigern in Kurzschreibweise wird lediglich auf ein einzelnes Element des zu durchsuchenden XML-Datensatzes gezeigt. Um das zu erreichen, gibt man den Namen des Elements an. Wenn mehrere gleiche Elemente existieren, wird lediglich das erste angezeigt.

Die folgenden Regeln legen fest, wann ein Element als treffend ausgewählt wird:

- Wenn das Element ein `id`-Attribut besitzt, dessen Wert mit dem gesuchten Namen übereinstimmt.
- Wenn das Element ein Kindelement besitzt, auf das die zuvor genannte Bedingung zutrifft.

Es ist zu beachten, dass das id-Attribut an drei Stellen definiert sein kann. Diese drei Stellen sind

- XML Schema,
- DTD und
- eine externe Definition.

Wenn das `id`-Attribut innerhalb einer DTD spezifiziert wird, muss es sich dabei tatsächlich um den Typ ID handeln. Gleiches gilt, wenn das `id`-Attribut durch ein XML Schema definiert wurde. Auch hier muss es vom Typ ID abstammen und zum XML-Namensraum gehören. Externe `id`-Attribute können hingegen durch die entsprechenden verarbeitenden Programme auffindbar gemacht werden. Wird das Element allerdings nicht gefunden, sollte/muss dass verarbeitende Programm eine Fehlermeldung ausgeben.

5.3.3.2 Ausführliche Notation von Zeigern

Bei der ausführlichen Notation setzt sich der Zeiger immer aus zwei Teilen zusammen. Das ist einmal der Zeigername und einmal ein Zeigerteil, der in Klammern gesetzt wird.

```
news(definition)
```

Dabei zeigt der Zeigername an, wie der innerhalb der Klammern stehende Zeigerteil verarbeitet werden soll. Zur Verfügung stehen dafür die beiden nachfolgend detailliert beschriebenen beiden Zeigertypen `xpointer()` und `element()` sowie die Verknüpfung `xmlns()`.

5.3.3.3 Der Zeigertyp `xpointer()`

Beim Zeigertyp `xpointer()` handelt es sich zunächst einmal um einen ganz normalen XPath-Ausdruck. Allerdings lassen sich hier auch Funktionen und Schreibweisen von XPath einsetzen. Werfen Sie für ein besseres Verständnis einen Blick auf das folgende Beispiel:

```
news.xml#xpointer(id("beitrag"))
```

Durch diese Syntax wird auf ein Element gezeigt, das sich innerhalb des Dokuments `news.xml` befindet und die ID `beitrag` besitzt.

Ganz ähnlich sieht die Syntax übrigens aus, wenn mehrere Zeiger miteinander verknüpft werden sollen. In diesem Fall werden einfach Leerzeichen verwendet.

```
news.xml#xpointer(id("beitrag1")) xpointer(id("beitrag2"))
```

Durch diese Syntax wird zunächst innerhalb des Dokuments `news.xml` nach dem Element mit der ID `beitrag1` gesucht. Ist dieses Element nicht vorhanden, werden die Alternativen von links nach rechts durchgegangen. Im aktuellen Beispiel wird also anschließend nach dem Element mit der ID `beitrag2` „gefahndet“. Sobald ein Element gefunden wird, wird die Suche beendet.

5.3.3.4 Der Zeigertyp `element()`

Der Zeigertyp `element()` verweist auf ein einzelnes Element innerhalb eines Dokuments. Sie können dazu entweder eine Indexsequenz oder die entsprechende Element-ID angeben. Eine Indexsequenz setzt sich aus den einzelnen Indexnummern der Elemente zusammen, die jeweils durch einen Schrägstrich voneinander getrennt notiert werden müssen. Eingeleitet wird die Indexsequenz ebenfalls durch einen Schrägstrich.

Auch hierzu wieder ein Beispiel.

Listing 5.19 Ein einfaches XML-Dokument

```
<news id="ein">
   <beitrag1></beitrag1>
   <beitag2>
      <beitrag3 />
   </beitrag2>
</news>
```

Wollen Sie jetzt beispielsweise auf das Element `beitrag3` zugreifen, verwenden Sie die folgende Syntax:

```
#element(/1/2/1)
```

Ähnlich einfach ist es, die Sequenz mit einer ID zu verknüpfen. Das sieht folgendermaßen aus:

```
#element(erster/2/1)
```

5.3.3.5 Verknüpfungen mit `xmlns()`

Um eine XML-Ressource an einen Namensraum zu binden, wird die Verknüpfung `xmlns()` verwendet. Das funktioniert prinzipiell genauso wie bei `xmlns:Name="http://..."`. Im Zusammenhang mit XPath wird innerhalb der Klammern von `xmlns()` der Name des Namensraums angegeben. Darin schließen sich ein Gleichheitszeichen und die URL an.

```
xmlns(hv=http://www.hanser.de/)
```

Sobald ein entsprechendes Programm auf eine solche Verknüpfung stößt, wird diese ausgeführt und anschließend trotzdem der nächste Zeiger behandelt. Das geschieht, obwohl eigentlich nach dem ersten Erfolg keine zusätzliche Verarbeitung erfolgen würde.

Auch hierzu wieder ein typisches Beispiel, in dem die Quelldatei folgendermaßen aussieht:

Listing 5.20 Das ist das XML-Dokument.

```
<feed xmlns="http://content">
   <meldung xmlns="http://news">
   </meldung>
</feed>
```

Durch den folgenden Aufruf

```
xmlns(Inhalt=http://content) xmlns(Neuigkeiten=http://news)
xpointer(/Inhalt:feed/Neuigkeiten:meldung)
```

lässt sich über die Namensräume hinweg auf die Elemente zugreifen. Interessant ist das vor allem, wenn es mehrere gleichnamige Elemente gibt, die aber zu einem jeweils anderen Namensraum gehören.

5.3.4 XPointer innerhalb von URIs

Ein URI, durch den ein Dokument spezifiziert wird, sieht üblicherweise folgendermaßen aus:

```
http://www.hanser.de/content/news.html
```

Dabei gibt zunächst `http` darüber Auskunft, welches Protokoll die Applikation verwenden soll, um das Dokument zu beziehen. Die Quelle `www.hanser.de/` wiederum gibt an, auf welchem Host die Quelle liegt. Zu guter Letzt werden über `content/news.html` Pfad und Dokument bestimmt. Diese Aspekte sind Ihnen bereits aus XPath bekannt. Ebenfalls bekannt ist Ihnen möglicherweise die Tatsache, dass in einigen URIs Fragmentbezeichner enthalten sein können, die einen benannten Anker innerhalb des Dokuments kennzeichnen, über den der URI-Pfad identifiziert wird. Dazu wird innerhalb des URI eine Raute gesetzt, die den Anker vom eigentlichen Pfad trennt.

```
http://www.hanser.de/news.html#oben
```

Wird nun einem Link zu diesem URI gefolgt, sucht der Browser innerhalb des Dokuments `news.html` nach dem Anker `oben`. Definiert wurde der innerhalb des Dokuments folgendermaßen:

```
<a name="oben"></a>
```

Wird der Anker gefunden, verschiebt sich das Browser-Fenster automatisch an die Position, an welcher der Anker definiert wurde. Für HTML ist diese Variante ideal. Hinsichtlich solcher Dokumente, bei denen es sich nicht um normale Webseiten handelt, stößt diese Technik allerdings schnell an ihre Grenzen. Denn um auf eine bestimmte Stelle innerhalb eines Dokuments verweisen zu können, muss an der gewünschten Stelle ein Anker definiert sein. Und genau das setzt voraus, dass man Zugriff auf das Zieldokument hat und dort einen Anker einfügen kann.

XPointer versucht diese Beschränkungen aufzuheben, in dem es ermöglicht, mittels vollständiger XPath-Ausdrücke als Fragmentbezeichner festzulegen, wohin ein Link gerichtet wird. Zudem erweitert XPointer XPath dahingehend, dass Operationen zur Verfügung gestellt werden, mit denen sich Punkte und Bereiche in einem XML-Dokument auswählen lassen, die nicht unbedingt mit einem Knoten oder einer Knotenmenge übereinstimmen.

Listing 5.21 Hier ein paar typische Beispiele für XPointer

```
xpointer(/)
xpointer(//vname)
xpointer(id('eng'))
xpointer(/autoren/autor/name/vname/text( ))
xpointer(//beruf[.="Schriftsteller"])
xpointer(//mittelinitial[position( )=1]/../vname)
xpointer(/child::autoren/child::autor/attribute::id)
```

Ob ein XPointer keinen, einen oder mehrere Knoten identifizieren kann, hängt davon ab, in Bezug auf welches Dokument ein XPointer bewertet wird. Normalerweise handelt es sich bei den gefundenen Knoten um Elemente. Ebenso kann es sich aber auch um Attribut- oder Textknoten und um Punkte und Bereiche handeln.

Nicht immer ist klar, ob ein XPointer auch tatsächlich etwas findet. Um Fehler vorzubeugen, können Sie einen alternativen XPointer anlegen. Hierzu ein Beispiel:

```
xpointer(//vname)xpointer(//nname)
```

Dieser XPointer sucht zunächst nach `vname`-Elementen. Findet er keins, werden `nname`-Elemente gesucht. Beachten Sie, dass in diesem Beispiel die `nname`-Elemente nur dann gefunden werden, wenn keine `vname`-Elemente vorhanden sind.

Die Anzahl der XPointer ist übrigens nicht begrenzt. Wenn Sie wollen, können Sie also auch fünf XPointer hintereinander notieren.

```
xpointer(//vname)xpointer(//nname)xpointer(//synomym)
```

Werden hier weder `vname`- noch `nname`-Elemente gefunden, wird nach `synonym` gesucht. Sollte die Suche erfolglos bleiben, wird eine leere Knotenmenge zurückgegeben.

Noch ein Hinweis zur Syntax. Im vorherigen Beispiel wurden die einzelnen XPointer direkt hintereinander notiert. Wenn Ihnen diese Variante zu unübersichtlich ist, können Sie jeweils ein Leerzeichen einfügen.

```
xpointer(//vname) xpointer(//nname) xpointer(//synomym)
```

Am Ergebnis ändert sich nichts.

5.3.5 XPointer innerhalb von Hyperlinks

Worauf ein XPointer tatsächlich verweist, hängt letztendlich von dem Dokument ab, auf das er angewendet wird. Dieses Dokument wiederum wird durch den URI bestimmt, mit dem der XPointer verknüpft ist.

Wollen Sie beispielsweise einen URI bekommen, der auf das erste `name`-Element in dem Dokument *http://www.hanser.de/books.xml* verweist, dann sähe die Syntax folgendermaßen aus:

```
http://www.hanser.de/books.xml#xpointer(//name[position( )=1])
```

Vielleicht kennen Sie die Codierung von URIs bereits aus Programmiersprachen wie PHP. Genauso wie dort müssen solche Zeichen, die innerhalb von URIs nicht erlaubt sind, auch bei XPointer speziell ausgezeichnet werden. Solche Zeichen sind beispielsweise die folgenden:

- `<`
- Alle nicht ASCII-Zeichen
- Doppelte Anführungszeichen

Jedes dieser Zeichen muss speziell maskiert werden. Dazu gibt man das Prozentzeichen, gefolgt vom Hexadezimalwert jedes Bytes des Zeichens in der UTF-8-Kodierung ein. Hier mal ein paar Beispiele für eine solche Umwandlung:

- Aus `<` wird `%3C`.
- Aus `"` wird `%22`.
- Aus ë wird `%c3%ab`.
- Aus § wird `%c2%a7`.

Das soll es an dieser Stelle mit Beispielen zu diesem Thema gewesen sein. Eine gute Übersicht über UTF-8 finden Sie unter *http://www.utf8-zeichentabelle.de/*.

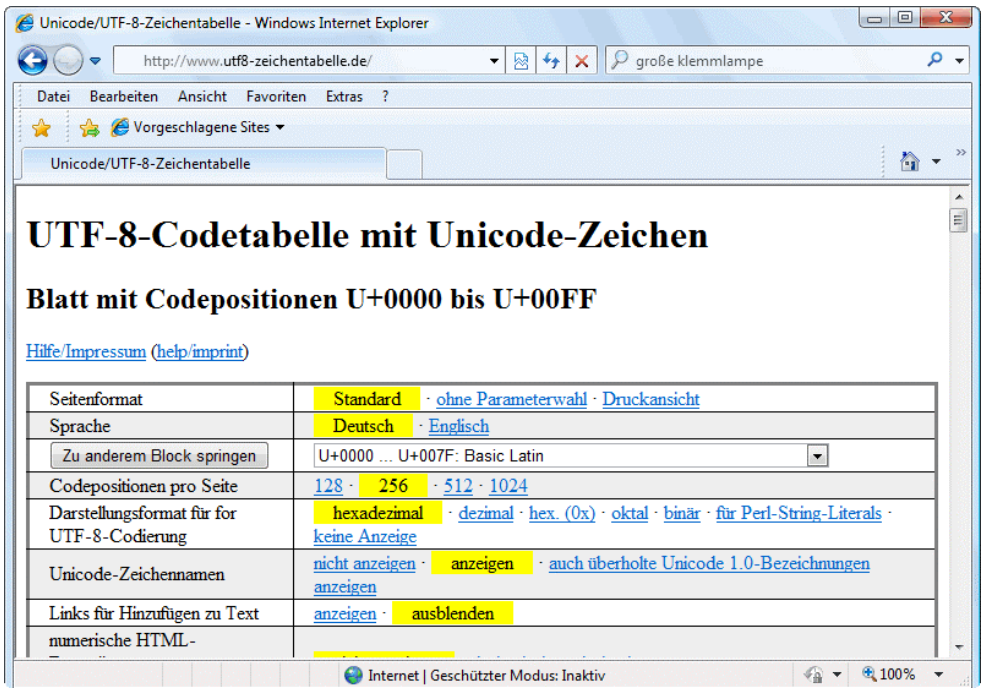

Abbildung 5.9 Gute Übersicht über die UTF-8-Zeichen

Listing 5.22 Ein typischer Verweis in XPointer.

In HTML können URLs in a-Elementen einen XPoint-Fragmentbezeichner enthalten.

```
<a href = "http://www.hanser.de/buch.xml#xpointer
  (//name[position( )=1])">XML
</a>
```

Wird dieser Link allerdings im Browser angeklickt, dann würde das gesamte Dokument geladen, das unter *http://www.hanser.de/buch.xml* zu finden ist, und zusätzlich an das erste name-Element verschoben. Allerdings gibt es bislang noch keinen Browser, der diese XPointer-Syntax tatsächlich umsetzt. Ausnahme bilden hier die Mozilla-Browser ab Version 1.4. Diese unterstützen zumindest das XPointer-Schema `xpath1()`. Entwickelt wurde dieses Schema von Simon St. Laurent. Ausführliche Informationen dazu finden Sie unter *http://www.simonstl.com/ietf/draft-stlaurent-xpath-frag-00.html*. An dieser Stelle nur so viel: `xpath1()` funktioniert prinzipiell ähnlich wie `xpointer()`. Allerdings schließt `xpath1()` nicht die Erweiterungen für Punkte und Bereiche mit ein. Es werden vielmehr ausschließlich Ausdrücke unterstützt, die in XPath 1.0 implementiert sind.

XPointer können ausschließlich auf XML-Dokumente zeigen. Das liegt daran, dass XPath nur Knoten innerhalb wohlgeformter XML-Dokumente finden kann. Im Umkehrschluss können Sie aber XPointer innerhalb beliebiger Nicht-XML-Dateien wie beispielsweise HTML- und Text-Dateien verwenden.

XPointer werden sehr oft innerhalb von XLinks verwendet. Auch hierzu wieder ein kleines Beispiel:

Listing 5.23 So sieht der Zugriff auf Elemente aus.

```
<verweis xlink:type="simple"
xlink:href="buch.xml#xpointer(/bibliothek/buecher/buch[position( )=1])">
   XML-Buch
</verweis>
```

Durch die gezeigte Syntax wird auf das erste `buch`-Element des Kindelements `buecher` vom Wurzelelement `bibliothek` innerhalb des Dokuments `buch.xml` gezeigt.

Innerhalb erweiterter Links können XPointer bei der Identifizierung von Anfangs- und Zielressourcen eines Bogens helfen. Auch dieser Aspekt lässt sich wieder am besten anhand eines kleinen Beispiels zeigen.

Listing 5.24 Hier wurde ein erweiterter Link verwendet.

```
<bibliothek xlink:type="extended"
xmlns:xlink="http://www.w3.org/1999/xlink">

<buch xlink:type="locator" xlink:label="parsons"
xlink:href="parsons.xml#xpointer(//seite[position()=last( )])"/>

<buch xlink:type="locator" xlink:label="hornby"
xlink:href="hornby.xml#xpointer(//seite[position( )=1])" />

<nächstes xlink:from="parsons" xlink:to="hornby"/>
<vorheriges xlink:from="hornby" xlink:to="parsons"/>

</bibliothek>
```

In diesem Beispiel legt der erweiterte Link einen Bogen zwischen dem letzten `seite`-Element des Dokuments `parsons.xml` und dem ersten `seite`-Element des Dokuments `hornby.xml`. Anschließend erstellt er einen Link vom ersten `seite`-Element in `hornby.xml` zu dem `seite`-Element innerhalb von `parsons.xml`.

5.3.6 XPath-Erweiterungen in XPointer

XPointer erweitert XPath um einige zusätzliche Funktionen und Typen. Bevor im nächsten Abschnitt detailliert auf die einzelnen Funktionen eingegangen wird, an dieser Stelle ein paar allgemeine Hinweise zum Thema.

5.3.6.1 Punkte und Bereiche

XPointer erweitert XPath um die beiden Typen `point` und `range`. Dadurch wird es möglich, anstelle von ganzen Knoten auch auf bestimmte Stellen oder Bereiche innerhalb eines Knotens bzw. über mehrere Knoten hinaus auszuwählen.

Ein Punkt kann durch eine feste Position oder durch eine von XPointer zur Verfügung gestellte Funktion ausgewählt werden. (Mehr zu diesen Funktionen dann später.) An dieser Stelle zunächst die Positionsvariante.

```
xpointer(id("element")[0])
```

Durch diese Syntax wird die erste Stelle des Elementinhalts ausgewählt.

Bereiche werden jeweils durch zwei Punkte, und zwar durch Start- und Endpunkt definiert.

5.3.6.2 Funktionen

Bevor auf den folgenden Seiten die in XPointer zur Verfügung stehenden Funktionen im Detail vorgestellt werden, hier ein kurzer, knackiger Überblick. Damit Sie gleich wissen, was auf Sie zukommt, hier die entsprechenden Funktionen:

- `range()`
- `range-inside()`
- `range-to()`
- `string-range()`
- `covering-range()`
- `start-point()`
- `end-point()`
- `here()`
- `origin()`

Und dann auch gleich noch ein Beispiel für den Einsatz dieser Funktionen.

```
xpointer(id("element")[0]/range-to(id("element")[0]))
```

In diesem Beispiel wird die XPointer-Funktion `range-to()` eingesetzt. Wird diese Funktion mit einem Punkt verknüpft, bekommt man einen Bereich. Wie bereits geschrieben: So viel in aller Kürze zu den Funktionen. Auf den folgenden Seiten finden Sie ausführliche Informationen zum Thema.

5.3.7 Mit Funktionen arbeiten

In XPointer gibt es einige Funktionen, die sich bezogen auf Punkte und Bereiche anwenden lassen. Welche das sind, zeigen die folgenden Seiten:

5.3.7.1 Die Funktion `range()`

Die Funktion `range()` übernimmt einen XPath-Ausdruck, der als Argument eine Lokalisierungsmenge zurückliefert. Für alle Knoten innerhalb der Lokalisierungsmenge wird ein Bereich zurückgegeben, durch den der Bereich abgedeckt wird. Wichtig ist hierbei zu verstehen, was Start- und was Endpunkt des Bereichs ist. Der Startpunkt liegt unmittelbar vor dem Knoten, während sich der Endpunkt unmittelbar hinter dem Knoten befindet. Sollte es sich bei dem Knoten um ein Element handeln, dann beginnt der Knoten direkt vor dem Start-Tag des Elements und endet direkt hinter dem schließenden Tag.

Das folgende Beispiel zeigt den Einsatz der `range()`-Funktion:

Listing 5.25 Das ist das Ausgangsdokument.

```
<bibliothek>

   <autor ID="a1">
      <name>Nick Hornby</name>
      <buch>About a boy</buch>
```

```
      <preis>19,95</preis>
   </autor>

   <autor ID="a2>
      <name>Rainald Grebe</name>
      <buch>Global Fish</buch>
      <preis>14,90</preis>
   </autor>

</bibliothek>
```

Mittels `range()` soll nun auf den Inhalt des Elements `buch` von `a1` zugegriffen werden. Der entsprechende Aufruf sieht folgendermaßen aus:

```
xpointer(range(//ID(„a1“)/buch))
```

Es wird also exakt der Bereich ausgewählt, der exakt das einzelne `buch`-Element mit der ID a1 umfasst.

```
About a boy
```

Alle anderen `buch`-Elemente bleiben von der Abfrage unberührt.

5.3.7.2 Die Funktion range-inside()

Die Funktion `range-inside()` übernimmt einen XPath-Ausdruck, der eine Lokalisierungsmenge zurückliefert als Argument. Für jeden Knoten dieser Lokalisierungsmenge wird der Bereich zurückgeliefert, der den Inhalt des Knotens umfasst. Das Ergebnis ist immer identisch mit dem Ergebnis der `range()`-Funktion. Ausnahme bildet hier lediglich ein Elementknoten, bei dem dieser Bereich alles umfasst, was sich innerhalb des betreffenden Elements befindet, bis auf Start- oder End-Tag.

Auch zur `range-inside()`-Funktion wieder ein Beispiel:

Listing 5.26 Dieses Dokument kennen Sie bereits.

```
<bibliothek>

   <autor ID="a1">
      <name>Nick Hornby</name>
      <buch>About a boy</buch>
      <preis>19,95</preis>
   </autor>

   <autor ID="a2>
      <name>Rainald Grebe</name>
      <buch>Global Fish</buch>
      <preis>14,90</preis>
    </autor>

</bibliothek>
```

Durch den folgenden Aufruf soll auf den Inhalt des Elements, das innerhalb des `autor`-Elements mit der ID a1 steht, zugegriffen werden.

```
xpointer(range-inside(//ID(„a1“)/preis))
```

Ausgegeben wird Folgendes:

```
19,95
```

5.3.7.3 Die Funktion `range-to()`

Durch die `range-to()`-Funktion wird ein Bereich geliefert, der von der aktuellen Knotenposition bis zur angegebenen Zielposition reicht. `range-to()` kann als Schritt in einem Pfad auftreten.

Listing 5.27 Ein Beispiel für `range-to()`

```
<bibliothek>

   <autor ID="a1">
      <name>Nick Hornby</name>
      <buch>About a boy</buch>
      <preis>19,95</preis>
   </autor>

   <autor ID="a2">
      <name>Rainald Grebe</name>
      <buch>Global Fish</buch>
      <preis>14,90</preis>
   </autor>

   <autor ID="a3">
      <name>Tony Parsons</name>
      <buch>Als wir unsterblich waren</buch>
      <preis>8,90</preis>
   </autor>

   <autor ID="a4">
      <name>Benjamin von Stuckrad-Barre</name>
      <buch>Soloalbum</buch>
      <preis>8,00</preis>
   </autor>

</bibliothek>
```

Durch die `range-to()`-Funktion lässt sich nun der gewünschte Bereich auswählen.

```
xpointer(//autor[2]range-to(id("a4"))))
```

Dieser Funktionsaufruf liefert das folgende Ergebnis:

Listing 5.28 Das ist das Ergebnis.

```
<autor ID="a2">
   <name>Rainald Grebe</name>
   <buch>Global Fish</buch>
   <preis>14,90</preis>
</autor>

<autor ID="a3">
   <name>Tony Parsons</name>
   <buch>Als wir unsterblich waren</buch>
   <preis>8,90</preis>
</autor>

<autor ID="a4">
   <name>Benjamin von Stuckrad-Barre</name>
   <buch>Soloalbum</buch>
   <preis>8,00</preis>
</autor>
```

5.3.7.4 Die Funktion `string-range()`

Durch `string-range()` werden alle Textbereiche geliefert, die der angegebenen Zeichenfolge entsprechen. Durch zusätzliche Argumente können die Nummern des ersten Zeichens und die Anzahl der Zeichen angegeben werden, die übernommen werden sollen.

Listing 5.29 Das ist das Ausgangsdokument.

```
<bibliothek>

   <autor ID="a1">
      <name>Nick Hornby</name>
      <buch>About a boy</buch>
      <preis>19,95</preis>
   </autor>

   <autor ID="a2>
      <name>Rainald Grebe</name>
      <buch>Global Fish</buch>
      <preis>14,90</preis>
   </autor>

</bibliothek>
```

Durch den Aufruf

```
xpointer(string-range(//*,"Grebe")
```

wird folgendes Ergebnis erzielt:

```
Grebe
```

5.3.7.5 Die Funktion `covering-range()`

Die Funktion `covering-range()` gibt eine Menge von Bereichen wieder, die zur angegebenen Definition passen. Zu diesem Zweck wird jedem der Position entsprechenden Knoten, Punkt oder Bereich eine sogenannte Umrandung zugewiesen, die ausgegeben wird. Dies geschieht folgendermaßen: Bei Punkten wird ein leerer Bereich ausgegeben, der als Start- und Endpunkt eben jenen Punkt besitzt. Bei Element-, Root-, PI- und Kommentarknoten wird der Start- mit dem Endpunkt des Bereichs gleichgesetzt. Und zwar mit dem Anfang und dem Ende des jeweiligen Knotens. Bei Attribut- und Namespace-Knoten wird als Bereich hingegen das Attribut bzw. der Namespace selbst wiedergegeben.

Listing 5.30 Das ist das Ausgangsdokument.

```
<bibliothek>

   <autor ID="a1">
      <name>Nick Hornby</name>
      <buch>About a boy</buch>
      <preis>19,95</preis>
   </autor>

   <autor ID="a2>
      <name>Rainald Grebe</name>
      <buch>Global Fish</buch>
      <preis>14,90</preis>
   </autor>

</bibliothek>
```

Der Aufruf

```
xpointer(covering-range(string-range(//autor[2]/*,"Global",2,2))
```

liefert das folgende Ergebnis:

```
<buch>Global Fish</buch>
```

5.3.7.6 Die Funktion `start-point()`

Gibt eine Menge von Punkten aus, die dem Anfangspunkt der angegebenen Position entsprechen. Ein Beispiel:

Listing 5.31 So sieht das Ausgangsdokument aus.

```
<bibliothek>

   <autor ID="a1">
      <name>Nick Hornby</name>
      <buch>About a boy</buch>
      <preis>19,95</preis>
   </autor>

   <autor ID="a2>
      <name>Rainald Grebe</name>
      <buch>Global Fish</buch>
      <preis>14,90</preis>
   </autor>

</bibliothek>
```

Durch die Syntax

```
xpointer(start-point(string-range(//autor[2]/name,"Rainald")))
```

wird der Startpunkt festgelegt. Im Ergebnisdokument habe ich den Startpunkt hier mit einem X gekennzeichnet.

Listing 5.32 Achten Sie auf das `X`.

```
<bibliothek>

   <autor ID="a1">
     <name>Nick Hornby</name>
      <buch>About a boy</buch>
      <preis>19,95</preis>
   </autor>

   <autor ID="a2>
      <name>X Rainald Grebe</name>
      <buch>Global Fish</buch>
      <preis>14,90</preis>
   </autor>

</bibliothek>
```

5.3.7.7 Die Funktion `end-point()`

`end-point()` ist das Gegenstück zu `start-point()`. Die Funktion gibt eine Menge von Punkten aus, die dem Anfangspunkt der angegebenen Position entsprechen.

5.3.7.8 Die Funktion `here()`

Diese Funktion gibt die aktuelle Position zurück, an der sich der XPointer befindet.

Listing 5.33 Liefert den XPointer()

```
<button xlink:type="simple"
xlink:href="#xpointer(here()
/ancestor::slide[1]/preceding::slide[1])">
   Vor
</button>
```

5.3.7.9 Die Funktion `origin()`

Wird anstelle von `here()` eingesetzt, wenn Links verwendet werden, die sich außerhalb des Quell- und des Zieldokuments befinden. (Typisches Beispiel dafür sind Linkdatenbanken.)

5.4 XLink – Links in XML definieren

Wenn man sich die Frage über den Grund für den Erfolg von HTML stellt, kommen unweigerlich die Hyperlinks ins Spiel. Denn erst durch die kann sich der Benutzer ganz nach Belieben die miteinander verknüpften Informationen bzw. Webseiten anzeigen lassen.

Nun sollen natürlich nicht nur HTML-, sondern auch XML-Dokumente miteinander verknüpft werden. An diesem Punkt kommt XLink ins Spiel. Bei dieser vom W3C entwickelten Sprache handelt es sich um eine attributbasierte Syntax zur Definition von Links innerhalb von XML-Dokumenten. Die Originalspezifikation finden Sie auf der Seite *http://www.w3.org/TR/2001/REC-xlink-20010627/*. Parallel dazu existiert aber auch eine deutsche Übersetzung.

Abbildung 5.10 Die deutsche Übersetzung der Spezifikation gibt es unter *http://www.edition-w3c.de/TR/2001/REC-xlink-20010627/*.

Nun ist XLink allerdings nicht einfach die XML-Umsetzung der Link-Fähigkeiten, die HTML hinsichtlich der Hyperlinks zu bieten hat. Denn HTML weist hier deutliche Schwächen auf.

- In HTML existieren lediglich zwei Elementtypen, die für Hyperlinks genutzt werden können. Das sind a und – zumindest eingeschränkt – img.
- Links in HTML können ausschließlich auf ein Verweisziel zeigen. Das gleichzeitige Verweisen auf mehrere Ziele ist nicht möglich.
- HTML-Links gehen immer in eine Richtung. Dabei führen sie von der Stelle, an der sie im Ausgangsdokument stehen, zu dem Verweisziel, das im Link über eine URI-Referenz angegeben ist.
- Die Hyperlinks müssen tatsächlich im Dokument stehen. Man kann also nicht etwa eine Verknüpfung einrichten, die von einem Dokument ausgeht, das beispielsweise aufgrund von fehlenden Schreibrechten nicht bearbeitet werden darf.

Diese Probleme sollen durch XLink gelöst werden. Und auch wenn die Implementierung des Standards nur schleppend vorangeht, hält das W3C weiter an XLink fest. Dabei soll XLink die folgenden Kriterien erfüllen:

- Hyperlinks können mehrere Ziele ansteuern.
- Metadaten können den Hyperlinks zugewiesen werden.

- Es ist möglich, Hyperlinks in beide Richtungen zu begehen.
- Hyperlinks lassen sich unabhängig vom Dokument speichern. So wird es möglich, Hyperlinks auch zu solchen Dokumenten zu definieren, die sich nicht bearbeiten lassen.

Verabschiedet wurde Version 1.0 der XLink-Spezifikation bereits im Jahr 2001. Die folgenden Seiten bringen Ihnen diese Sprache näher.

Bevor es jedoch ins Detail geht, noch ein paar Worte zu den grundlegenden Begrifflichkeiten, die Ihnen im XLink-Umfeld immer wieder begegnen werden. Zunächst einmal stand das W3C vor dem Problem, den Begriff Links flexibler zu definieren. Dazu wurden Links als eine explizite Beziehung zwischen Teilen von Ressourcen oder ganzen Ressourcen definiert. Dabei ist eine Ressource eine beliebig adressierbare Information, die man über einen URI erreichen kann.

XLinks werden in Elemente mit beliebigen Elementnamen integriert, indem man den Elementen bestimmte XML-Attribute des XML-Namensraums http://www.w3.org/1999/xlink hinzufügt. Hier unterscheidet sich XLink also deutlich von anderen Sprachen wie XSLT oder XSL, wo eigene Elemente verwendet werden.

Folgende Begriffe sind bei XLink besonders wichtig:

- Arcs – Hierdurch wird der Pfad benannt, der beim Folgen eines Links gegangen werden muss. Dabei hat ein Arc immer eine Richtung. Will man einen bidirektionalen Link definieren, muss man zwei Arcs verwenden, die jeweils vertauschte Start- und Zielressourcen besitzen.
- Outbound Link – Der Arc zeigt von einer lokalen auf eine entfernte Ressource.
- Inbound Link – Der Arc zeigt von einer entfernten auf eine lokale Ressource.
- Third-Party-Links – Die Arcs gehen von einer entfernten zu einer entfernten Ressource. Dabei sind die Links von den Ressourcen getrennt, zwischen denen eine Verknüpfung hergestellt werden soll. Bezeichnet werden solche XML-Dateien oder Datenbanken als Linkbases.
- Traversal – So wird die Verwendung eines Links genannt.

Nachdem die Grundbegriffe geklärt sind, geht es nun um die Link-Typen und verschiedene Attribute.

5.4.1 Link-Typen und andere Attribute

Bei XLink gibt es eine Vielzahl globaler Attribute, mit denen die verschiedenen XLink-Elementtypen beschrieben werden. Für diesen Zweck existiert ein spezieller Namensraum.

```
xmlns:xlink="http://www.w3.org/1999/xlink
```

Diese Namensraumdeklaration muss in das XML-Dokument eingebunden werden, bevor man XLink-Attribute verwendet.

XML-Elemente können als XLink-Element verwendet werden. Dazu muss ihnen mindestens das Attribut `type` mit einem gültigen Wert zugewiesen werden. Welche Werte das sind, das zeigt die folgende Übersicht:

- `arc` – Liefert Traversal-Regeln für die an dem Link beteiligten Ressourcen.
- `extended` – Definiert einen erweiterten Link. (Ein Element vom Typ `extended` ist außer `simple` das einzige Element, über das man einen XLink erzeugen kann.)
- `locator` – Hierüber werden die entfernten Ressourcen angegeben, die zu einem Link gehören.
- `none` – Bei dem Element handelt es sich im Sinne von XLink um keinen Link.
- `resource` – Liefert lokale Ressourcen, die Teil der Verknüpfung sind. (`ressource` taucht dabei innerhalb eines `extended`-Elements auf.)
- `simple` – Ein `simple`-Element ist eine Kombination aus Funktionalität `extended`-Element mit je einem `resource`-, `locator`-, `arc`- und `title`-Element. Es handelt sich hierbei um einen einfachen Link, der dem aus HTML bekannten `a`-Element entspricht.
- `title` – Für den Link wird ein Label bzw. Titel erzeugt.

Von dieser Liste liefern lediglich die beiden Werte `extended` und `simple` ein entsprechendes Link-Element. Durch die anderen werden hingegen zusätzliche Informationen für erweiterte Links angegeben.

Die folgende Auflistung zeigt die möglichen Attribute. Diese werden hier zunächst ganz allgemein vorgestellt. Im weiteren Verlauf dieses Kapitels wird dann detailliert beschrieben, ob sie für erweiterte oder einfache Links verwendet werden.

- `actuate` – Hierüber wird bestimmt, durch welches Ereignis die Traversalisierung ausgelöst werden soll. Mit `onLoad` wird bestimmt, dass die Zielressource zusammen mit der Startressource geladen werden soll. Soll die Zielressource erst auf Anforderung des Benutzers geladen werden, wird `onRequest` verwendet. Durch `other` kann die betreffende Anwendung nach zusätzlicher Markup suchen, in der es andere Hinweise gibt. Um der Anwendung die vollständige Entscheidungsfreiheit zu gewähren, wird `none` verwendet.
- `arcrole` – Ermöglicht mittels eines URI, die spezielle Bedeutung der Zielressource zu beschreiben. Das gilt allerdings nur, wenn diese über den entsprechenden Pfad angesteuert wird.
- `href` – In Form eines URI werden die Daten geliefert, mit denen entfernte Ressourcen gefunden werden können.
- `role` – Hierüber können mittels eines URI Art und Zweck der Zielressource angegebenen wird. So lässt sich z.B. beschreiben, dass es sich um einen Kommentar handelt.
- `show` – Damit wird bestimmt, auf welche Art und Weise die Zielressource angezeigt werden soll. Hierbei stehen mehrere Werte zur Verfügung. Mit `embed` wird die Zielressource innerhalb der aufrufenden Seite angezeigt. Über `other` erlaubt man es einer Anwendung, nach Markup zu suchen, in der entsprechende Hinweise stehen. `replace`

sorgt dafür, dass die Zielressource die aufrufende Seite ersetzt. Und schlussendlich gibt es noch `none`, wodurch die Anwendung selbst entscheidet.

- `title` – Weist dem Link eine Beschriftung zu..

Je nachdem, welchen Link-Typ Sie einsetzen, unterliegen Sie bestimmten Einschränkungen. So sind einige Attribute bei manchen Typen erlaubt, während andere nicht eingesetzt werden dürfen. Tabelle 5.19 liefert die entsprechende Übersicht.

Tabelle 5.19: Die möglichen Attribute

Attributtyp	arc	extended	locator	resource	simple	title
actuate	optional				optional	
arcrole	optional				optional	
from	optional					
href			erforderlich		optional	
label			optional	optional		
role		optional	optional	optional	optional	
show	optional				optional	
title	optional	optional	optional	optional	optional	
to	optional					
type	erforderlich	erforderlich	erforderlich	erforderlich	erforderlich	erforderlich

5.4.2 Einfache Links anlegen

Ein einfacher Link setzt exakt zwei Ressourcen zueinander in Beziehung. Dabei läuft der Link immer von der lokalen zur entfernten Ressource, es handelt sich somit stets um einen ausgehenden Link. Einfache Links können in einem beliebigen Element stehen und werden durch die folgenden Attribute beschrieben:

- Über `type` definiert man einen einfachen Link.
- Über `href` wird das eigentliche Verweisziel angegeben.

Diese Attribute genügen, um einen einfachen Link zu erzeugen.

Listing 5.34 Das ist ein einfacher Link.

```
<verweis xmlns:xlink="http://www.w3.org/1999/xlink"
         xlink:type="simple"
         xlink:href="http://www.hanser.de/">Hanser</meinLink>
```

Neben den beiden genannten Attributen sind die folgenden optional möglich[2]:

[2] Was es mit diesen Attributen auf sich hat, wurde im vorherigen Abschnitt gezeigt.

- actuate
- arcrole
- role
- show
- title

Jetzt noch einmal das gleiche Beispiel wie zuvor, dieses Mal wurde jedoch zusätzlich das role-Attribut mit aufgenommen.

Listing 5.35 Ein weiteres Attribut wurde aufgenommen.

```
<verweis xmlns:xlink=http://www.w3.org/1999/xlink
          xlink:type="simple"
          xlink:role="http://www.hanser.de/news.php"
          xlink:href="http://www.hanser.de/">Hanser</meinLink>
```

5.4.3 Erweiterte Links definieren

Ein erweiterter Link assoziiert eine beliebige Anzahl von Ressourcen. Üblicherweise werden erweiterte Link getrennt von den Ressourcen gespeichert. Interessant ist diese Linkart vor allem dann, wenn die Ressourcen nicht verändert werden können bzw. eine Anpassung sehr aufwendig wäre.

Erweiterte Links können beliebige Elemente mit einem Attribut type aus dem XLink-Namensraum sein, dem der Attributwert extended zugewiesen wurde.

Diese Syntax bildet den Rahmen. Innerhalb dieses Rahmens können weitere Elemente stehen, die den Link genauer beschreiben.

Listing 5.36 Ein erweiterter Link.

```
<verweis xlink:type="extended">
   ... weitere Definitionen zum Link ...
</verweis >
```

So ausgezeichneten Elementen können optional die beiden Attribute

```
tilte
```

und

```
role
```

zugewiesen werden. Zusätzlich dürfen die folgenden Elemente in beliebiger Reihenfolge enthalten sein:

- arc
- locater
- resource

In der W3C-Spezifikation zu den XLinks wird explizit darauf hingewiesen, dass es kein Fehler ist, wenn ein erweiterter Link auf weniger als zwei Ressourcen verweist.

Zudem ist darauf zu achten, dass die beschreibenden Elemente innerhalb des erweiterten Links direkte Nachfahren des `extended`-Elements sein müssen. Unterelemente vom Typ `simple` oder `extended`, die innerhalb eines Vaterelements `extended` stehen, haben keine XLink-spezifizierte Bedeutung.

5.4.3.1 Lokale Ressourcen erstellen

Ein erweiterter Link weist die dazugehörenden Ressourcen über spezielle Unterelemente aus, die innerhalb des erweiterten Links auftauchen. Ein vollständiges Unterelement bildet, zusammen mit seinem gesamten Inhalt, die lokale Ressource. Das entsprechende Element muss dabei das Attribut `type` besitzen, dem der Wert `ressource` zugewiesen wird. Parallel dazu können die folgenden drei Attribute angegeben werden:

- `label` – Hierüber wird dem Element ein Label zugewiesen, unter dem die zusammengehörenden Ressourcen verknüpft sind. Durch `label` wird es möglich, dass sich ein Element vom Typ arc bei der Erzeugung einer Traversierungskante auf die lokale Ressource beziehen kann.
- `role` – Dieses Attribut beschreibt eine Eigenschaft der Ressource.
- `title` – Der hier angegebene Text gibt den Titel der Zielressource an.

Auch hierzu wieder ein Beispiel:

Listing 5.37 Eine lokale Ressource

```
<verweis xlink:type="extended">
   <text xlink:type="resource"
         xlink:role="http://www.hanser.de/text">
      Neuigkeiten
    </text>
</verweis>
```

5.4.3.2 Auf externe Ressourcen verweisen

Es handelt sich bei einem erweiterten Link bekanntermaßen auch um einen Link im herkömmlichen Sinn. Daher sollten auch eine oder mehrere externe Ressourcen beschrieben werden, auf die der Link verweist. Das XLink-Element für Adressangaben ist eines mit dem Attribut `type` aus dem XLink-Namensraum, das den Attributwert `locator` besitzt. Ein solches Element muss das XLink-Attribut `href` haben, das wiederum auf die externe Ressource verweist. Ein Element vom Typ `locator` darf die folgenden Attribute besitzen:

- `role`
- `title`
- `label`

`locator`-Elemente zählen für sich gesehen nicht als Links[3], da es keine durch XLink bestimmte Assoziation zwischen sich und der entfernten Ressource erzeugt.

[3] Auch wenn das `href`-Attribut etwas anderes vermuten lässt.

Listing 5.38 Der Verweis auf externe Ressourcen

```
<verweis xlink:type="extended">

 <extern xlink:type="locator"
         xlink:href=http://www.hanser.de/
         xlink:title="HANSER" />

 <extern xlink:type="locator"
         xlink:href="http://www.hanser-verlag.de/"
         xlink:title="HV" />

</verweis>
```

Wird ein `locator`-Element definiert, muss diesem das `href`-Attribut in jedem Fall zugewiesen werden. Zusätzlich muss `href` einen Attributwert besitzen.

5.4.3.3 Die Traversierungsregeln bestimmen

Erweiterte Links können Traversierungsregeln zwischen beteiligten Ressourcen definieren. Um die Traversierungsregeln für einen Verweis festzulegen, wird ein Element mit dem Attribut `type` aus dem XLink-Namensraum ausgestattet, dem der Attributwert `arc` zugewiesen wird.

Folgende Attribute sind verfügbar:

- `actuate` – Hierüber wird bestimmt, wann der Verweis aktiviert wird. Mögliche Werte sind `onRequest` (Ziel sollte geladen werden, wenn es der Benutzer anfordert.), `onLoad` (Ziel sollte mit dem Laden der Anwendung geladen werden.), `other` (Keine spezielle Angabe zum Ladezeitpunkt. Der Browser sollte aber nach Informationen im umgebenden Text suchen.) und `none` (Keine spezielle Angabe zum Ladezeitpunkt.).
- `arcrole` – Beschreibt die semantische Bedeutung der verknüpften Ressource.
- `from` – Gibt die Namen der Ressourcen an, von denen die Traversierung gestartet werden darf. Dabei müssen die Ressourcen jeweils mit dem XLink-Attribut `label` ausgestattet werden. Achten Sie außerdem darauf, dass es sich um einen qualifizierten Namen handeln muss.
- `to` – Legt die Namen der Ressourcen fest, bei denen die Traversierung endet. Die entsprechenden Ressourcen müssen auch hier mit dem XLink-Attribut `label` versehen werden. Achten Sie außerdem darauf, dass es sich um einen qualifizierten Namen handeln muss.
- `show` – Legt fest, wie eine Anwendung dem entsprechenden Link folgen soll. Als Werte sind `embed` (der Inhalt der Ressource wird anstelle des Verweisinhalts angezeigt), `replace` (das Ziel soll im selben Fenster angezeigt werden), `new` (Das Ziel soll in einem neuen Fenster angezeigt werden.), `other` (Keine spezielle Angabe zum Ladezeitpunkt. Der Browser sollte aber nach Informationen im umgebenden Text suchen.) und `none` (Es wird überhaupt keine Darstellung bestimmt.).
- `title` – Gibt eine Beschreibung für den Link an. Zudem kann über `title` jedem Element eine Beschreibung zugewiesen werden.

Und auch hierzu wieder ein Beispiel:

Listing 5.39 Bestimmung der Traversierungsregeln.

```
<verweis xlink:type="extended">
   <extern xlink:type="locator"
         xlink:label="Homepage"
         xlink:href=http://www.hanser.de/
         xlink:title="Zur Homepage" />

   <extern xlink:type="locator" xlink:label="News"
        xlink:href="http://www.hanser.de/news.php"
         xlink:title="Zu den News" />

   <extern xlink:type="locator" xlink:label="DVD"
         xlink:href="http://www.hanser.de/dvd.php"
         xlink:title="Zu den DVDs" />

 <beschreibung xlink:type="arc" xlink:from="Homepage"
              xlink:to="News" xlink:show="new" />

 <beschreibung xlink:type="arc" xlink:from="Homepage"
               xlink:to="DVD" xlink:show="replace" />

</verweis>
```

An dieser Stelle noch einmal der Hinweis, dass alle Attribute optional sind. So könnten beispielsweise auch die beiden `to`- und `from`-Attribute weggelassen werden. In diesem Fall würde die Beschreibung dann für alle möglichen Traversierungen gelten, die von dieser Ressource zu einer der externen Ressourcen gehen.

```
<beschreibung xlink:type="arc" xlink:to="Start"
              xlink:actuate="onRequest" />
```

5.4.3.4 Den Titel angeben

Bei Elementen, die vom Typ `extended`, `locator` oder `arc` sind, können Sie anstelle des XLink-`title`-Attributs auch ein oder mehrere Kindelemente vom Typ `title` verwenden. In der Praxis sieht das folgendermaßen aus:

Listing 5.40 So werden die Titel angegeben.

```
<verweis xlink:type="extended">
   <text xlink:type="title">Bitte folgen</text>
   <text xlink:type="title">Hier geht's zur Startseite</text>
</verweis>
```

Durch diese Syntaxvariante lassen sich einem Element mehrere Titel zuweisen.

5.4.3.5 Mit Linkdatenbanken arbeiten

Bei Linkbases bzw. Linkdatenbanken handelt es sich um XML-Dokumente, in denen Links enthalten sind. Dadurch wird es möglich, Links separat zu speichern.

Da sich innerhalb des Dokuments kein Hinweis auf eine Linkdatenbank befindet, muss diese in dem Moment geladen werden, wenn das Quelldokument geöffnet wird. Zu diesem Zweck wird ein spezieller Wert für das `arcrole`-Attribut verwendet. Dieser Wert zeigt der Anwendung an, dass eine Linkbase existiert.

Listing 5.41 Verweise auf Linkdatenbanken

```
<link xlink:type="extended">
<doc xlink:type="locator"
```

```
      xlink:href="document.xml"
      xlink:label="document">

   <links xlink:type="locator"
      xlink:href="linkbase.xml"
      xlink:label="linkbase">

   <arc xlink:type="arc"
      xlink:from="document" xlink:to="linkbase"
      xlink:actuate="onLoad"
      xlink:arcrole="http://www.w3.org/1999/xlink/
      properties/linkbase">

   </link>
```

Dem `arcrole`-Attribut wird der Wert *http://www.w3.org/1999/xlink/properties/linkbase* übergeben. Die entsprechende Anwendung erkennt man daran, dass es sich bei der Ressource, auf die verwiesen wird, um eine Linkdatenbank handelt, und kann diese dann entsprechend verarbeiten.

Listing 5.42 So kann eine Linkdatenbank aussehen.

```
<link xmlns:xlink=http://www.w3.org/1999/xlink
xlink:type="extended">

<hier xlink:type="resource" xlink:label="a">A</hier>
<dort xlink:type="locator" xlink:href="http://www.b.de/"
xlink:label="b" xlink:title="Seite B" />

<dort xlink:type="locator" xlink:href="http://www.c.de/"
xlink:label="c" xlink:title="Seite C" />

<dort xlink:type="locator" xlink:href="http://www.d.de/"
xlink:label="d" xlink:title="Seite D" />

<dort xlink:type="locator" xlink:href="http://www.e.de/"
xlink:label="e" xlink:title="Seite E" />

<gehe xlink:type="arc" xlink:from="e" xlink:to="a" />
<gehe xlink:type="arc" xlink:from="e" xlink:to="d" />
<gehe xlink:type="arc" xlink:from="b" xlink:to="c" />
<gehe xlink:type="arc" xlink:from="b" xlink:to="d"/>
<gehe xlink:type="arc" xlink:from="c" xlink:to="b"/>
<gehe xlink:type="arc" xlink:from="c" xlink:to="e"/>
<gehe xlink:type="arc" xlink:from="a" />

<titel xlink:type="title">XLink</titel>

</link>
```

5.4.4 DTDs für XLinks definieren

Was DTDs sind, wie sie definiert und eingesetzt werden, haben Sie bereits gesehen. Nun lassen sich eben diese DTDs auch dafür verwenden, XLinks einfacher und übersichtlicher zu gestalten. Interessant ist das vor allem, wenn dieselbe Linkstruktur innerhalb eines Dokuments mehrmals verwendet werden soll.

Wie das funktioniert, wird hier anhand eines einfachen Links gezeigt. Normalerweise definiert man den folgendermaßen:

Listing 5.43 Eine DTD wird angegeben.

```
<meinlLink xmlns:xlink=http://www.w3.org/1999/xlink
   xlink:type="simple"
   xlink:href=http://www.hanser.de
   xlink:title="Neuigkeiten">
   Zu den News
</meinLink>
```

Will man diesen Link nun 20 Mal im Dokument verwenden, würde das den Code unnötig aufblähen. In solchen Fällen bietet sich der Einsatz einer DTD an. Für den gezeigten einfachen Link könnte sie folgendermaßen aussehen:

Listing 5.44 Das ist die DTD.

```
<!ELEMENT meinLink ANY>
<!ATTLIST meinLink
xmlns:xlink CDATA #FIXED
http://www.w3.org/1999/xlink
xlink:type (simple) #FIXED "simple"
xlink:href CDATA #IMPLIEDxlink:title CDATA #IMPLIED>
```

Die Definition des Links lässt sich dadurch vereinfachen.

Listing 5.45 Das ist die Linkdefinition.

```
<link xlink:href=http://www.hanser.de xlink:title="Neuigkeiten">
   Zu den News
</link>
```

Auf diese Weise spart man deutlich Tipparbeit und hält den Quellcode klein.

5.5 XML Base – eine Linkbasis schaffen

Vielleicht kennen Sie aus HTML das Element `base`, über das sich die Basisadresse für URIs spezifizieren lässt. Diese Basisadresse kann dann bei der Auflösung der innerhalb des Dokuments verwendeten relativen URIs genutzt werden. So wird es dem Webbrowser, der diese Informationen ausliest, ermöglicht, z.B. in Fehlersituationen besser auf verknüpfte oder referenzierte Dateien zugreifen zu können. In HTML sieht das dann folgendermaßen aus:

Listing 5.46 Das kennen Sie vielleicht aus HTML.

```
<head>
   <base href=http://www.hanser.de/ />
   <!-- ...  ... -->
</head>
```

Über das `href`-Attribut wird der URI der Datei angegeben. Nun einmal angenommen, dass innerhalb der HTML-Datei die folgende Grafikreferenz enthalten ist:

```
<img src="/images/logo.gif" />
```

Aufgrund des angegebenen Basis-URIs ermittelt der WWW-Browser diese Grafik mit dem absoluten URI.

```
<img src="http://www.hanser.de/images/logo.gif" />
```

Genau diesem Prinzip folgt die im Juni 2001 verabschiedete Empfehlung zu XML Base. Denn im Laufe der Zeit hatte sich herauskristallisiert, dass für jedes Element innerhalb eines XML-Dokuments ein Basis-URI angegeben werden kann, der wiederum im Kontext des Vaterelements interpretiert wird. Auf diese Weise ergibt sich eine viel feinere Aufgliederung der URI-Berechnung, die vor allem im Zusammenhang mit XLink-Anwendungen sinnvoll ist.

Abbildung 5.11 Die Spezifikation zu XML Base gibt es unter *http://www.w3.org/TR/xmlbase/*.

Bei XML Base handelt es sich um eine Spezifikation, die eine XML-Entsprechung des HTML-Elements `base` zur Verfügung stellt. In XML Base gibt es dafür das Attribut `xml:base`. Der Wert dieses Attributs wird als URI interpretiert. Das verwendete Präfix `xml` ist an den Namensraum *http://www.w3.org/19998/namespace* gebunden.

Zunächst ein einfaches Beispiel, das die Anwendung von `xml:base` auf ein Dokument zeigt, in dem XLinks enthalten sind.

Listing 5.47 Der Einsatz von `xml:base`

```
<?xml version="1.0"?>
   <doc xml:base=http://www.hanser.de/
     xmlns:xlink="http://www.w3.org/1999/xlink">
  <head>
    <title>Linkliste</title>
  </head>
  <body>
    <absatz>Linkliste:
```

```
        <link xlink:type="simple"
        xlink:href="news.html">Neuigkeiten</link>!
      </absatz>

      <liste xml:base="/links/">

        <item>
          <link xlink:type="simple" xlink:href="books.html">
             Books
          </link>
        </item>

        <item>
          <link xlink:type="simple" xlink:href="software.html">
             Software
          </link>
        </item>

        <item>
          <link xlink:type="simple" xlink:href="dvd.html">DVD</link>
        </item>

      </listet>

    </body>

  </doc>
```

Die in diesem Dokument verwendeten Links werden folgendermaßen zu vollständigen URIs aufgelöst:

- `Neuigkeiten – http://www.hanser.de/news.html`
- `Books – http://www.hanser.de/links/books.html`
- `Software – http://www.hanser.de/links/software.html`
- `DVD – http://www.hanser.de/links/dvd.html`

Ein weiteres Beispiel für den Einsatz von XML Base zeigt die folgende Syntax. Dabei soll das Dokument unter dem URI *ftp://hanser.de/content.xml* verfügbar sein. Innerhalb des Dokuments wurden einige XLinks definiert. Der Einfachheit halber wird davon ausgegangen, dass die notwendigen Typ-Attribute für die Elementtypen in einer DTD definiert wurden.

Listing 5.48 Auch das ist möglich.

```
<basis xml:base="content">
   <extended xml:base="neuigkeiten">
      <locator xlink:href="books.html" />
      <locator xlink:href="icon.png" xml:base="../images/" />
     <locator xlink:href="annotations" />
   </extended>

   <simple xlink:href="dvd.html" />

   <extended xml:base="/">
      <locator xlink:href="next" xml:base="neuigkeiten.html#news" />
      <locator xlink:href="TR/xmlbase/"
      xml:base="http://www.w3.org/" />
    </extended>

</basis>
```

Das Beispiel enthält das Element `basis`, in dem das Attribut `xml:base` steht. Zusammen mit dem URI des Dokuments ergibt sich der Basis-URI *ftp://hanser.de/content/*. Dabei

werden sämtliche Pfadbestandteile entfernt, die hinter dem letzten Schrägstrich stehen. Wenn Sie also einen Basis-URI angeben, dann beachten Sie bei den Pfadbestandteilen, die nicht entfernt werden sollen, dass diese immer durch einen Schrägstrich abgeschlossen werden müssen.

6

6 Ausgabe mit CSS

XML selbst ist eine Sprache für die Strukturierung von Daten. Für die Ausgabe bzw. Formatierung kann man auf verschiedene Stylesheet-Sprachen zurückgreifen. Als erste der dabei möglichen Varianten werden in diesem Buch die Cascading Stylesheets (CSS) vorgestellt, schließlich gehört CSS zu den Sprachen, die in der Webentwicklergemeinde am häufigsten verwendet wird. Nun wird CSS natürlich hauptsächlich im Zusammenspiel mit HTML eingesetzt. Allerdings muss der Einsatz darauf nicht beschränkt bleiben. Ebenso lässt sich CSS nämlich auch auf XML-Dateien bzw. XML-Elemente anwenden. Dieses Kapitel zeigt, wie das funktioniert, und klärt dann auch über die Grenzen dieser XML-CSS-Kombination auf.

6.1 Schnelleinstieg in CSS

Dank CSS und XML können Inhalt und Form von Dokumenten getrennt werden. Das bedeutet zunächst natürlich einen riesigen Vorteil, schließlich kann dieselbe Quelldatei in ganz unterschiedlichen Varianten ausgegeben werden. So lässt sich z.B. die gleiche Ausgangsdatei am normalen Monitor und am Handy in einem jeweils völlig anderen Design anzeigen.

An dieser Stelle wird nur kurz auf die CSS-Grundlagen eingegangen. Hier steht vielmehr der Einsatz von CSS im Zusammenhang mit XML im Vordergrund. Einen guten Einstieg in die CSS-Thematik liefert aber z.B. die Seite CSS 4 You (*http://www.css4you.de/*).

Das CSS-Prinzip lässt sich vielleicht am ehesten mit Formatvorlagen vergleichen. Formatvorlagen kennen Sie aus Textverarbeitungsprogrammen. Dort dienen sie der einheitlichen Gestaltung und Formatierung von Dokumenten. Durch den Einsatz von Formatvorlagen ist es möglich, bestimmte Textpassagen in einem speziellen Design darzustellen. Je nach Wunsch lassen sich die Formatvorlagen jederzeit anpassen. Die gemachten Änderungen werden dann automatisch für das gesamte Dokument übernommen.

Die beiden Begriffe Dokumentvorlage und Formatvorlage werden übrigens gerne verwechselt. Dabei ist der Unterschied eigentlich ganz einfach zu erklären: In einer Doku-

mentvorlage werden allgemeine Informationen über die Datei zusammengefasst. Dabei handelt es sich zum Beispiel um die Randabstände oder den kompletten Satz der Formatvorlagen. Auf CSS angewandt könnte man also sagen, dass eine CSS-Datei die Dokumentvorlage für ein XML-Dokument ist. Eine Formatvorlage wiederum beschreibt, wie ein bestimmtes Element (Textabsatz, Überschrift etc.) aussehen soll.

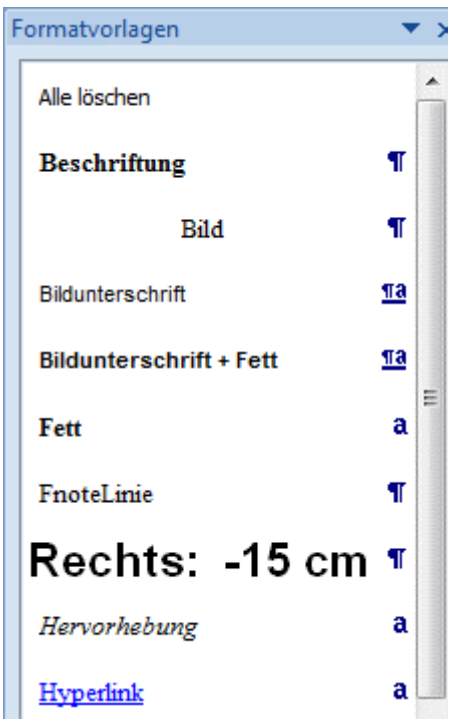

Abbildung 6.1 Einige Formatvorlagen in Word.

Was Formatvorlagen im Bereich Textverarbeitung sind, das ist CSS für HTML und XML. Denn genau genommen handelt es sich bei den Stylesheets um frei definierbare Formatvorlagen. Auf ein XML-Dokument umgemünzt bedeutet das: Sie können für XML-Elemente jeweils eine Formatvorlage definieren und so alle Elemente des gleichen Typs formatieren. Werden später Formatänderungen nötig, müssen diese nur an einer zentralen Stelle, nämlich im Stylesheet, vorgenommen werden.

6.1.1 CSS-Maßeinheiten

Beim Umgang mit CSS werden Sie immer wieder Maßeinheiten einsetzen. Dieser Abschnitt zeigt, welche Maßeinheiten zur Verfügung stehen und welche Auswirkungen diese jeweils haben. Denn es ist von großer Bedeutung, ob man z.B. die Schriftgröße in px oder pt angibt.

Tabelle 6.1: Relative Längenangaben

Einheit	Beschreibung	Beispiel
`em`	Bezogen auf die Schriftgröße des Elements. Ausnahme: Wird die Schriftgröße (`font-size`) mit `em` definiert, ist sie bezogen auf die Schriftgröße des Elternelements.	`font-size: 0.5em;`
`ex`	Bezogen auf die Höhe des Kleinbuchstabens `x` in diesem Element. Ausnahme: Wird die Schriftgröße (`font-size`) selbst mit `ex` definiert, ist sie bezogen auf die Höhe des Kleinbuchstabens `x` im Elternelement.	`font-size: 2.3ex;`
`px`	Hierüber wird die Pixelanzahl angegeben. Die tatsächliche Größe ist abhängig von der Pixeldichte des Ausgabegeräts.	`border-width: 2px;`

Tabelle 6.2: Absolute Längenangaben

Einheit	Beschreibung	Beispiel
`cm`	Zentimeter	`font-size: 1cm;`
`in`	Steht für Inch. 1 Inch entspricht 2,54 Zentimetern.	`margin-left: 2in;`
`mm`	Millimeter	`border-width: 3mm;`
`pc`	Diese Abkürzung steht für die typografische Maßeinheit Pica. 1 Pica entspricht 12 Punkt.	`font-size: 4pc;`
`pt`	Steht für die typografische Maßeinheit Punkt. 1 Punkt entspricht 0,0138 Inch oder 0,3527 Millimeter.	`line-height: 14pt;`

Tabelle 6.3: Prozentuale Angaben

Einheit	Beschreibung	Beispiel
`%`	Je nach verwendeter CSS-Eigenschaft relativ zur elementeigenen Größe oder zu der des Elternelements oder einem allgemeinen Kontext.	`line-height: 120%;`

Tabelle 6.4: Winkel

Einheit	Beschreibung	Beispiel
`deg`	Grad (Kreis = 360 deg)	`azimuth: 70deg;`
`grad`	Neugrad (Kreis = 400 grad)	`azimuth: 90grad;`
`rad`	Radiant (1 rad = 57.296 deg)	`azimuth: 20rad;`

Tabelle 6.5: Zahlenwerte

Einheit	Beispiel
Positive Ganzzahl	`24`
Negative Ganzzahl	`-30`
Positive reelle Zahl	`2.6`
Negative reelle Zahl	`-4.3`

6.1.2 Mit Selektoren arbeiten

Für die Zuordnung eines Stylesheets zu einem Elemente gibt es in CSS verschiedene Typen von Selektoren, von denen die wichtigsten auf den folgenden Seiten vorgestellt werden. Für die Erklärung der verschiedenen Selektor-Varianten wird immer auf eine einfache XML-Struktur zurückgegriffen.

Listing 6.1 Das ist das Ausgangsdokument.

```
<adresse>
   <anrede>Herr</anrede>
   <name>Michael Meyer</name>
   <strasse>Rheinsberger Straße 12</strasse>
   <plz>10436</plz>
   <ort>Berlin</ort>
</adresse>
```

Selektoren filtern Elemente über Suchmuster, die wiederum miteinander kombiniert werden können. Auf diese Weise lassen sich aus sehr einfachen Selektoren komplexe Filter erzeugen. Stimmen die definierten Filter mit einem Element überein, können die angegebenen Regeln darauf angewendet werden.

6.1.2.1 Element-Selektor

Über den Element-Selektor können allen Elementen vom selben Typ im aktuellen Dokument die gewünschten Eigenschaften zugewiesen werden.

```
anrede {color: blue;}
```

Durch diese Syntax wird für alle `anrede`-Elemente Blau als Hintergrundfarbe bestimmt.

6.1.2.2 Klassen-Selektor

Die Funktionsweise des Element-Selektors ist zwar in vielen Fällen gewünscht, sonderlich flexibel ist man damit allerdings nicht. Oft ist es besser, auf Klassen-Selektoren zurückzugreifen. Dabei werden die CSS-Eigenschaften nicht an ein spezielles Element gebunden, sondern „global“ angelegt. Die Klasse kann dann auf verschiedene XML-Elemente angewendet werden. Einzige Voraussetzung dafür: Den betreffenden XML-Elementen muss man das `class`-Attribut zuweisen.

Listing 6.2 Eine Klasse wurde zugewiesen.

```
<adresse>
   <anrede class="blau">Herr</anrede>
   <name>Michael Meyer</name>
   <strasse>Rheinsberger Straße 12</strasse>
   <plz class="blau">10436</plz>
   <ort>Berlin</ort>
</adresse>
```

Die Klassen-Definition sieht dann folgendermaßen aus:

```
.blau {color: blue;}
```

Vor dem beim `class`-Attribut angegebenen Klassennamen muss ein Punkt notiert werden. Im folgenden Beispiel werden zwei Klassen-Selektoren kombiniert. Dadurch werden Elemente angesprochen, die sowohl zur Klasse `blau` als auch zur Klasse `fett` gehören.

```
.blau.fett {color: blue;}
```

6.1.2.3 ID-Selektor

Auf ähnliche Weise lassen sich die ID-Selektoren einsetzen. Allerdings werden hiermit ausschließlich solche Elemente formatiert, die dateiweit mit einer eindeutigen ID gekennzeichnet wurden.[1] Im aktuellen Beispiel könnte das zum Beispiel der Fall sein, wenn Sie die Namen der einzelnen Personen unterschiedlich formatieren wollen. Auch hierzu wieder ein Beispiel:

Listing 6.3 So sieht das Ausgangsdokument aus.

```
<adresse>
   <anrede>Herr</anrede>
   <name id="meyer">Michael Meyer</name>
   <strasse>Rheinsberger Straße 12</strasse>
   <plz >10436</plz>
   <ort>Berlin</ort>
</adresse>

<adresse>
   <anrede>Herr</anrede>
   <name id="kruger">Friedhelm Kruger</name>
   <strasse>Elbgaustraße 15</strasse>
   <plz >22524</plz>
   <ort>Hamburg</ort>
</adresse>
```

Den betreffenden Elementen wird das `id`-Attribut mit einem eindeutigen Wert zugewiesen. Um auf die mit `id` gekennzeichneten Elemente mittels CSS zuzugreifen, wird die folgende Syntax verwendet:

```
#meyer {color: blue;}
#kruger {color: red;}
```

Dem ID-Namen wird also das Zeichen `#` vorangestellt.

Das `id`-Attribut wird leider immer wieder falsch eingesetzt. Es sollte lediglich dort verwendet werden, wo ein Element dateiweit eindeutig gekennzeichnet werden soll. Für andere Zwecke verwendet man das `class`-Attribut.

6.1.2.4 Attribut-Selektor

Wenn Elemente Attribute besitzen, können diese Attribute ebenfalls als Selektor verwendet werden.

Zum besseren Verständnis zunächst ein Beispiel: Angenommen, das Element `buch` besitzt das Attribut `seitenzahl`. Dann sähe das Element folgendermaßen aus:

```
<buch seitenzahl="212">
```

[1] Zusätzlich werden mit dem #-Attribut Elemente als Zielanker einer Verweisdefinition gekennzeichnet.

Mittels des Attribut-Selektors können Sie nun z.B. alle `buch`-Elemente auswählen, die das `seitenzahl`-Attribut besitzen. Die entsprechende Syntax sieht so aus:

```
buch[seitenzahl]
```

Welche Möglichkeiten es noch gibt, zeigt die folgende Übersicht:

- `[att]` – Das Element muss nur das Attribut enthalten. Ob auch ein Wert übergeben wird, ist unerheblich.
- `a[href]` – Es werden alle Hyperlinks (`<a href=...>`) markiert. Auf Anker-Definitionen (`<a name=...>`) trifft das jedoch nicht zu. Denn Anker werden ohne `href`-Attribut definiert.
- `[align=left]` – Markiert alle Elemente, deren Attribut `align` den Wert `left` besitzt.
- `[attr~=wert]` – Es werden all die Elemente markiert, in denen `wert` in einer durch Leerzeichen getrennten Werteliste enthalten ist.
- `[attr|=wert]` – Markiert ein Element, wenn innerhalb des Attributs der angegebene Wert am Anfang einer durch Bindestrich getrennten Zeichenfolge steht.
- `img[width="200"]` – Hier werden alle Grafiken markiert, die 200 Pixel breit sind.

Das folgende Beispiel zeigt noch einmal den Einsatz eines Attribut-Selektors. Zunächst die XML-Syntax.

Listing 6.4 Das ist die XML-Struktur.

```
<?xml version="1.0"?>
<?xml-stylesheet href="attribute.css" type="text/css"?>
<!DOCTYPE beitrag [
  <!ELEMENT beitrag (ueberschrift, text+)>
  <!ELEMENT ueberschrift (#PCDATA)>
  <!ATTLIST ueberschrift format CDATA #FIXED "farbig">
  <!ELEMENT text (#PCDATA)>
]>

<beitrag>
 <ueberschrift format="farbig">Hallo, Welt</ueberschrift>
 <inhalt>Das ist der eigentliche Text</inhalt>
</beitrag>
```

Die entsprechende CSS-Datei sieht folgendermaßen aus:

Listing 6.5 Die Formatierungen werden festgelegt.

```
ueberschrift[format] {
   color:blue;
   font-weight:bold;
   display:block;
}
```

Leider werden die meisten der Attribut-Selektoren bislang von den Browsern nicht unterstützt. Allerdings können zumindest Safari, Opera, Mozilla und iCab die ersten beiden Varianten interpretieren. **Abbildung 6.2** zeigt, wie das Beispiel vom Firefox interpretiert wird.

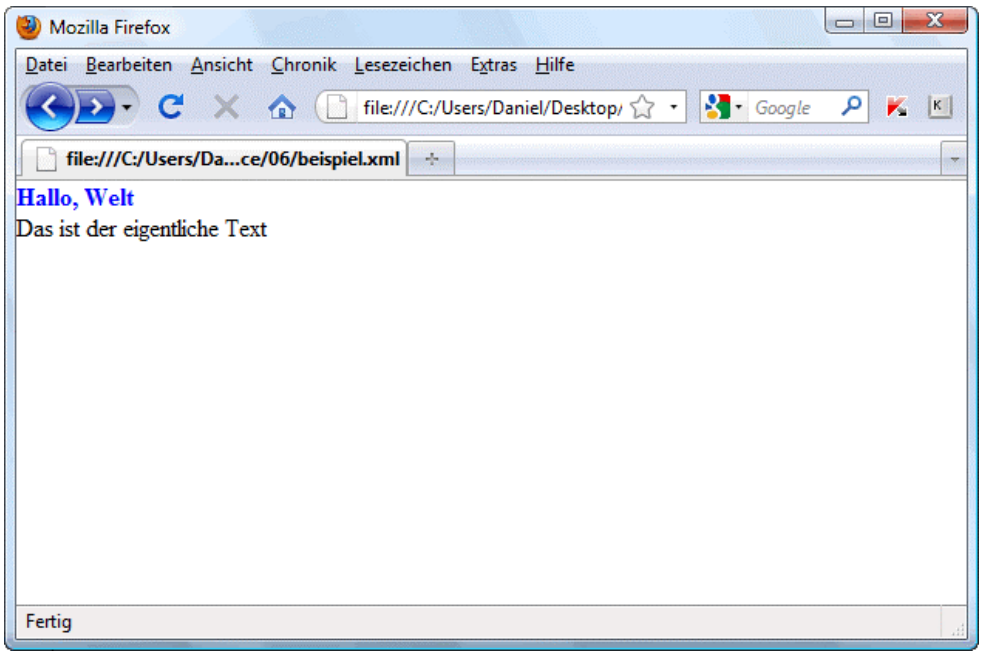

Abbildung 6.2 Der Einsatz eines Attribut-Selektors im Firefox

6.1.2.5 Universal-Selektor

Mit dem Universal-Selektor lässt sich jedes vorhandene Element mit einer Stylesheet-Definition verknüpfen. Dieser Selektor ist überall dort nützlich, wo man mehreren Elementen die gleichen Eigenschaften zuweisen will.

Listing 6.6 So wird der Universal-Selektor eingesetzt.

```
* {
   font-size: 14px;
   color: #ff0000;
}
```

Da der Universal-Selektor auf alle Elemente zutrifft, kann er meistens weggelassen werden. Folgende Syntax ist mir der vorherigen somit identisch:

Listing 6.7 Das bewirkt das gleiche Ergebnis.

```
{
   font-size: 14px;
   color: #ff0000;
}
```

Der Internet Explorer bis Version 6 entfernt automatisch alle führenden * eines Selektors. Erst danach sucht er die passenden Elemente. Somit wird `* html` also genauso wie ein alleinstehendes `html` behandelt. Genau dieses Verhalten ist es, das man für CSS-Hacks nutzen kann.

6.1.2.6 Kind-Selektor

Mit Kind-Selektoren können Kindelemente angesprochen werden. Innerhalb eines XML-Dokuments handelt es sich dabei um Elemente, die zwischen den Anfangs- und einem End-Tag eines anderen Elements liegen.

Listing 6.8 Das ist das Ausgangsdokument.

```
<autor>
   <name>Tony Parsons</name>
</autor>
```

In diesem Beispiel ist `name` das Kindelement von `autor`. Durch die folgende Syntax wird festgelegt, dass `name`-Elemente, die direkt innerhalb eines `autor`-Elements stehen, mit einem roten Hintergrund ausgestattet werden.

Listing 6.9 Es wird auf das `name`-Element zugegriffen.

```
autor>name {
   background: red;
}
```

6.1.2.7 Folgeelement-Selektor

Mit dieser Selektor-Variante wird ein Element markiert, das unmittelbar auf ein anderes folgt und das gleiche Elternelement besitzt.

Durch die folgende Syntax wird der obere Abstand eines `name`-Elements auf vier Pixel festgelegt, wenn dieses direkt unterhalb eines `autor`-Elements steht.

```
autor+name { margin-top: 4px; }
```

6.1.3 Das Prinzip der Kaskade

Eines der Hauptprinzipien im Zusammenhang mit den Stylesheets ist die Kaskade. Nur wenn man dieses Prinzip versteht, kann man später Stylesheets effektiv entwickeln und eingreifen, wenn etwas nicht wie gewünscht formatiert wird.

Innerhalb von Stylesheets passiert es immer mal wieder, dass in den notierten CSS-Regeln Konflikte auftreten, also verschiedene Deklarationen auf ein und dasselbe Element zutreffen. Ein typisches Beispiel dafür ist, wenn Sie für das Element `name` einmal eine blaue und einmal eine rote Vordergrundfarbe angegeben haben.

Listing 6.10 Die CSS-Definitionen widersprechen sich.

```
name {
   color: blaue;
}

[...]

name {
   color: red;
}
```

Tatsächlich treten solche doppelten Definitionen immer mal wieder in CSS-Dateien auf. In diesem Fall muss der Browser entscheiden, in welcher Farbe er das Element anzeigt.

Das Prinzip der Kaskade wird besonders deutlich, wenn Sie sich die Herkunft des Wortes Kaskade ansehen. Denn bei einer Kaskade handelt es sich im Allgemeinen um einen mehrstufigen Wasserfall. Und wie bei einem solchen Wasserfall laufen auch die verschiedenen Deklarationen eines Stylesheets über mehrere Stufen. In CSS lässt sich durch die Anweisung `!important` der Wasserfallverlauf beeinflussen. Dazu aber später mehr.

Die Kaskade kennt mehrere Stufen.

6.1.3.1 Stufe 1

In dieser Stufe werden alle Deklarationen gefunden, die für das aktuelle Element infrage kommen. Das ist dann der Fall, wenn der innerhalb der Deklaration verwendete Selektor und das Element übereinstimmen.

- Wenn es nur eine Deklaration für die Element-Eigenschaft-Kombination gibt, wird diese angewendet.
- Gibt es mehr als eine Deklaration für die Element-Eigenschaft-Kombination, wird zur 2. Stufe gegangen.

6.1.3.2 Stufe 2

Hier werden die vorhandenen Deklarationen nach ihrer Herkunft und ihrer Wichtigkeit sortiert. Die Wichtigkeit wiederum wird von der `!important`-Anweisung bestimmt.

Die Sortierreihenfolge richtet sich dabei nach folgenden Regeln:

1. Benutzer-Stylesheet ist in der `!important` enthalten.
2. Deklaration auf der Webseite, in der `!important` enthalten ist.
3. Deklaration auf der Webseite, die `!important` nicht enthält.
4. Benutzer-Stylesheet, in dem `!important` nicht enthalten ist.
5. Stylesheet des Anwenderprogramms (Browser etc.)

Folgende Entscheidung wird getroffen:

- Wenn es für die aktuelle Element-Eigenschaft-Kombination nur Deklarationen mit unterschiedlicher Wichtigkeit gibt, wird die Deklaration mit der höchsten Wichtigkeit angewendet. Anschließend wird zum nächsten Element gegangen.
- Existieren mehrere Deklarationen mit der gleichen Gewichtung für die aktuelle Element-Eigenschaft-Kombination, wird mit Stufe 3 fortgefahren.

6.1.3.3 Stufe 3

In der 3. Stufe werden die Selektoren nach ihrer Spezifität sortiert. (Was es mit dieser Spezifität genau auf sich hat, wird im weiteren Verlauf dieses Kapitels noch gezeigt.)

- Wenn es mehrere Deklarationen unterschiedlicher Spezifität für die aktuelle Element-Eigenschaft-Kombination gibt, wird die Deklaration mit der höchsten Spezifität angewendet.
- Sollte es für die aktuelle Element-Eigenschaft-Kombination mehrere Deklarationen mit der gleichen Spezifität geben, wird mit Schritt 4 fortgefahren.

6.1.3.4 Stufe 4

Hier wird nach der Reihenfolge des Auftretens sortiert. Sollten zwei Stylesheet-Regeln das gleiche Gewicht, den gleichen Ursprung und die gleich hohe Spezifität besitzen, wird die zuerst notierte Regel bevorzugt behandelt. Dabei werden die importierten Stylesheets immer bevorzugt vor den im Head-Bereich stehenden Angaben behandelt.

6.1.4 Die Spezifität ermitteln

Es wurde bereits auf den Aspekt der Spezifität hingewiesen. Dabei werden zunächst sämtliche Selektoren in ihre Bestandteile zerlegt und in verschiedene Kategorien aufgeteilt.

- A – Erhält den Wert `1`, wenn CSS-Deklarationen dem Element über das `style`-Attribut zugewiesen wurden.
- B – Entspricht der Anzahl der selektierten `id`-Attribute.
- C – Entspricht der Anzahl der selektierten anderen Attribute wie zum Beispiel Pseudo-Klassen.
- D – Entspricht der Anzahl der selektierten Elementnamen und Pseudo-Elemente.

In Tabelle 6.6 sind zum besseren Verständnis einige Beispiele enthalten. Dabei ist die Kategorisierung der Bestandteile der Selektoren in absteigender Reihenfolge aufgeführt.

Tabelle 6.6: Ein Beispiel für die Ermittlung der Spezifität

Selektor	A	B	C	D
`style="…"`	1	0	0	0
`#nav a.xy`	0	1	1	1
`#nav li a`	0	1	0	2
`ul li .xy`	0	0	1	2
`a:link`	0	0	1	1
`li a`	0	0	0	2
`a:first-line`	0	0	0	2

In diesem Beispiel besitzt der Selektor `#nav a.xy` den höchsten Wert. Er überschreibt somit alle anderen Deklarationen. Die sich daran anschließenden Selektoren `#nav li a` und `li a` werden übergangen, da sie eine niedrigere Spezifität besitzen.

Ist in einem Stylesheet lediglich ein Selektor mit der höchsten ermittelten Spezifität enthalten, wird die angegebene Regel angewendet. Existieren hingegen zwei oder mehrere Selektoren mit der gleichen Gewichtung, unterscheidet die Reihenfolge des Vorkommens. Dabei werden die später angegebenen Selektoren von den vorherigen überschrieben.

6.1.5 Ein Beispiel für das Vererbungsprinzip

Das Prinzip der Kaskade ist also recht komplex. Die Erfahrung zeigt aber, dass sich dieser Aspekt sehr gut anhand eines Beispiels zeigen lässt und dann auch verständlicher wird. Hier zunächst der entsprechende Quellcode:

Listing 6.11 Eine typische XHTML-CSS-Kombination

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Die Kaskade</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<style type="text/css">
<!--
body{
   font-family: Arial, sans-serif;
   font-size: 12px
}

p{
   color: black
}

h1{
   font-size: 130%;
   font-weight: normal
}

b{
   color: gray;
   font-weight: bold
}

h1 b{
   color: silver
}


.wichtig b{
   color: gray;
   font-weight: bold;
   text-decoration: underline
}

#text b em{
   color: red;
   font-weight: bold;
   font-style: italic
}
-->
</style>
</head>
<body>
<p>Ich bin ein normaler Absatz</p>
<p>Ein normaler Absatz mit einem <b>fetten</b> Wort.</p>
<h1>Eine Überschrift mit einem <b>fetten</b> Wort</h1>
<p class="wichtig">Ich bin nicht nur wichtig und <b>fett</b> , sondern
```

```
auch noch unterstrichen.</p>
<h1 class="wichtig">Ich bin als eine <b>wichtige &Uuml;berschrift</b>
formatiert.</h1>
<div id="text">Ich bin ein Text mit <b>fetten und
<em>hervorgehobene</em></b> Wörtern.</div>
</body>
</html>
```

Im Browser stellt sich das Ergebnis dann so dar, wie es auf **Abbildung 6.3** zu sehen ist.

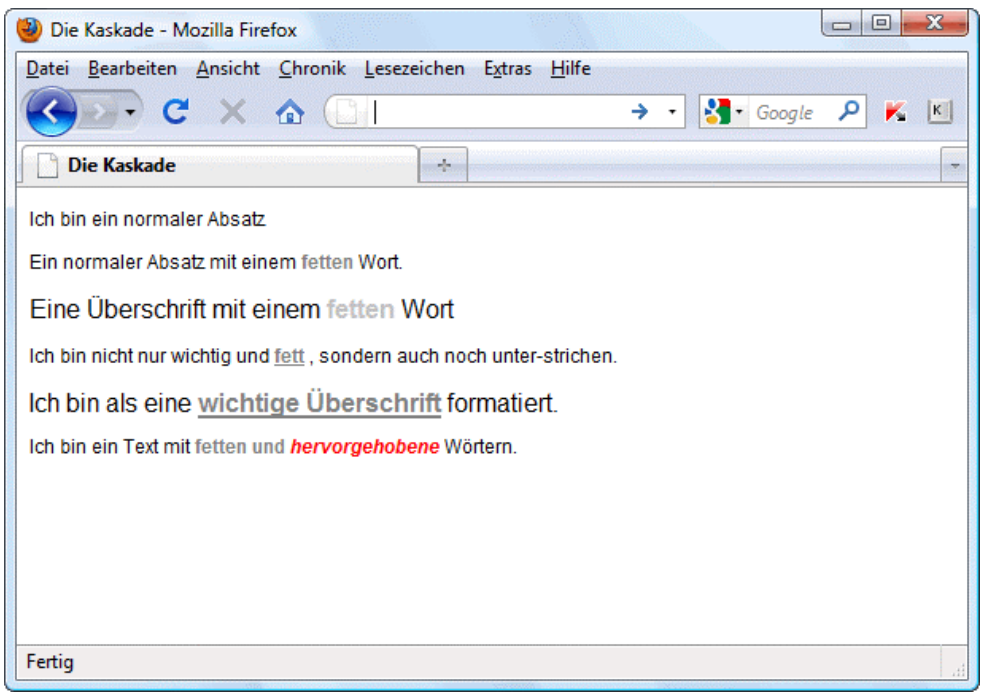

Abbildung 6.3 Mit zunehmender Anzahl an Formaten wird es komplizierter.

Interessant ist jetzt natürlich die Frage, wie sich die Rangfolge der Selektoren für dieses Beispiel ermitteln lässt.

- Für das `body`-Element werden Arial als Schriftart und 12 Pixel als Schriftgröße festgelegt. Diese Eigenschaften werden auf alle untergeordneten Elemente angewendet, solange es keine widersprüchlichen Anweisungen gibt.
- Überschriften der 1. Ordnung erhalten die Schriftgröße 130 Prozent und ein normales Schriftgewicht.
- Die mit `b` ausgezeichneten Bereiche werden in grauer Schrift angezeigt und fett gekennzeichnet. Wenn sich `b` allerdings innerhalb einer `h1`-Überschrift befindet, wird `silver` als Schriftfarbe verwendet. Bestimmt wurde das durch die Anweisung `h1 b`.
- Die mittels `.wichtig b` ausgezeichneten Passagen werden höherwertig als normale Abschnitte, die mit `b` und `h1` gekennzeichnet sind, betrachtet. Alle mit `.wichtig b` markierten Texte werden zudem unterstrichen angezeigt.

- Der mit `<div id="text">` markierte Abschnitt enthält einen `b`-Bereich, der zunächst auch wie ein `b`-Bereich formatiert wird. Allerdings sorgt die `em`-Anweisung dafür, dass die CSS-Definition `#text b em` angewendet wird. Dadurch werden die vorherigen Definitionen überschrieben.

Dieses Beispiel macht noch einmal deutlich, wie komplex die Regeln sind, denen Stylesheets unterliegen. Es kann und wird Ihnen also immer mal wieder passieren, dass Elemente anders dargestellt werden, als Sie das wollen und als es auf den ersten Blick innerhalb der Stylesheet-Definition angegeben ist. Um solche Fehler zu beheben, gehen Sie das Stylesheet am besten von oben nach unten durch und kommentieren „fragwürdige“ Passagen vorübergehend aus.

6.2 XML mit CSS formatieren

Nach dem vorherigen kurzen Ausflug in die relevanten CSS-Grundlagen, geht es nun darum, wie sich XML-Dokumente mittels Stylesheets formatieren lassen. Der erste Schritt – natürlich neben der Definition der XML- und CSS-Dateien – ist, das XML-Dokument mit einem Stylesheet zu verknüpfen. Dafür wird im Prolog der XML-Datei eine Verarbeitungsanweisung – die sogenannte Processing Instruction – eingefügt.

Listing 6.12 So sieht die Grundstruktur aus.

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/css" href="adressen.css" ?>
[...]
```

Über das `href`-Attribut gibt man den Pfad zur CSS-Datei an. Anders als im aktuellen Beispiel können Sie übrigens auch mehrere Stylesheets einbinden. Auch dabei müssen dann alle Angaben an einer beliebigen Stelle innerhalb des Prologs, aber in jedem Fall noch vor dem Wurzelelement des Dokuments notiert werden.

Innerhalb der Verarbeitungsanweisung können verschiedene Pseudoattribute notiert werden. In welcher Reihenfolge diese angegeben werden, ist nicht vorgeschrieben. Hier die möglichen Attribute:

- `href CDATA #REQUIRED`
- `type CDATA #REQUIRED`
- `title CDATA #IMPLIED`
- `media CDATA #IMPLIED`
- `charset CDATA #IMPLIED`
- `alternate (yes|no) "no"`

Die mit `REQUIRED` gekennzeichneten Attribute müssen angegeben werden. Die Bedeutung von `href` haben Sie bereits kennengelernt. Mit `type` wird der MIME-Typ des Stylesheets beschrieben. Handelt es sich um ein Cascading Stylesheet, wird der Wert `text/css` erwartet.

Interessant ist auch das `media`-Attribut. Denn hierüber lassen sich Formate für unterschiedliche Ausgabemedien definieren. Ein XML-Dokument kann mittels `media` also für Drucker, Handy, Bildschirm usw. optimiert werden. Das entsprechende Ausgabegerät sollte anhand des angegebenen Medientyps erkennen, welche CSS-Datei zu laden ist.

- `all` – Die CSS-Datei gilt für alle Medientypen.
- `aural` – Die CSS-Datei gilt für Sprachausgabesysteme. Was vor einiger Zeit noch wie Zukunftsmusik klang, lässt sich mittlerweile praktisch einsetzen. Als erster Browser beherrscht Opera die Sprachausgabe. Dazu wird das von IBM entwickelte Sprachmodul durch Einstellungen in Opera (*Extras*/*Einstellungen*/*Erweitert*/*Sprache*/ *Sprachgesteuerte Bedienung aktivieren*) aktiviert. Um die Sprachausgabe zu testen, können Sie sich über die Anweisung `<p style="speak:spell-out">Diesen Text buchstabieren</p>` den so ausgezeichneten Text buchstabieren lassen. In CSS 2.1 wird `aural` missbilligt, da die Eigenschaft in CSS3 durch `speech` ersetzt wird.
- `braille` – Die CSS-Datei gilt für Ausgabegeräte mit sogenannter Braille-Zeile.
- `embossed` – Die CSS-Datei gilt für Braille-Drucker. Bei diesen Druckern wird der Text in das Papier eingestanzt. Geeignet ist das zum Beispiel für Blindenschrift.
- `handheld` – Die Anzeige ist für Handheld-PCs gedacht.
- `print` – Das Stylesheets ist für den Ausdruck auf Papier bestimmt. WWW-Browser verwenden diese Formatdefinitionen, wenn der Besucher eine Seite ausdrucken will.
- `projection` – Die CSS-Datei wird für die Projektion von Daten mittels Beamern und Ähnlichem verwendet.
- `screen` – Das Stylesheet ist für die Anzeige am Bildschirm gedacht.
- `tty` – Die CSS-Datei gilt für nicht grafische Ausgabegeräte mit einer festen Zeilenbreite. Zu dieser Gattung gehören z.B. Fernschreiber. Ebenso sind diese Stylesheets aber auch für reine Text-Browser interessant.
- `tv` – Das Stylesheet sollte für Fernseher und Ähnliches optimiert sein. Die Geräte zeichnen sich durch grobe Bildschirmauflösung und mangelnde Scroll-Unterstützung aus.

Sie können für eine Seite mehrere Stylesheets anbieten. Wenn es der Browser des Anwenders erlaubt, kann der Besucher aus den angebotenen Stylesheets das für ihn passende auswählen.

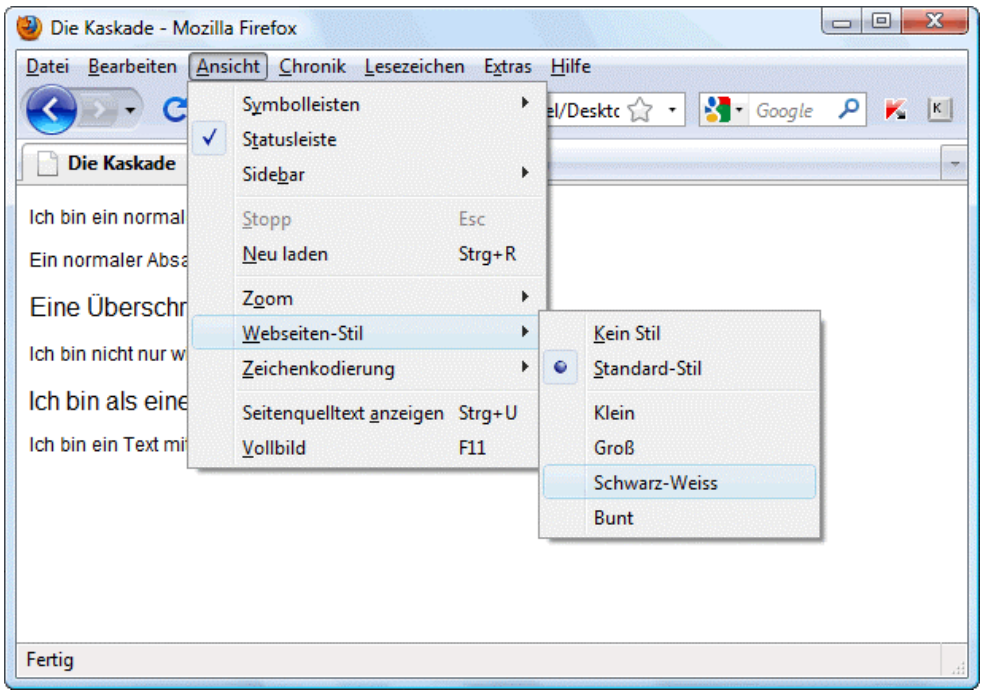

Abbildung 6.4 Das gewünschte Stylesheet kann ausgewählt werden.

Im Firefox sind die alternativen Stylesheets über *Ansicht/Webseiten-Stil* abrufbar.

Listing 6.13 Mehrere Stylesheets stehen zur Auswahl.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<?xml-stylesheet type="text/css" href="adressen.css" ?>
<?xml-stylesheet alternate="yes" href="mittel.css" title="Mittel"
type="text/css"?>
<?xml-stylesheet alternate="yes" href="gross.css" title="Gross"
type="text/css"?>
```

Wie bei allen Verarbeitungsanweisungen wird ein XML-Prozessor, der Stylesheets nicht auswerten kann, die Angaben ignorieren.

6.2.1 XML-Elemente formatieren

Die Elemente eines XML-Dokuments lassen sich mit CSS denkbar einfach gestalten. Wie das funktioniert, lässt sich am besten anhand eines Beispiels zeigen. Dabei wird als Basis ein typisches XML-Dokument genommen.

Listing 6.14 Das ist das Ausgangsdokument.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<?xml-stylesheet type="text/css" href="adressen.css" ?>
<adressen_db>

   <adresse>
      <anrede>Herr</anrede>
```

```
      <name>Michael Meyer</name>
      <strasse>Rheinsberger Straße 12</strasse>
      <plz>10436</plz>
      <ort>Berlin</ort>
   </adresse>

   <adresse>
      <anrede>Herr</anrede>
      <name>Fred Krüger</name>
     <strasse>Elbgaustraße 15</strasse>
      <plz>10436</plz>
      <ort>Hamburg</ort>
   </adresse>

</adressen_db>
```

Über `<?xml-stylesheet type="text/css" href="adressen.css" ?>` wird die Stylesheet-Datei `adressen.css` eingebunden. Innerhalb dieser Datei werden nun die in der XML-Datei vorhandenen Elemente formatiert. Das funktioniert genauso wie bei der Formatierung von HTML-Elementen.

Listing 6.15 Und so sieht das Stylesheet aus.

```
adressen_db {
   position:absolute;
   top: 20;
   left: 40;
   padding: 3;
   border: 1 dotted black;
   font-family: Arial;
   font-size: 12pt;
}

adresse {
   width: 150;
   padding-top: 5;
   padding-bottom: 5;
   position: relative;
   background-color: #f5f5f5;
   display: block;
}

anrede {
   display: block;
   font-weight: bold;

}

name {
   font-weight: bold;
   display: block;
}

strasse {
   display: block;
}
```

Wird die XML-Datei nun in einem entsprechenden Browser aufgerufen, der XML und CSS interpretieren kann, ergibt sich die Darstellung aus **Abbildung 6.5**.

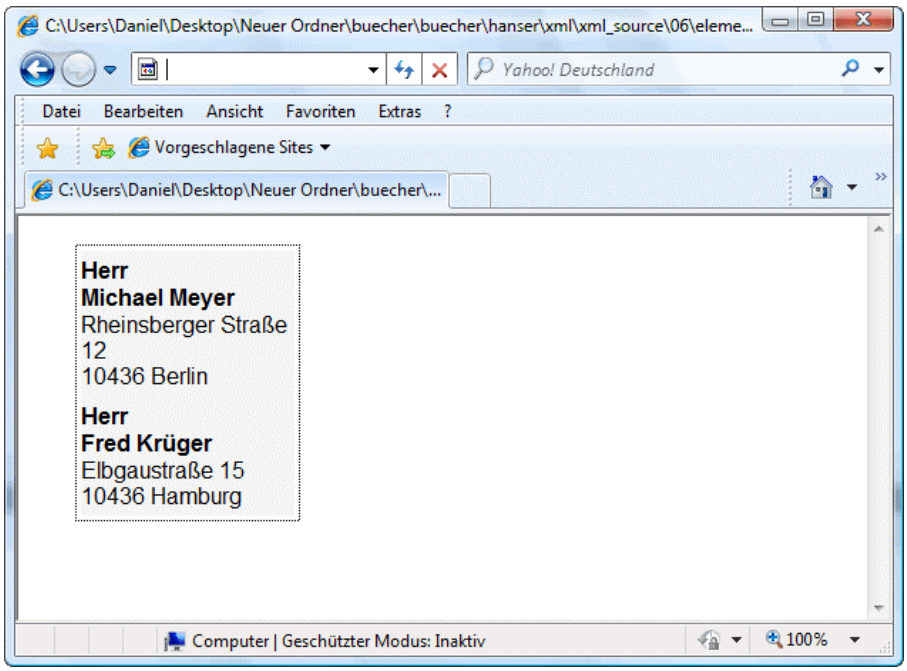

Abbildung 6.5 So einfach lassen sich XML-Dokumente mit CSS formatieren.

Wie Sie sehen, lässt sich mit vergleichsweise wenig Aufwand ein durchaus ansprechendes Ergebnis erzielen.

6.3 Die Schwächen von CSS (in Bezug auf XML)

Auf den vorherigen Seiten wurde gezeigt, wie einfach sich XML-Dokumente mittels CSS formatieren lassen. Und auf den ersten Blick wirkt das durchaus verlockend. So finden erfahrungsgemäß all diejenigen Entwickler die XML-CSS-Variante interessant, die bislang ihre HTML-Dokumente mit CSS formatiert haben. Schließlich ist bei denen kein „Umdenken“ nötig, sondern sie können auf bestehendes Know-how zurückgreifen. So schön die Kombination aus CSS und XML auf den ersten Blick aber auch sein mag, sie hat doch gravierende Nachteile:

- Ältere Browser können mit XML und CSS nichts anfangen. Die zeigen dann entweder gar nichts oder unformatierten Text.
- In CSS hat man kaum die Möglichkeit, Daten bei der Ausgabe zu filtern. Aber eben genau das ist in XML oft gewünscht. So will man z.B. häufig nicht alle Datensätze einer XML-Struktur ausgeben. An solchen Filterungen scheitert CSS.
- Wichtige Funktionen, die z.B. mit XSLT möglich sind, lassen sich mit CSS nicht umsetzen. Beispielhaft seien dafür das Gruppieren und Sortieren genannt.

Diese Einschränkungen sprechen gegen den Einsatz von CSS.

7

7 Transformation mit XSLT

XSLT ist ein Teil von XSL. Mit der XSL Transformation steht eine Sprache für die Transformation von XML-Dokumenten zur Verfügung. In diesem Kapitel lernen Sie den Umgang mit XSLT kennen und erfahren, wie Sie mit einer Stylesheet-Sprache – im Unterschied zu CSS – tatsächlich programmieren können.

Am Anfang wird allerdings die Frage geklärt, was es denn eigentlich mit den vielen Abkürzungen auf sich hat. Denn erfahrungsgemäß fällt es auch XML-erfahrene Entwicklern schwer zu erklären, wo eigentlich die Unterschiede zwischen XSLT, XSL-FO und XSL liegen.

7.1 Sprachverwirrung: XSLT, XSL und XSL-FO

Sehr oft wird im XML-Umfeld abwechselnd von XSL, XSLT und von XSL-FO gesprochen. Zu allem Überfluss werden die Begriffe oft auch noch miteinander kombiniert oder im schlimmsten Fall verwechselt. Bevor es in diesem Kapitel um den Einstieg in die XSLT-Welt geht, noch ein paar klärende Worte hinsichtlich der Begrifflichkeiten.

Eigentlich ist es ganz einfach: Bei XSL, der Extensible Stylesheet Language, handelt es sich um eine Sprachfamilie für die Erzeugung von Layouts für XML-Dokumente.

XSL setzt sich insgesamt aus den folgenden Sprachen zusammen:

- XSL-FO
- XSLT
- XPath

In vielen Büchern wird XSL oft als Synonym für XSLT genommen. Der Hauptgrund für diese Verwechslung dürfte sicherlich sein, dass das üblicherweise in XSLT verwendete Namensraumpräfix `xsl` lautet.

An anderen Stellen wird wiederum XSL-FO mit XSL gleichgesetzt, was jedoch ebenso falsch ist. Denn die XSL-Spezifikation besagt ausdrücklich, dass sich XSL aus einer Spra-

che für die Transformation (XSLT), einer Sprache für die Formatierung (XSL-FO) und einer Abfragesprache (XPath) zusammensetzt.

Ein Blick auf die offizielle XSL-Seite des W3C unter *http://www.w3.org/Style/XSL/* gibt noch einmal Aufschluss darüber. In dem Papier *What is XSL* heißt es da ausdrücklich:

XSL shares the functionality and is compatible with CSS2 (although it uses a different syntax). It also adds:

A transformation language for XML documents: XSLT. Originally intended to perform complex styling operations, like the generation of tables of contents and indexes, it is now used as a general purpose XML processing language. XSLT is thus widely used for purposes other than XSL, like generating HTML web pages from XML data.

Advanced styling features, expressed by an XML document type which defines a set of elements called Formatting Objects, and attributes (in part borrowed from CSS2 properties and adding more complex ones).

Diese Aussage macht noch einmal deutlich, was XSL eigentlich genau ist und was es mit XSLT auf sich hat.

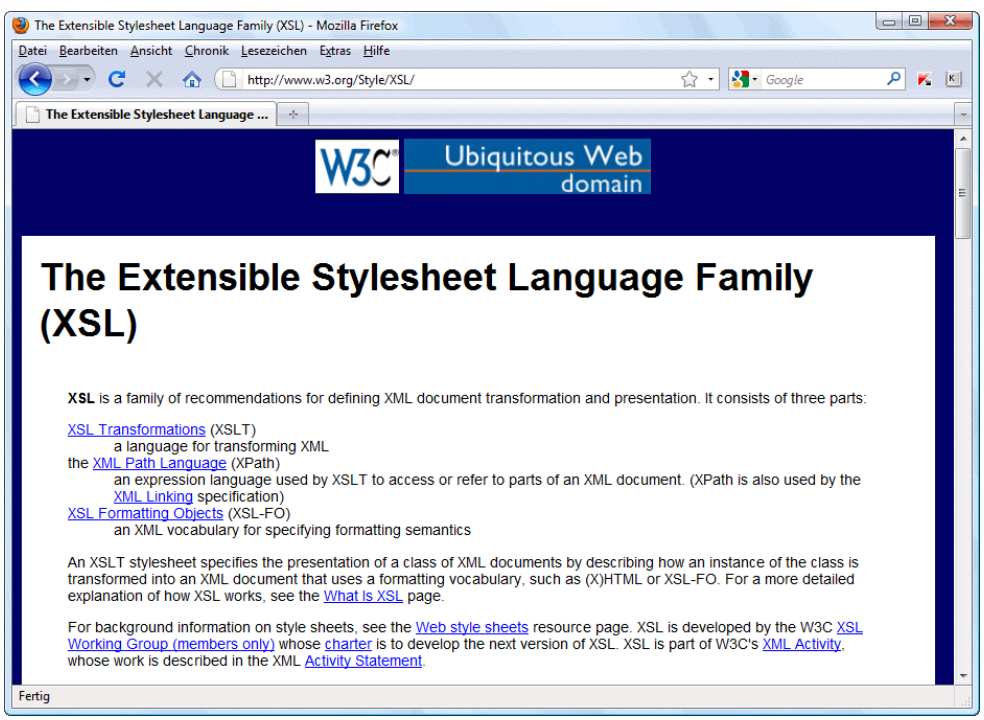

Abbildung 7.1 Hier gibt es ausführliche Informationen zu XSL.

Für die Arbeit an und mit XSLT sind Kenntnisse in XPath nötig. Nur so nämlich können Sie ganz gezielt auf die gewünschten Elemente bzw. Knoten von XML-Dokumenten zugreifen.

7.2 Das Grundprinzip der Transformation

Am Anfang dieses Kapitels muss zunächst der Begriff Transformation geklärt werden. Mit Transformation ist ganz allgemein die Veränderung der Struktur oder Gestalt gemeint. Auf (XML-)Dokumente umgemünzt bedeutet die Transformation die Überführung in ein anderes Format oder die Umformung der Datenstruktur.

Die Gründe für eine mögliche Umwandlung eines XML-Quelldokuments können ganz unterschiedlicher Natur sein.

- Die Übersetzung zwischen unterschiedlichen XML-Vokabularen.
- Daten sollen anhand bestimmter Kriterien sortiert und gefiltert werden.
- XML-Inhalte sollen in HTML oder XHTML übersetzt werden.
- Aus XML-Inhalten sollen einfache Textinhalte gemacht werden.

In den meisten Fällen können die unterschiedlichen Strukturen von XML-Dokumenten ineinander überführt werden. So können z.B. Dokumente sogenannte inhaltsorientierte Auszeichnungen enthalten, die beim Publizieren durch entsprechende layoutorientierte Repräsentation ersetzt werden. Sollen Inhalte dargestellt werden, die in XML vorliegen, müssen diese in Dokumente eines Präsentationsvokabulars transformiert werden. Und eben diese Transformationen lassen sich innerhalb von XSLT-Dokumenten beschreiben und mittels XSLT-Prozessoren durchführen.

Momentan wird für die meisten XML-Dokumente HTML oder XHTML als Präsentationssprache verwendet. Eine bestimmte Transformation von Inhalten ist unabhängig von Formatierungsobjekten, wenn man sie dafür verwendet, zwischen XML-Vokabularien zu übersetzen, oder wenn der vorhandene Quelltext z.B. erweitert werden soll. Bei der Umwandlung von XML in (X)HTML werden allerdings inhaltliche und formale Elemente in einem gewissen Umfang vermischt.

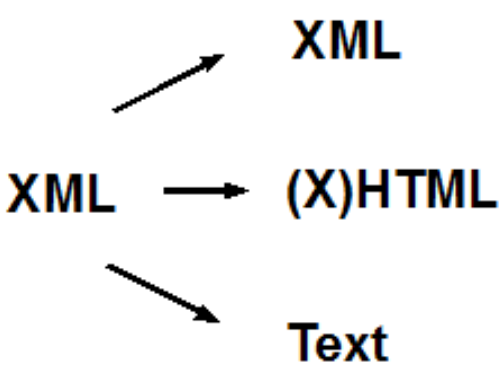

Abbildung 7.2 Die Vermischung der Inhalte

Der XML-Standard XSLT wurde bereits im Jahr 1999 verabschiedet. XSLT baut auf XML 1.0 und XML Namespaces auf. Zudem wird XPath 1.0 vorausgesetzt. Dabei wurde XPath 1.0 zunächst als ein Teil von XSLT entwickelt. Seit Oktober 2001 liegt die Empfehlung für XSL vor, bei der es sich sozusagen um den Elternentwurf von XSLT handelt.

Auch wenn XSLT in XSL als notwendiger Bestandteil eingeschlossen ist, kann die Sprache trotzdem unabhängig agieren. So gibt es mittlerweile zahlreiche Tools, die XSLT unterstützen.

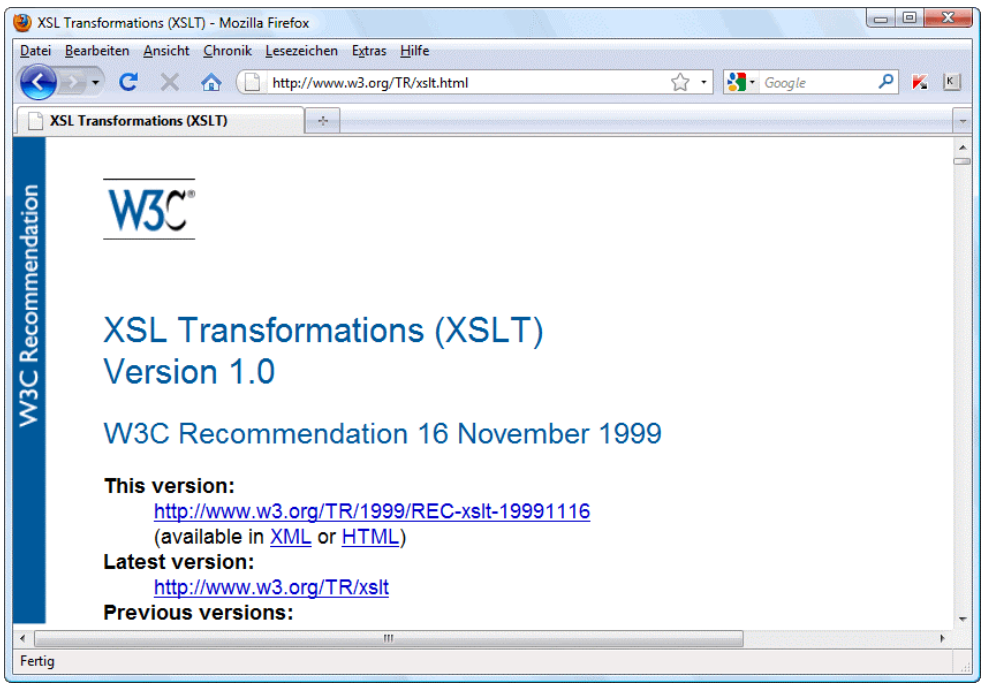

Abbildung 7.3 Alles zum Thema XSLT gibt es in dieser Spezifikation.

Die Umwandlung von XML-Dokumenten in Dokumente eines anderen Typs ist mit verschiedenen Programmiersprachen realisierbar. Im Vergleich zu anderen Sprachen bietet XSLT allerdings den Vorteil, dass lediglich Regeln dafür angegeben werden müssen, welche Strukturen des Quelldokuments in welche Strukturen des Zieldokuments überführt werden sollen. Wie die Überführung letztendlich realisiert wird, dafür ist der XSLT-Prozessor bzw. dessen Implementierung verantwortlich. In XSLT legt man also lediglich das gewünschte Ergebnis fest. Wie dieses Ergebnis letztendlich erzielt wird, spielt hingegen keine Rolle.

XSLT hat einen weiteren entscheidenden Vorteil. Denn XSLT-Dokumente sind selbst XML-Dokumente. So können Sie XSLT-Dokumente mit klassischen XML-Tools er- und verarbeiten. Zudem ist die Lernkurve für erfahrene XML-Entwickler ziemlich niedrig. Sprich: Wer XML beherrscht, wird sich auch schnell in XSLT einarbeiten können.

7.2.1 XSLT-Prozessoren im Einsatz

Für die Transformation von XML-Dokumenten mittels XSLT wird eine spezielle Software, und zwar ein XSLT-Prozessor, benötigt. Der Markt an XSLT-Prozessoren ist mittlerweile recht umfangreich geworden. Die einzelnen Produkte unterscheiden sich vor allem darin, wie vollständig der XSLT-Standard implementiert wurde.

Der XSLT-Prozessor liest das XSLT-Stylesheet sowie die XML-Quelldatei ein, führt die im Stylesheet enthaltenen Anweisungen aus und generiert das entsprechende Ergebnisdokument. Dabei setzt der XSLT-Prozessor die Anweisungen, die innerhalb des Stylesheets definiert wurden, selbstständig um.

Die Transformation kann dabei auf ganz unterschiedliche Arten erfolgen. So ist es beispielsweise möglich, dass das Stylesheet bereits auf dem Webserver auf das betreffende XML-Dokument angewendet wird. Bei dieser Variante liefert der Webserver dann das Ergebnis-Dokument an den Client. Ebenso lässt sich das Stylesheet aber auch direkt auf Clientseite anwenden. Das geht beispielsweise in einem entsprechend modernen Browser oder anderer „Spezialsoftware“.

7.2.1.1 Xalan

Welchen XSLT-Prozessor Sie letztendlich einsetzen, hängt zunächst einmal von den Anforderungen und Ihren persönlichen Vorlieben ab. Ein weit verbreiteter Open-Source-XSLT-Prozessor ist beispielsweise Xalan. Entwickelt wird dieser Prozessor vom Apache XML Project. Die Projektwebseite, von der übrigens der Prozessor auch heruntergeladen werden kann, finden Sie unter *http://xalan.apache.org/index.html*. Xalan gibt es in einer Java- und in einer C++-Version. Der Xalan-Prozessor unterstützt XSLT 1.0 und XPath 1.0 jeweils vollständig. Die Verwendung von Xalan ist denkbar einfach. Wie das funktioniert, wird hier anhand der Java-Windows-Version gezeigt.

1. Entpacken Sie das heruntergeladene Archiv.
2. Nehmen Sie mindestens für die folgenden Dateien einen *classpath*-Eintrag vor: *xalan.jar*, *xml-apis.jar* und *xercesImpl.jar*.
3. Seit Windows 2000 können Sie diese Einstellungen ganz bequem über die Systemsteuerung vornehmen. Unter Windows Vista rufen Sie dazu *Start*/*Systemsteuerung* auf und öffnen den Punkt *System*. Über *Erweiterte Systemeinstellungen* und *Umgebungsvariablen* wird das entsprechende Dialogfenster geöffnet.

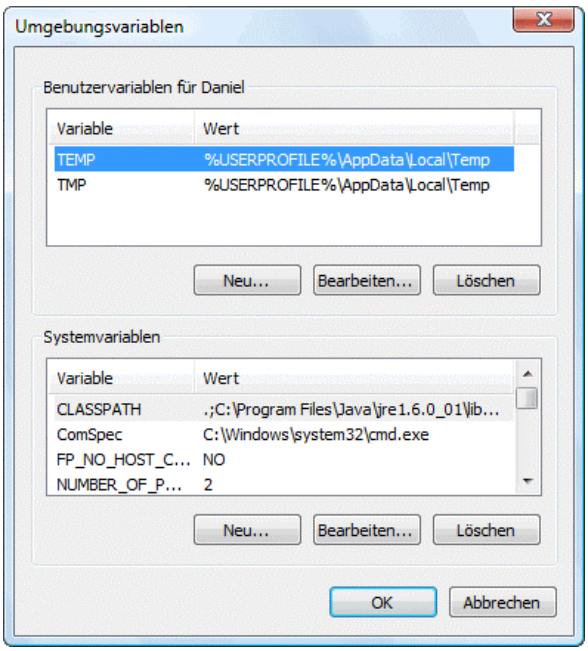

Abbildung 7.4 Die Umgebungsvariable wird definiert.

4. Über die Schaltfläche *Neu* können die entsprechenden Einstellungen vorgenommen werden.

Damit sind die Vorarbeiten auch bereits abgeschlossen, und Xalan ist einsatzbereit. Um ein XML-Dokument zu transformieren, kann die Eingabeaufforderung genutzt werden. Hier ein typischer Aufruf, bei dem die beiden Dateien *welt.xml* und *welt.xsl* eingelesen werden.

java org.apache.xalan.xslt.Process -in welt.xml

-xsl welt.xsl -out welt.html

Das Ergebnisdokument ist in diesem Beispiel die Datei *welt.html*.

7.2.1.2 Saxon

Ein anderer beliebter XSLT-Prozessor ist Saxon. Saxon ist Java-basiert und unterstützt XSLT 2.0, Teile von XSLT 2.0 sowie XQery 1.0. In früheren Versionen war Saxon frei verfügbar. Das hat sich mit Version 7 allerdings geändert. Jetzt gibt es nur noch eine abgespeckte Version kostenlos, in der die Schema-Unterstützung fehlt. Dieses Manko ändert allerdings nichts daran, dass Saxon ein erstklassiger XSLT-Prozessor ist. Heruntergeladen werden kann die kostenlose Version von der Projektseite http://saxon.sourceforge.net/. Saxon ist dabei mehr als ein normaler XSLT-Prozessor. Denn das Tool kennt zahlreiche Erweiterungen, zu denen zusätzliche Elemente, Attribute und Funktionen gehören. Wer möchte kann sogar eigene Erweiterungen programmieren.

Die zuvor erwähnte kommerzielle Saxon-Variante finden Sie unter *http://www.saxonica.com/*. Dort können Sie Saxon als 30-Tage-Testversion herunterladen.

Wenn Sie sich anschließend für einen Kauf entscheiden, kommen je nach Version zwischen 50 und 300 US-Dollar an Lizenzgebühren auf Sie zu.

Um Saxon verwenden zu können, wird eine Java-Laufzeitumgebung in Version 1.5 oder höher vorausgesetzt. Nach dem Herunterladen der Archivdatei von der Seite *http://saxon.sourceforge.net/* entpacken Sie diese in das gewünschte Verzeichnis. Anschließend kann Saxon genutzt werden. Ein typischer Beispielaufruf sieht folgendermaßen aus:

java –jar $pfad/saxon9.jar welt.xml welt.xsl

Wobei *$pfad* durch den tatsächlichen Pfad zu Saxon zu ersetzen ist. Als weitere notwendige Argumente werden dem XSLT-Prozessor die Namen des zu transformierenden XML-Dokuments und des XSLT-Stylesheets übergeben. Saxon hat auch zahlreiche Kommandozeilenoptionen zu bieten, mit denen die Verarbeitung des Eingabedokuments sowie die Art des Ausgabedokuments beeinflusst werden können. Eine vollständige Liste dieser Optionen können Sie sich durch die Eingabe des Arguments *-?* anzeigen lassen.

7.2.1.3 XMLBlueprint

Der Editor XMLBlueprint ist kein reiner XSLT-Prozessor. Vielmehr handelt es sich bei XMLBlueprint um eine vollwertige XML-Entwicklungsumgebung für Windows. Hier die wichtigsten Funktionen, die XMLBlueprint auszeichnen:

- Syntax-Highlighting
- Autovervollständigung
- Validierung
- Unicode

Neben diese allgemeinen Funktionen ist – und genau deswegen taucht der Editor in dieser Übersicht auf – eben auch ein XSLT-Prozessor. Die Anwendung dieses Features ist dabei denkbar einfach. Zunächst einmal können Sie in XMLBlueprint wie üblich Ihre XSLT-Dateien erstellen.

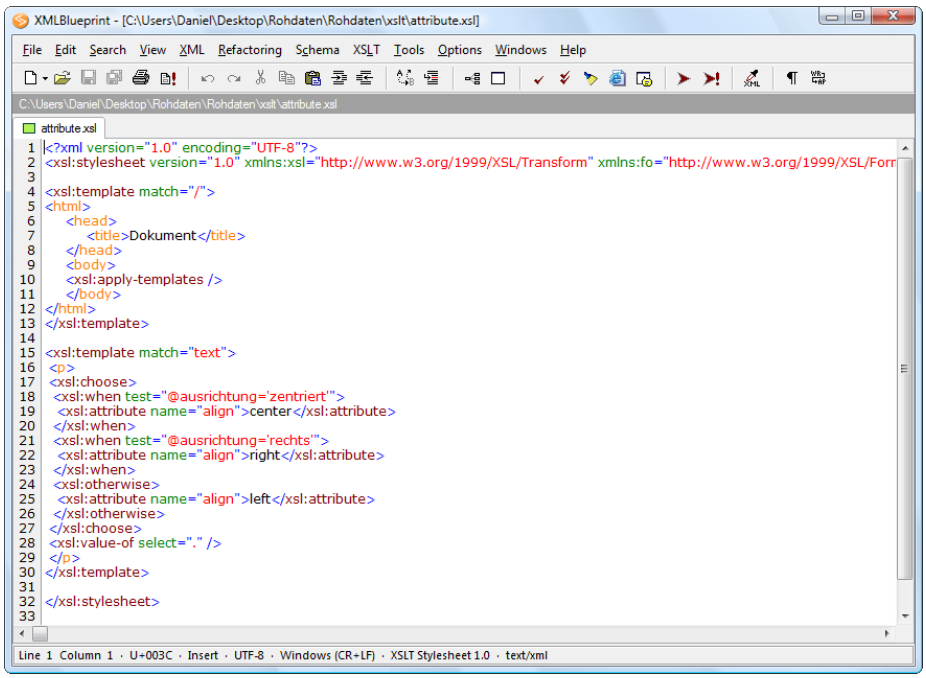

Abbildung 7.5 Ein Editor mit zahlreichen Funktionen

Das ist zunächst einmal nichts Besonders. Interessant wird es allerdings, wenn es um die Transformation geht. Denn das funktioniert in XMLBlueprint völlig unkompliziert.

Um ein XML-Dokument zu transformieren, öffnen Sie es zusammen mit dem XSLT-Stylesheet. Anschließend bracht man nur noch die Taste *[F9]* zu drücken oder alternativ *XSLT/Run XSLT Transformation* zu wählen.

Wurde kein spezielles XML-Dokument angegeben, öffnet sich ein entsprechendes Dialogfenster, in dem das Dokument ausgewählt werde kann.

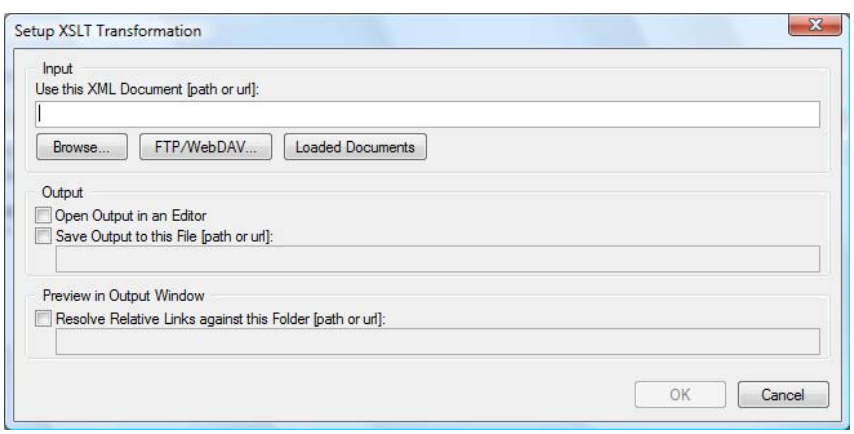

Abbildung 7.6 Darüber kann das Dokument ausgewählt werden.

Die eigentliche Transformation leitet man mit *OK* ein. Die Ausgabe lässt man sich dann üblicherweise im Ausgabefenster von XMLBlueprint anzeigen.

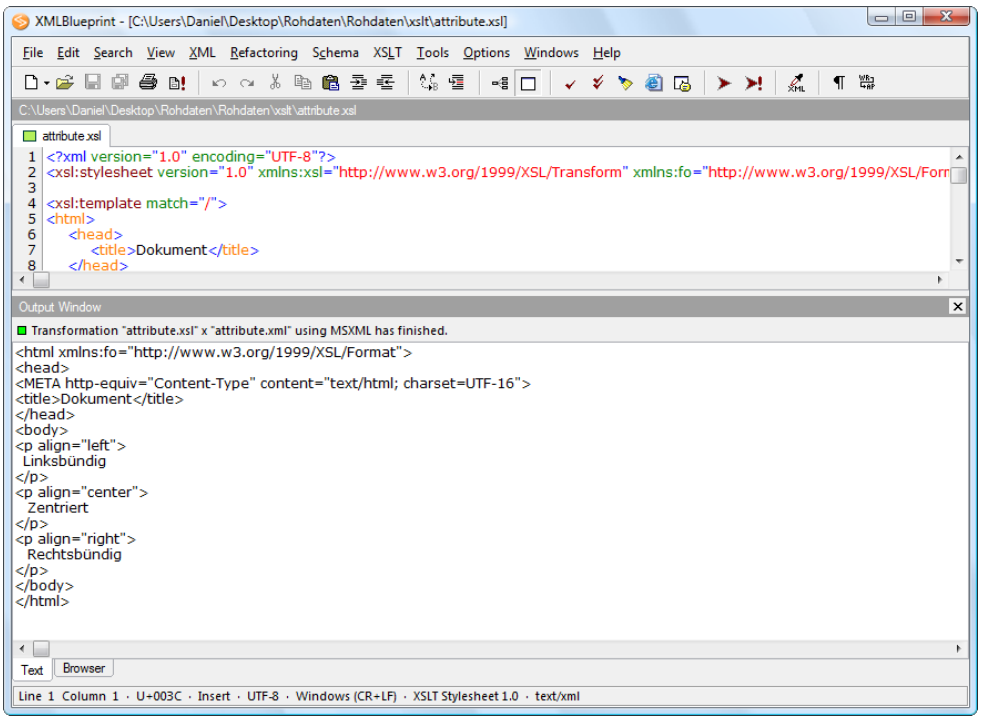

Abbildung 7.7 Auch ein Ausgabefenster gehört mit dazu.

Dieses Beispiel hat gezeigt, dass XMLBlueprint sicherlich zu den am einfachsten zu bedienenden Anwendungen gehört. Ausführliche Informationen zu diesem Tool finden Sie unter *http://www.xmlblueprint.com/*. Dort kann XMLBlueprint als Testversion heruntergeladen werden. Für den Kauf einer Vollversion werden 70 US-Dollar fällig.

7.2.1.4 Einsatz im Browser

Innerhalb moderner Browser können Sie Ihre XSLT-Anwendungen direkt testen. Denn die Browser transformieren XML-Dokumente während der Laufzeit. Einzige Voraussetzung dafür: Eine Processing Instruction verweist aus der XML-Datei heraus auf das Stylesheet, und dieses Stylesheet ist dann auch tatsächlich verfügbar.

Wie das in einem geeigneten Browser funktioniert, wird nachfolgend anhand einer einfachen XML-Datei gezeigt, die im Internet Explorer aufgerufen wird. Zunächst die XML-Datei ohne Processing Instruction:

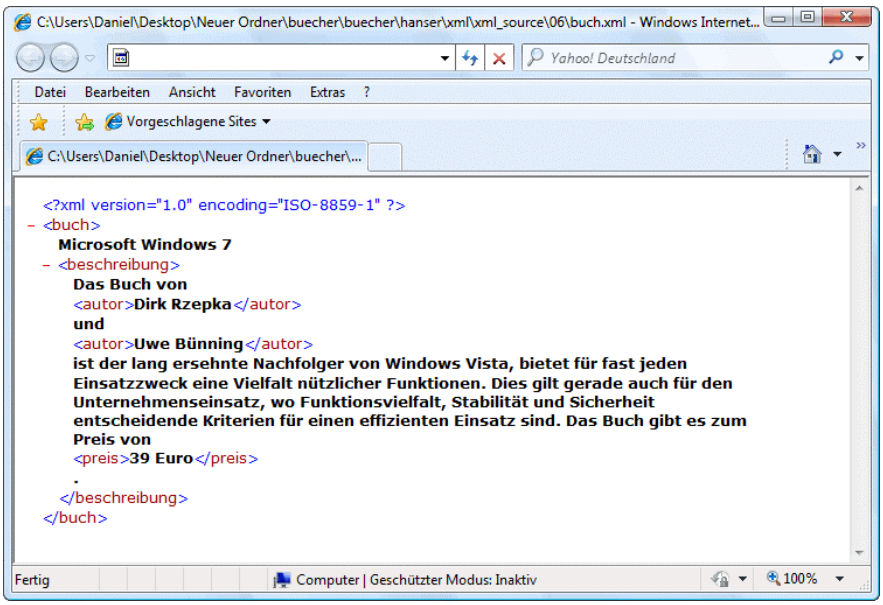

Abbildung 7.8 XML im Internet Explorer

Wie erwartet stellt der Internet Explorer die Baumstruktur des XML-Dokuments dar. Anders sieht es aus, wenn man die XML-Datei mit einem entsprechenden Stylesheet verknüpft. Erweitern Sie die XML-Datei dazu um einen Verweis auf ein XSL-Stylesheet.

Listing 7.1 Ein Stylesheet wird aufgerufen.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="ausgabe.xsl" ?>
<buch>
   Microsoft Windows 7
<beschreibung>Das Buch von <autor>Dirk Rzepka</autor> und <autor>Uwe Bün-
ning</autor> ist der
[...]
```

Die verwendete *ausgabe.xml* enthält Stylesheet-Informationen, über die das XML-Dokument formatiert wird. Der Vollständigkeit halber, ohne jedoch bereits jetzt auf die Syntax einzugehen, zunächst der Inhalt dieser XSL-Datei:

Listing 7.2 Dieses Stylesheet wird verwendet.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
   <html>
     <head>
     </head>
   <body style="font-family:Verdana; font-size:20px; color:red">
      <xsl:apply-templates />
   </body>
   </html>
</xsl:template>
```

```
<xsl:template match="beschreibung">
   <p style="font-family:Verdana; font-size:12px; color:black">
      <xsl:apply-templates />
   </p>
</xsl:template>

<xsl:template match="autor">
   <span style="font-weight:bold; color:red">
      <xsl:value-of select="." />
   </span>
</xsl:template>

<xsl:template match="preis">
   <b>
      <xsl:value-of select="." />
   </b>
</xsl:template>

</xsl:stylesheet>
```

Wird das XML-Dokument nun erneut im Browser aufgerufen, dann ergibt sich folgendes Bild:

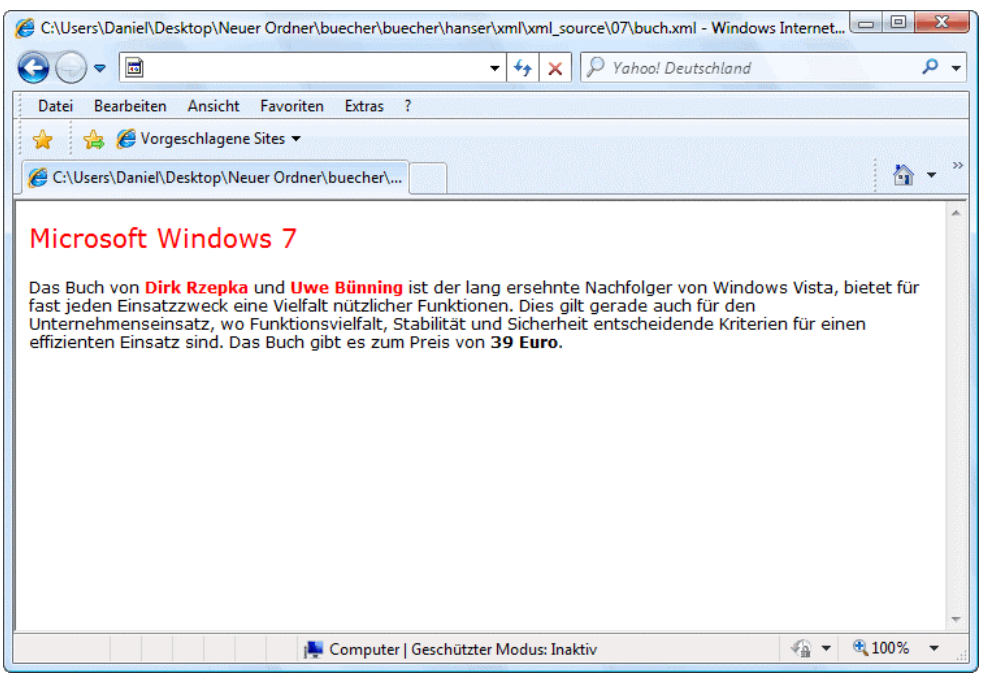

Abbildung 7.9 Auch das ist eine XML-Datei.

Das XML-Dokument wird wie eine ganz normale HTML-Webseite angezeigt. Von XML ist hier – zumindest vordergründig – nichts zu sehen. Auf den folgenden Seiten geht es exakt um diesen Punkt: Wie können XML-Dokumente mittels XSLT transformiert werden?

7.3 Der Einstieg in XSLT: *Hallo, Welt!*

Wie so oft in der Programmierwelt, soll der Einstieg in XSLT anhand des klassischen *Hallo, Welt!* erleichtert werden.

XSLT lässt sich auf zwei verschiedene Arten auf XML-Dokumente anwenden.

- XSLT wird direkt in das XML-Dokument eingebettet.
- XSLT und XML stehen in separaten Dokumenten, und beide Dokumente werden an einen XSLT-Prozessor übergeben.

Im folgenden *Hallo, Welt*-Beispiel wird davon ausgegangen, dass alle notwendigen Dateien im gleichen Verzeichnis liegen. Zunächst die DTD:

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

Es handelt sich hier um eine sehr einfache DTD, in der lediglich ein Element definiert wurde. Gespeichert wird diese DTD unter dem Namen *welt.dtd*. In diesem Zusammenhang noch ein Hinweis: Im vorliegenden Kapitel wird der Fokus auf der Arbeit mit XSLT liegen. Aus diesem Grund wird im weiteren Verlauf des Kapitels auf die Definition von DTDs verzichtet.

Werfen Sie nun einen Blick auf das XML-Dokument.

Listing 7.3 Eine sehr einfache XML-Datei

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE welt SYSTEM "welt.dtd">
<?xml-stylesheet type="text/xsl" href="welt.xsl" ?>
   <welt>Hallo, Welt!</welt>
```

Hier wird zunächst über das `DOCTYPE`-Element die externe DTD eingebunden. Anschließend wird eine Processing Instruction definiert. Dabei werden der Typ (`type`) sowie der Pfad der einzubindenden Datei angegeben. Diese Elemente wurden in diesem Buch bereits ausführlich vorgestellt. Interessant für dieses Kapitel ist in erster Linie der Aufbau bzw. Inhalt der XSL-Datei *welt.xsl*, die die Formatierungen enthält.

Listing 7.4 Das ist die XSL-Datei.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
   <html>
      <head>
      </head>
   <body>
      <h1>
        <xsl:value-of select="." />
      </h1>
   </body>
   </html>
</xsl:template>
</xsl:stylesheet>
```

Wird die eigentliche XML-Datei z.B. im Firefox oder im Internet Explorer aufgerufen, ergibt sich ein Anblick wie auf **Abbildung 7.10**:

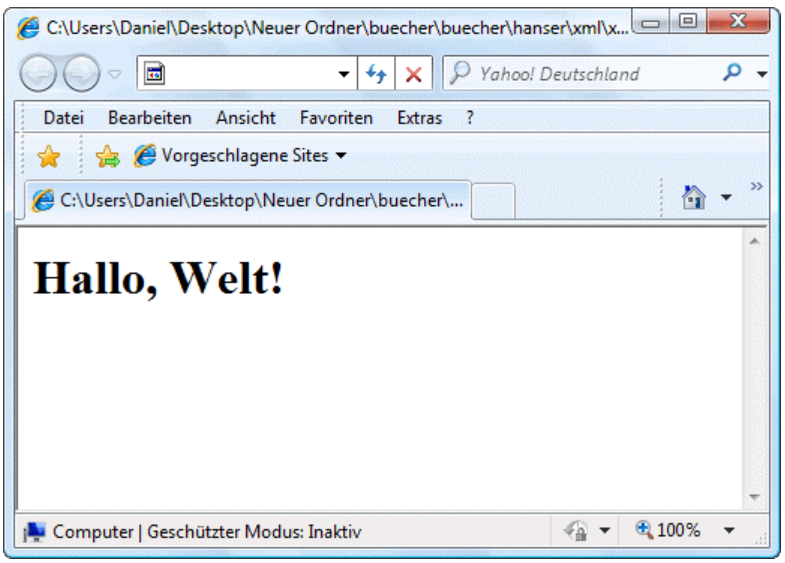

Abbildung 7.10 Das Dokument im Internet Explorer

Die optischen Auswirkungen der XML- und XSL-Syntax haben Sie gesehen. Jetzt wird noch ein detaillierter Blick auf den eigentlichen Quelltext geworfen. Am Anfang jeder XSL-Datei steht die XML-Deklaration.

```
<?xml version="1.0" encoding="UTF-8"?>
```

Die eigentliche XSL-Datei wird vollständig von dem Element `xsl:stylesheet` umrahmt. Die äußere Struktur der Datei sieht somit folgendermaßen aus:

Listing 7.5 Der Rahmen einer XSLT-Datei

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

…

</xsl:stylesheet>
```

Hierbei handelt es sich übrigens um die typische Schreibweise, wenn innerhalb einer Datei mehrere Namensräume verwendet werden sollen. Innerhalb von XSLT-Dokumenten muss in jedem Fall der XSLT-Namensraum angegeben werden. Zusätzlich können Sie Namensräume für XML-Schema und XPath definieren. Dazu muss das einleitende `xsl:stylesheet`-Element entsprechend erweitert werden.

Listing 7.6 Zusätzliche Angaben wurden eingefügt.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
```

```
xmlns:xsl="http://www.w3.org/1999/XSL/Transform
xmlns:xs=http://www.w3.org/2001/XMLSchema
xmlns:fn=http://www.w3.org/2005/xpath-functions
xmlns:xdt="http://www.w3.org/2005/xpath-datatypes">
```

Der wichtigste Bestandteil von XSLT-Dokumenten sind Templates. Über solche Templates wird die Ausgabe gesteuert. Dabei gibt man über das `match`-Attribut an, welches Element ausgegeben bzw. transformiert werden soll. Bei dem Attributwert handelt es sich um einen XPath-Ausdruck. Soll das Wurzelelement, also das zentrale Template, angegeben werden, sieht die Syntax folgendermaßen aus:

Listing 7.7 Die Definition eines Templates

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

Um innerhalb der Stylesheets auf Werte zuzugreifen, wird das `select`-Attribut des `xsl:value`-Elements verwendet.

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

Über diese Syntax wird der Knoten ausgewählt, dessen Inhalt ausgegeben werden soll.

7.3.1 Das Element `xsl:stylesheet`

Da es sich bei den Stylesheets um XML-Dokumente handelt, werden sie mit der XML-Deklaration eingeleitet. Daran schließen sich alle Komponenten an, eingefasst in das vorgegebene Wurzelelement `xsl:stylesheet`. Als Synonym dafür kann auch `xsl:transform` verwendet werden. Das kann beispielsweise sinnvoll sein, um Anwendungen, die nur Daten transformieren, von solchen zu unterscheiden, die auch für Formatierungen zuständig sind.

Noch ein Wort zur allgemeinen Schreibweise: Alle zu XSLT gehörenden Element-, Attribut- und Funktionsnamen werden kleingeschrieben.

Dem `xsl:stylesheet`-Element muss das `version`-Attribut zugewiesen werden. Dessen Wert kann `1.0` oder `2.0` sein. Geben Sie die Version an, die Sie innerhalb Ihres Stylesheets verwenden.

Ebenfalls zwingend ist die Angabe eines Namensraums, der die vorgegebenen Elemente und Attribute umfasst. Der entsprechende URI des XSLT-Namensraums ist *http://www.w3org/1999/XSL/Transform*.

Listing 7.8 Der Namensraum wurde angegeben.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

Standardmäßig wird das Präfix `xsl` verwendet, Sie können aber natürlich auch ein anderes einsetzen.

7.3.2 Top-Level-Elemente

Das Wurzelelement `xsl:stylesheet` haben Sie im vorherigen Abschnitt kennengelernt. Nun geht es um die möglichen Kindelemente, die als Top-Level-Elemente bezeichnet werden.

- `xsl:import`
- `xsl:include`
- `xsl:namespace-alias`
- `xsl:attribute-set`
- `xsl:template`
- `xsl:variable`
- `xsl:key`
- `xsl:output`
- `xsl:strip-space`
- `xsl:param`
- `xsl:decimal-format`
- `xsl:preserve-space`

Diese Elemente können innerhalb des Stylesheets in beliebiger Reihenfolge notiert werden. Ausnahme davon bilden `xsl:import`-Elemente, die immer am Anfang des Dokuments stehen müssen. Was es mit den einzelnen Elementen auf sich hat, wird auf den folgenden Seiten ausführlich gezeigt.

Folgende Syntax zeigt die allgemeine Struktur eines Stylesheets. Die in diesem Beispiel enthaltenen Punkte zeigen an, an welchen Stellen Attributwerte und Inhalte weggelassen wurden.

Listing 7.9 Das ist die Struktur.

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:import href="..."/>
  <xsl:include href="..."/>
  <xsl:strip-space elements="..."/>
  <xsl:preserve-space elements="..."/>
  <xsl:output method="..."/>
  <xsl:key name="..." match="..." use="..."/>
  <xsl:decimal-format name="..."/>
  <xsl:namespace-alias stylesheet-prefix="..." result-prefix="..."/>
  <xsl:attribute-set name="...">

    ...

  </xsl:attribute-set>

  <xsl:variable name="...">

    ...

  </xsl:variable>
```

```
  <xsl:param name="...">
     ...
  </xsl:param>
  <xsl:template match="...">
    ...
  </xsl:template>
  <xsl:template name="...">
    ...
  </xsl:template>
</xsl:stylesheet>
```

Neben den genannten Top-Level-Elementen können Sie eigene Top-Level-Erweiterungselemente verwenden. In der Praxis könnte man darüber beispielsweise Metadaten für ein Stylesheet angeben. Für diese Elemente muss ein eigener Namensraum definiert werden. Zudem steht es XSLT-Prozessoren frei, diese Elemente umzusetzen.

Im folgenden Beispiel wird diese Möglichkeit genutzt, um den Wert einer Variablen während der Laufzeit ändern zu können.[1] Für diesen Zweck gibt es eine Erweiterung für den Saxon-Prozessor, mit dem genau so etwas umsetzbar ist.

Listing 7.10 Varialben in Aktion

```
<xsl:stylesheet xmlns:xsl=http://www.w3.org/1999/XSL/Transform
     xmlns:saxon=http://icl.com/saxon
     extension-element-prefixes="saxon"
     version="1.0">

<xsl:output method="xml" omit-xml-declaration="yes"/>
  <xsl:variable name="color"
  saxon:assignable="yes">blue</xsl:variable>
  <xsl:template match="/">
    <saxon:assign name="color">yellow</saxon:assign>

    Farbe:

    <xsl:value-of select="$color"/>
  </xsl:template>

</xsl:stylesheet>
```

Um diese Erweiterung zu nutzen, muss zunächst der Namensraum *http://icl.com/saxon* angegeben werden. Im vorliegenden Beispiel wurde dieser Namensraum mit dem Präfix `saxon` verbunden. Dieses wurde dem Attribut `extension-element-prefixes` als Wert zugewiesen.

Eines der genannten Top-Level-Elemente verdient noch einmal eine genauere Betrachtung. Denn mittels `xsl:output` können Sie die Methode bestimmen, mit der das Ergebnisdokument ausgegeben werden soll. Wollen Sie als Ergebnis z.B. eine HTML-Datei, wählen

[1] Was so in XSLT standardmäßig übrigens nicht möglich ist.

Sie `method="html"`. Insgesamt stehen für `method` drei verschiedene Werte und somit Ausgabevarianten zur Verfügung.

- `html` – Es werden HTML-Elemente und Attribute erkannt und interpretiert.
- `text` – Damit werden die String-Werte aller Textknoten ausgegeben, die im Ergebnisbaum enthalten sind. Die Reihenfolge entspricht dabei der im Dokument.
- `xml` – Hierdurch wird ein wohlgeformtes XML-Dokument erzeugt.

Wenn Sie `xsl:output` nicht angeben, wird von den Prozessoren standardmäßig `xml` oder `html` verwendet. Welches letztendlich zum Einsatz kommt, hängt davon ab, ob das `html`-Element als erstes gefunden wird oder nicht.

7.4 Templates definieren

Das Kernkonzept von XSLT stellen die Templates dar. An dieser Stelle zunächst einige grundlegende Hinweise zu diesem Thema. Denn nicht alles, was im Zusammenhang mit XSLT als Template bezeichnet wird, ist auch tatsächlich eins. In XSLT muss man zwischen Templates und Template Rules (Template-Regeln) unterscheiden. Eine Template-Regel definiert Regeln für den Transformationsprozess und besitzt immer ein Template.

Bei dem Template selbst handelt es sich um ein Stylesheet, das innerhalb des Ergebnisdokuments instanziiert wird.

7.4.1 Template-Regeln

Template-Regeln sind Vorlagen für die Transformation bestimmter Knoten. Bei diesen Knoten kann es sich z.B. um Attribute oder Elemente handeln. Durch jede Template-Regel wird festgelegt, wie Fragmente für den Ergebnisbaum generiert werden sollen. Das Fragment kann dabei sogar leer sein. Ebenso kann aber auch der gesamte Ergebnisbaum ausschließlich über eine Template-Regel definiert werden. Innerhalb jeder Template-Regel wird eine Bedingung definiert, wann die Regel anzuwenden ist. Das kann zunächst einmal ein Muster sein, das auf bestimmte Knoten im Eingabebaum passt. Ebenso kann man aber auch einen Namen angeben, über den die Regel immer – auch unabhängig von den Knoten des Eingabebaums – Anwendung findet.

Die wichtigsten Anweisungen für den XSLT-Prozessor werden innerhalb des Stylesheets in einer losen Folge von Template-Regeln definiert, die jeweils innerhalb des Elements

```
xsl:template
```

eingeschlossen sind. Es spielt keine Rolle, an welcher Stelle die Template-Regeln stehen. Das Ergebnis ist immer das gleiche.

Innerhalb des Templates trifft man auf zwei verschiedene Dinge. Bei dem einen handelt es sich um XSLT-Elemente mit Anweisungen an den XSLT-Prozessor. Zu erkennen sind diese daran, dass sie dasselbe Namensraumpräfix wie das Elternelement `xsl:template` haben.

Listing 7.11 Die Templates werden verwendet.

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

So handelt es sich in diesem Beispiel bei `xsl:apply-templates` um eine allgemeine Anweisung, durch die die innerhalb des Stylesheets vorhandenen Template-Regeln ausgeführt werden.

Innerhalb des einleitenden `xsl:template`-Elements ist das Attribut `match` zu finden. Darüber wird angegeben, welches Element des XML-Dokuments ausgewählt werden soll.

7.4.2 Suchmuster/Pattern einsetzen

Für den Zugriff auf Elemente können verschiedene Suchmuster verwendet werden. Will man z.B. auf den Wurzelknoten zugreifen, wählt man folgenden Ausdruck:

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

Ebenso einfach gelingt auch der gezielte Zugriff auf einzelne Elemente. Existiert beispielsweise ein `autor`-Element, könnte der Zugriff folgendermaßen aussehen:

```
<xsl:template match="autor">

...
```

In solchen Suchmustern sind die Bedingungen definiert, die erfüllt sein müssen, damit das Template auf einen Knoten angewendet wird.

An dieser Stelle kommt dann auch das in diesem Buch bereits vorgestellte XPath zum Einsatz. Denn mit einer Untermenge der XPath-Sprache werden solche Suchmuster definiert.

Tabelle 7.1 stellt einige dieser Muster bzw. Pattern vor.

Tabelle 7.1: Wichtige Patterns in der Übersicht

Pattern	Beschreibung
`/`	Wurzelelement
`*`	Jedes Element
`@*`	Jedes Attribut
`text()`	Textelemente
`Elementname`	Alle Elemente mit dem angegebenen Namen
`id(Name)`	Alle Elemente, die den Namen Name haben.
`@Name`	Alle Attribut-Knoten mit dem angegebenen Namen.
`list-el[position() mod 2 = 1]`	Alle ungeraden Elemente

Pattern	Beschreibung
`Name1 | Name2`	Jedes Element Name1 oder Name2
`Name1/Name2`	Jedes Element Name2 als Kind von Name1
`processing-instruction()`	Alle Procession Instructions
`node()`	Alle Knoten, die keine Attribute oder das Wurzelelement haben

7.5 Der Ablauf der Transformation

Bevor weiter ins Detail gegangen wird, soll und muss ein Blick darauf geworfen werden, wie XSLT-Prozessoren eigentlich Stylesheets abarbeiten.

Bekommt ein XSLT-Prozessor den Auftrag, ein bestimmtes Stylesheet auf ein XML-Dokument anzuwenden, um auf diese Weise das Ergebnisdokument zu generieren, werden zunächst beide Dokumente eingelesen und geparst. Zusätzlich baut der Prozessor in seinem Speicher für beide Dokumente jeweils eine interne Baumstruktur auf.

Erst danach wird die eigentliche Transformation eingeleitet. Bei der Abarbeitung des Stylesheets wird auf die Quelldaten exakt in dem Umfang zugegriffen, in dem das die Template-Regeln erforderlich machen. Aus den Quelldaten und den literalen Ergebnisdokumenten wird der Ergebnisbaum generiert. Durch Serialisierung wiederum werden aus diesem Ergebnisbaum die Ergebnisdokumente erzeugt, die vom Stylesheet angefordert werden. Das kann eine Textdatei, eine HTML- oder eine XML-Datei sein.

7.5.1 Alles beginnt beim Wurzelknoten

Jede Transformation beginnt gleich bzw. startet an der gleichen Position. Das ist immer das Wurzelelement des XML-Baums. Die Template-Regel, die für diesen Wurzelknoten vorhanden ist, wird demnach zuerst abgearbeitet. Dabei ist es unerheblich, in welcher Reihenfolge die Template-Regeln innerhalb des Stylesheets stehen. Der Prozessor durchsucht das Stylesheet immer nach der Template-Regel, die entweder speziell für das Wurzelelement definiert wurde oder die sich darauf anwenden lässt.

Stößt der Browser auf keine solche Regel, wird er normalerweise auf eine eingebaute Template-Regel zurückgreifen. Sollten mehrere Regeln vorhanden sein, die sich widersprechen bzw. in Konkurrenz zueinander stehen, tritt das Prinzip der Priorisierung in Kraft. Auch dazu dann später mehr.

7.5.2 Anwendung von Templates

Wie wichtig Templates sind, ist bereits angeklungen. Das sehen so auch die XSLT-Prozessoren. Werfen Sie zunächst einen Blick auf die folgende Syntax:

Listing 7.12 Das Template wird angewendet.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
   <html>
      <head>
      </head>
   <body>
      <xsl:apply-templates />
   </body>
   </html>
</xsl:template>

</xsl:stylesheet>
```

Stößt der Prozessor auf die Anweisung `xsl:apply-templates`, muss er diese erst ausführen, bevor er das `body`-Element schließen kann. Denn schließlich soll die Anweisung den eigentlichen Inhalt des `body`-Elements liefern.

Diese Anweisung tritt innerhalb der Template-Regel für den Wurzelknoten auf. Der Wurzelknoten bildet damit den maßgeblichen Kontext für diese Anweisung. Somit entspricht das einer neuen Aufforderung an den XSLT-Prozessor, alle Kindknoten des aktuellen Knotens auszuwählen und für jeden Knoten nach einer darauf passenden Regel zu suchen und diese auch entsprechend anzuwenden.

7.5.3 Verhalten bei Template-Konflikten

Interessant ist die Frage, was eigentlich passiert, wenn mehrere Templates für den gleichen Knoten gelten. In solchen Fälle treten dann ganz bestimmte Regeln in Kraft, die die Priorisierung bestimmen.

- Regeln für eine spezifische Information haben immer Vorrang vor einer Regel für allgemeinere Informationen. So besitzt z.B. `match="/buch/autoren/autor"` eine höhere Priorität als `match="autor"`.
- Suchmuster, bei denen Wildcards wie `@` oder `*` eingesetzt werden, sind allgemeiner als Muster ohne Wildcards.
- Sollte keines der genannten Kriterien weiterhelfen, ist die Reihenfolge, in der die Template-Regeln im Stylesheet definiert wurden, entscheidend. Dabei hat immer die zuletzt definierte Regel Vorrang. An dieser Stelle aber noch einmal der Hinweis, dass die Reihenfolge wirklich nur in einem solchen Fall wichtig ist.
- Man kann auch selbst Einfluss auf die Priorität nehmen. Verwendet wird dafür das Attribut `priority`. Dessen Wert muss eine echte Zahl sein, die negativ oder positiv sein kann. Eine höhere Zahl bedeutet eine höhere Priorität.

Ein Beispiel zeigt den Einsatz von `priority`.

Listing 7.13 Hier wurden Prioritäten gesetzt.

```
<xsl:template match="autor" priority="-1">
  <span class="autor">
```

```
    <xsl:apply-templates />
  </span>
</xsl:template>

<xsl:template match="buch/autor" priority="2">
  <span class="autor">
      <xsl:apply-templates select="Name" />
   </span>
</xsl:template>
```

7.5.4 Stylesheets mit nur einer Template-Regel

Solche Template-Regeln, die für die Dokumentwurzel gelten, enthalten zumeist einen Aufruf für weitere Regeln, die dann für einzelne Teile des Dokuments bestimmt sind. Ebenso ist es möglich, dass ein Stylesheet aus einer einzigen Template-Regel besteht. In einem solchen Fall kann man eine vereinfachte Syntax anwenden.

Listing 7.14 Eine neue Template-Regel wurde definiert.

```
<?xml version="1.0" encoding="UTF-8"?>
<html xsl:version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<head>
</head>
<body>
   <xsl:apply-templates />
</body>
</html>
```

In dieser Syntax werden die notwendigen Angaben direkt in das öffnende `html`-Element eingefügt. Bei diesen Angaben handelt es sich um `xsl:version` und den Wert für den Namensraum. Also immer, wenn ausschließlich eine Template-Regel verwendet wird, können Sie auf diese verkürzte Syntax zurückgreifen.

7.5.5 Überschreiben von Template-Regeln

Template-Regeln können sich gegenseitig überschreiben. Das ist wichtig zu verstehen, wenn man Stylesheets anlegt. Denn wenn das Ergebnis plötzlich anders als erwartet aussieht, könnte genau dieser Aspekt schuld daran sein.

Eine Template-Regel, die verwendet wird, um eine Template-Regel in einem importierten Stylesheet zu überschreiben, kann mittels des `xsl:apply-imports`-Elements die überschriebene Template-Regel aufrufen.

`xsl:apply-imports` verarbeitet den aktuellen Knoten nur mit den Template-Regeln, die in das Stylesheet-Element importiert wurden, in dem die aktuelle Template-Regel enthalten ist. Dabei ist es ein Fehler, wenn `xsl:apply-imports` instanziiert wird, während die aktuelle Template-Regel leer ist. Am besten lässt sich der Aspekt des Überschreibens anhand eines Beispiels zeigen.

Es wird davon ausgegangen, dass innerhalb des Stylesheets *autoren.xsl* eine Template-Regel für `autoren`-Elemente enthalten ist.

Listing 7.15 So sieht das `autoren`-Element aus.

```
<xsl:template match="autoren">
  <pre>
      <xsl:apply-templates/>
   </pre>
</xsl:template>
```

Nun könnte ein anderes Stylesheet *autoren.xsl* importieren und die Bearbeitung der `autoren`-Elemente wie folgt anpassen:

Listing 7.16 Die Datei wird importiert.

```
<xsl:import href="autoren.xsl"/>
<xsl:template match="autoren">
  <div style="border: solid red">
     <xsl:apply-imports/>
  </div>
</xsl:template>
```

Das Ergebnis dieser Variante wäre ein modifiziertes `autoren`-Element:

Listing 7.17 Das ist die Ausgabe.

```
<div style="border: solid red">
   <pre>...</pre>
</div>
```

7.5.6 Mit Modi Elemente mehrmals verarbeiten

Durch den Einsatz von Modi wird es möglich, Elemente mehrmals zu verarbeiten und dabei jedes Mal ein anderes Ergebnis zu erhalten. Den beiden Elementen `xsl:template` und `xsl:apply-templates` kann jeweils das optionale Attribut `mode` zugewiesen werden. So lassen sich unterschiedliche Template-Regeln für ein und dasselbe Pattern definieren. Wenn beim Ausführen eines Templates kein `mode`-Attribut spezifiziert wurde, wird eine Template-Regel ohne `mode`-Attribut gesucht und ausgeführt.

Als Wert des `mode`-Attributs wird ein Name erwartet, bei dem die in XML üblichen Regeln für die Definition von Namen einzuhalten sind.

Sollte `xsl:template` kein `match`-Attribut besitzen, darf es auch kein `mode`-Attribut enthalten. Wenn ein `xsl:apply-templates`-Element ein `mode`-Attribut besitzt, wird es nur die Template-Regeln der `xsl:template`-Elemente anwenden, die dasselbe `mode`-Attribut haben.

Listing 7.18 Das `mode`-Attribut wurde verwendet.

```
<xsl:template match="images" mode="einfach" >
  <!-- Transformiere ein images-Element in einfacher Form -->
</xsl:template>

<xsl:template match="images" mode="explizit" >
  <!-- Transformiere ein images-Element in expliziter Form -->
</xsl:template>
```

7.5.7 Eingebaute Template-Regeln

In XSLT gibt es verschiedene eingebaute Template-Regeln. Die erste, die hier vorgestellt wird, dient dazu, dass eine rekursive Verarbeitung bei einem Nicht-Vorhandensein eines passenden Mustertests zu einer expliziten Template-Regel innerhalb des Stylesheets fortgesetzt werden kann. Diese Template-Regel wird sowohl auf die Elementknoten wie auch auf den Wurzelknoten angewendet.

Listing 7.19 Eine Template-Regel

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

Testen lässt sich diese eingebaute Regel ganz einfach. Dazu brauchen Sie nur ein Stylesheet auf ein XML-Dokument anzuwenden, das noch keine Template-Regeln enthält. In diesem Fall gibt der Prozessor dann lediglich die String-Werte aller Elemente des Quelldokuments aus. Die Attribute werden dabei ignoriert. Jede explizite Template-Regel überschreibt die eingebaute Regel, wenn sie sich auf den Wurzelknoten bezieht.

Es gibt eine weitere Template-Regel. Die ist für den Einsatz unterschiedlicher Anwendungsmodi gedacht. Diese Regel erlaubt es, dass die rekursive Verarbeitung bei einem Nicht-Vorhandensein eines erfolgreichen Mustertests zu einer expliziten Template-Regel in dem Stylesheet in dem gleichen Modus fortgesetzt wird. Angewendet wird diese Template-Regel sowohl auf die Elementknoten wie auch auf den Wurzelknoten.

Listing 7.20 Jetzt wird diese Template-Regel verwendet.

```
<xsl:template match="*|/" mode="m">
  <xsl:apply-templates mode="m"/>
</xsl:template>
```

Eine andere Template-Regel gibt es für Text- und Attributknoten. Durch diese wird Text direkt kopiert. Allerdings wird diese Regel nur dann ausgeführt, wenn die Knoten ausgewählt wurden.

Listing 7.21 Der Text wird direkt kopiert.

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

Es gibt auch eine Template-Regel für Kommentare und Verarbeitungsanweisungen. Durch diese Regel werden beide Elementvarianten nicht in das Ergebnisdokument übernommen. Die Template-Regel ist leer, und der Prozessor macht in diesem Fall nichts.

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

Nun kann es natürlich durchaus sein, dass Sie Kommentare im Ergebnisdokument sichtbar machen wollen. Auch das ist möglich.

Listing 7.22 So werden Kommentare angezeigt.

```
<xsl:template match="comment()">
  <xsl:comment>
     <xsl:value of select="."/>
  </xsl:comment>
</xsl:template>
```

Durch diese Syntax wird der Inhalt eines Kommentars im Ausgabedokument wieder als Kommentar ausgezeichnet. Verwendet wird dafür die bekannte Kommentarsyntax:

Listing 7.23 Ein Kommentar wird definiert.

```
<!--
   Kommentar
-->
```

Die eingebauten Template-Regeln werden so verarbeitet, als ob sie implizit vor dem Stylesheet importiert worden wären. Die Template-Regeln haben also eine geringere Priorität als die im Stylesheet eingefügten Regeln. Sie werden demnach nur angewendet, wenn andere explizite Regeln fehlen.

7.6 Templates aufrufen

Die grundsätzliche Arbeitsweise von XSLT-Templates haben Sie auf den vorherigen Seiten kennengelernt. Nun geht es darum, wie sich die Templates aufrufen lassen. Verantwortlich für den Aufruf ist das Element `xsl:apply-templates`, das die beiden folgenden Attribute kennt:

- `select` – Zuerst wird die Knotenmenge gewählt, dann werden die Templates für die Knoten gesucht. Sollte eine `xsl:template`-Definition für das Element vorhanden sein, wird diese angewendet. Ist kein `select`-Attribut vorhanden, werden alle nächstuntergeordneten `xsl:template`-Definitionen eingesetzt.
- `mode` – Die bei `select` angegebene Knotenmenge wird nur dann verwendet, wenn sie den angegebenen Modus hat. Dafür muss bei der Template-Definition mit `xsl:template` der Modusname mittels des `modus`-Attributs angegeben werden.

Zunächst die allgemeine Syntax des in XSLT wahrscheinlich am häufigsten genutzten Elements.

Listing 7.24 So sieht die allgemeine Syntax aus.

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

Wenn das `select`-Attribut nicht vorhanden ist, verarbeitet `xsl:apply-templates` alle Kinder des aktuellen Knotens inklusive der Textknoten. Ausnahmen bilden lediglich solche Textknoten, bei denen Leerräume entfernt wurden. Sollte das Entfernen von Leerräumen nicht aktiviert worden sein, werden sämtliche Leerräume im Elementinhalt als Text verarbeitet.

Das folgende Beispiel zeigt die Verwendung von `xsl:apply-templates`. Zunächst die Syntax des Ausgangsdokuments:

Listing 7.25 Das ist das Ausgangsdokument.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="ausgabe.xsl" ?>
<bibliothek>
   <autoren>
      Es war ein <eigenschaft>guter
      </eigenschaft> Autor.
  </autoren>
</bibliothek>
```

Das entsprechende XSLT-Stylesheet soll nun dafür sorgen, dass HTML-Code generiert wird, der folgendermaßen aussieht:

Listing 7.26 Dieser Code soll herauskommen.

```
<p>
   Es war ein <i>guter</i> Autor.
</p>
```

Um dieses Ergebnis zu erreichen, muss das entsprechende Stylesheet definiert werden. Bereits ein kurzer Blick auf diese Syntax zeigt, dass dabei gleich zwei `xsl:apply-templates`-Elemente verwendet werden.

Listing 7.27 Dieses Stylesheet sorgt für die richtige Ausgabe.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
   <html>
      <head>
      </head>
   <body>
      <xsl:apply-templates />
   </body>
   </html>
</xsl:template>

<xsl:template match="autoren">
   <p>
      <xsl:apply-templates />
   </p>
</xsl:template>

<xsl:template match="eigenschaft">
   <i>
      <xsl:value-of select="." />
   </i>
</xsl:template>

</xsl:stylesheet>
```

Da wäre zunächst einmal das `xsl:apply-templates`-Element, das innerhalb der Template-Definition für die Dokumentwurzel auftaucht. Da dabei kein select-Attribut verwendet wurde, werden an der betreffenden Stelle innerhalb des generierten HTML-Codes die Templates für die nächstuntergeordneten Elemente angewendet. Bei dem gezeigten Beispiel handelt es sich dabei um das `autoren`-Element.

Für sämtliche `autoren`-Elemente wird mittels `xsl:template match="autoren"` jeweils ein eigenes Template definiert. Alle Elemente werden mit dem `p`-Element versehen. Für deren Inhalte müssen nun allerdings wieder die untergeordneten Elemente berücksichtigt werden. Daher wird ein Absatz mit einer entsprechenden Template-Anweisung definiert.

Listing 7.28 Ein Absatz wird erzeugt.

```
<p>
   <xsl:apply-templates />
</p>
```

Als untergeordnetes Element von `autoren` ist `eigenschaft` definiert. Untergeordnet bedeutet in diesem Fall für die Ausgabe Folgendes:

S**Listing 7.29** So sehen untergeordnete Elemente aus.

```
<p>
   Es war ein <i>guter</i> Autor.
</p>
```

Bei `i` handelt es sich um ein Kindelement von `p`. Für `eigenschaft` wird über `<xsl:template match="eigenschaft">` ebenfalls ein Template definiert. Text, der innerhalb von `eigenschaft` steht, wird mit `i` ausgezeichnet und somit als kursiv dargestellt. Innerhalb von `eigenschaft` muss nicht auf untergeordnete Elemente geachtet werden, sondern man kann sich ganz auf `<xsl:value-of select="." />` konzentrieren. Das generierte Dokument sieht im Browser dann folgendermaßen aus:

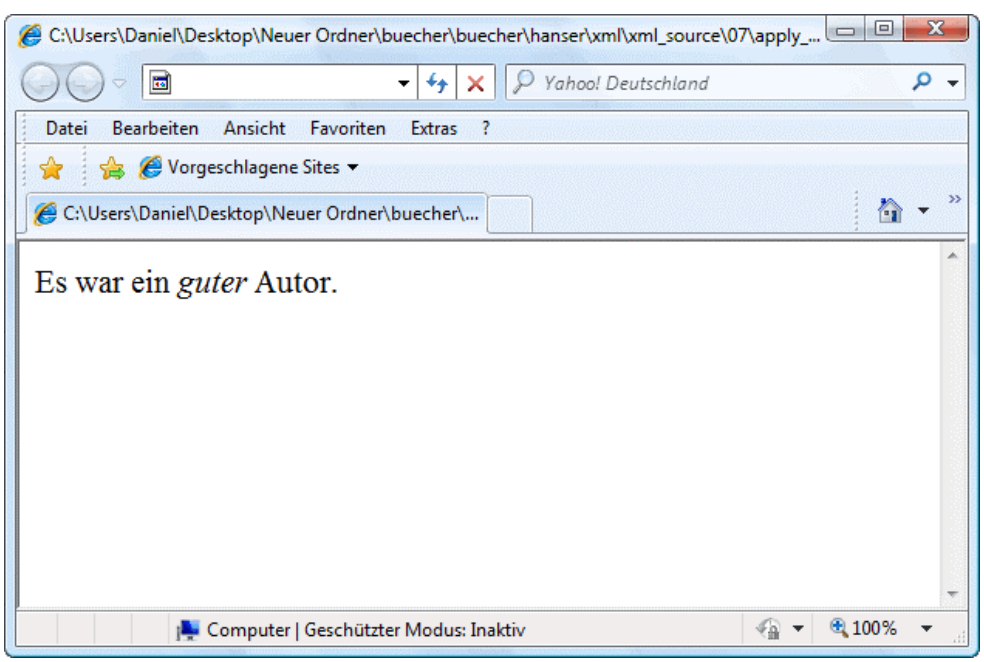

Abbildung 7.11 Die Ausgabe sieht wie gewünscht aus.

7.7 Templates einbinden

Templates lassen sich auch an solchen Stellen einbinden, an denen man das eigentlich nicht machen sollte. Ein typisches Beispiel dafür ist es, wenn überhaupt kein entsprechender Knoten vorhanden ist. Umgehen lässt sich dieses Problem, indem man das `xsl:call-template`-Element einsetzt. Hierüber lassen sich Templates über ihren Namen aufrufen. Das funktioniert allerdings nur, wenn das `xsl:template`-Element über das `name`-Attribut als benanntes Template spezifiziert wurde.

Listing 7.30 Ein benanntes Template wird spezifiziert.

```
<xsl:template name="Template-Name">
   Inhalt
</xsl:template>
```

Wenn ein `xsl:template`-Element ein `name`-Attribut besitzt, kann es ein `match`-Attribut besitzen, muss es aber nicht. Für den Aufruf eines solchen Templates wird `xsl:call-template` verwendet. Dieses Element besitzt ein notwendiges `name`-Attribut, über welches das Template angegeben wird.

```
<xsl:call-template name="Template-Name" />
```

Anders als bei `xsl:apply-template` wird der aktuelle Knoten oder die aktuelle Knotenliste bei `xsl:call-template` nicht verändert.

Die innerhalb eines Templates definierten Attribute `mode`, `match` und `priority` beeinflussen nicht, ob das Template durch `xsl:call-template` aufgerufen wird. Genauso wenig beeinflusst das `name`-Attribut von `xsl:template`, ob das Template von einem `xsl:apply-template`-Element aufgerufen wird.

Die genannten Aspekte lassen sich wieder am besten anhand eines Beispiels zeigen. Das Ausgangsdokument sieht folgendermaßen aus:

Listing 7.31 Das ist das Ausgangsdokument.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="ausgabe.xsl" ?>
<bibliothek>
   <autor telefon="040198998">
      Holger Geiger
   </autor>
</bibliothek>
```

Innerhalb des `autor`-Elements wurde das Attribut `telefon` definiert. Dessen Wert wiederum ist die Telefonnummer des Autors. Durch ein entsprechendes Stylesheet soll aus dem gezeigten XML-Code die folgende HTML-Syntax generiert werden:

Listing 7.32 Diese Syntax soll rauskommen.

```
<div>
   Holger Geiger, Telefon: 040198998
</div>
```

Damit das klappt, muss innerhalb des Stylesheets zunächst einmal ein Template definiert und diesem ein Name zugewiesen werden. Bevor die Erklärungen im Detail kommen, hier schon einmal das vollständige Stylesheet:

Listing 7.33 Das ist das Stylesheet.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
   <html>
      <head>
       </head>
    <body>
       <xsl:apply-templates />
    </body>
    </html>
</xsl:template>
<xsl:template name="TName">
   <xsl:text>, Telefon: </xsl:text>
</xsl:template>

<xsl:template match="autor">
   <div>
      <xsl:value-of select="." />
      <xsl:call-template name="TName" />
      <xsl:value-of select="@telefon" />
   </div>
</xsl:template>

</xsl:stylesheet>
```

Wie bereits eingangs beschrieben, wird dem Template der Name über das `name`-Attribut zugewiesen. Innerhalb des relevanten `TName`-Templates wird mittels `xsl:text` statischer Text ausgegeben.

Interessanter ist da schon das zweite Template. Dort wird über `<xsl:call-template name="TName" />` das zuvor definierte und benannte Template aufgerufen. Die übrigen Angaben im Stylesheet dienen dem Auslesen der relevanten Elemente des Ausgangsdokuments.

Ein anschließender Blick in das generierte Dokument liefert das Bild aus **Abbildung 7.12**.

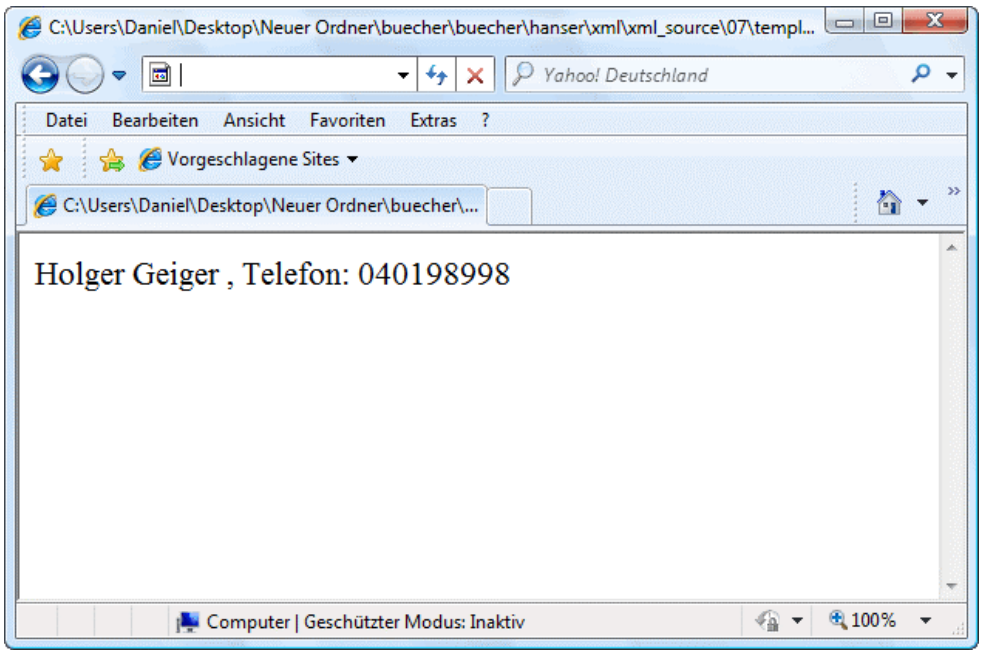

Abbildung 7.12 So sieht die Ausgabe aus.

Beim Aufruf des Templates ist beim Template-Namen unbedingt auf Groß- und Kleinschreibung zu achten.

7.8 Stylesheets einfügen, importieren und wiederverwenden

Ein Stylesheet kann andere Stylesheets einschließen und diese einen bestimmten Teil der Transformationsaufgaben übernehmen lassen. Stylesheets können also aus mehrfach verwendbaren Bausteinen bestehen.

XSLT unterstützt zwei Mechanismen zum Kombinieren von Stylesheets:

- Ein Importmechanismus, durch den es Stylesheets erlaubt wird, sich gegenseitig zu überschreiben.
- Ein Inklusionsmechanismus, durch den es Stylesheets erlaubt wird, miteinander kombiniert zu werden, ohne dass dabei die Semantik der kombinierten Stylesheets verändert wird.

Die einzelnen Varianten werden auf den folgenden Seiten vorgestellt. Den Anfang macht das normale Einfügen von Stylesheets.

7.8.1 Stylesheets einfügen

Die einfachste Variante ist sicherlich das Einfügen bzw. Inkludieren von Stylesheets. Dabei wird lediglich der Text des externen Stylesheets eingefügt. Die eingefügten Templates und Elemente werden dabei so behandelt, als ob sie von Anfang an zu dem Stylesheet gehört hätten. Wenn also beispielsweise innerhalb eines Stylesheets verschiedene Farbangaben für die spätere Verwendung definiert wurden, könnte das folgendermaßen aussehen:

Listing 7.34 Verschiedene Farbangaben werden definiert.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/ Transform">
   <xsl:variable name="TextFarbe" select="'Black'" />
   <xsl:variable name="KastenFarbe" select="'Red'" />
   <xsl:variable name="BackFarbe" select="'Blue'" />
</xsl:stylesheet>
```

In diesem Beispiel werden Variablen verwendet. Ausführliche Informationen dazu folgen im weiteren Verlauf dieses Kapitels. Um diese Variablendefinitionen einzufügen, wird innerhalb des Stylesheets das `xsl:include`-Element verwendet.

Listing 7.35 Die Variablendefinition wird verwendet.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

 <xsl:include href="farben.xslt"/>
 ...
```

Im gezeigten Beispiel wird davon ausgegangen, dass die Variablendefinitionen innerhalb der Datei *farben.xslt* stehen, die sich im gleichen Verzeichnis wie das Stylesheet befindet.

7.8.2 Stylesheets importieren

Das Importieren unterscheidet sich vom Einfügen. Denn hier unterscheidet der Prozessor sowohl bei den Templates als auch bei den möglicherweise vorhandenen globalen Variablen zwischen solchen, die ursprünglich zu dem importierten Stylesheet gehören, und denen, die importiert wurden. Es greift das Prinzip der Import-Präzedenz. Dabei wird den importierten Objekten ein geringerer Rang als den bereits vorhandenen zugestanden. Daher ist der Import immer dann sinnvoll, wenn allgemeine Vorgaben importiert werden sollen, die in bestimmten Fällen aber überschrieben bzw. deaktiviert werden müssen.

Im Konfliktfall haben Stylesheet-Definitionen, die in der aktuellen Datei definiert wurden, Vorrang vor importierten Stylesheet-Definitionen.

Für den Import wird das Element `xsl:import` verwendet. Dieses Element muss außerhalb von `xsl:template` stehen. Ein Beispiel zeigt, wie der Import funktioniert.

Listing 7.36 Ein Stylesheet wird aufgerufen.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="import.xsl" ?>
<bibliothek>
   <name>Arno Geiger</name>
</bibliothek>
```

Interessant ist zunächst das `href`-Attribut, über das das Stylesheet `import.xsl` aufgerufen und auf die Datei angewendet wird. Der Inhalt dieser Datei sieht folgendermaßen aus:

Listing 7.37 Dieses Stylesheet ruft ein weiteres auf.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:import href="import2.xsl" />

<xsl:template match="/">
   <html>
      <head>
      </head>
   <body>
      <xsl:apply-imports/>
   </body>
   </html>
</xsl:template>

</xsl:stylesheet>
```

Innerhalb dieses Stylesheets wird abermals über `xsl:import` ein Stylesheet importiert, dieses Mal allerdings die `import2.xsl`. In diesem Stylesheet wiederum wird für das `name`-Element ein Textabsatz definiert. Dabei wird das Stylesheet exakt an der Stelle angewendet, an der `xsl:apply-imports` steht.

Listing 7.38 Das ist das andere Stylesheet.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="name">
   <p>
      <xsl:value-of select="." />
   </p>
</xsl:template>

</xsl:stylesheet>
```

Dass der doppelte Stylesheet-Import funktioniert hat, zeigt **Abbildung 7.13**.

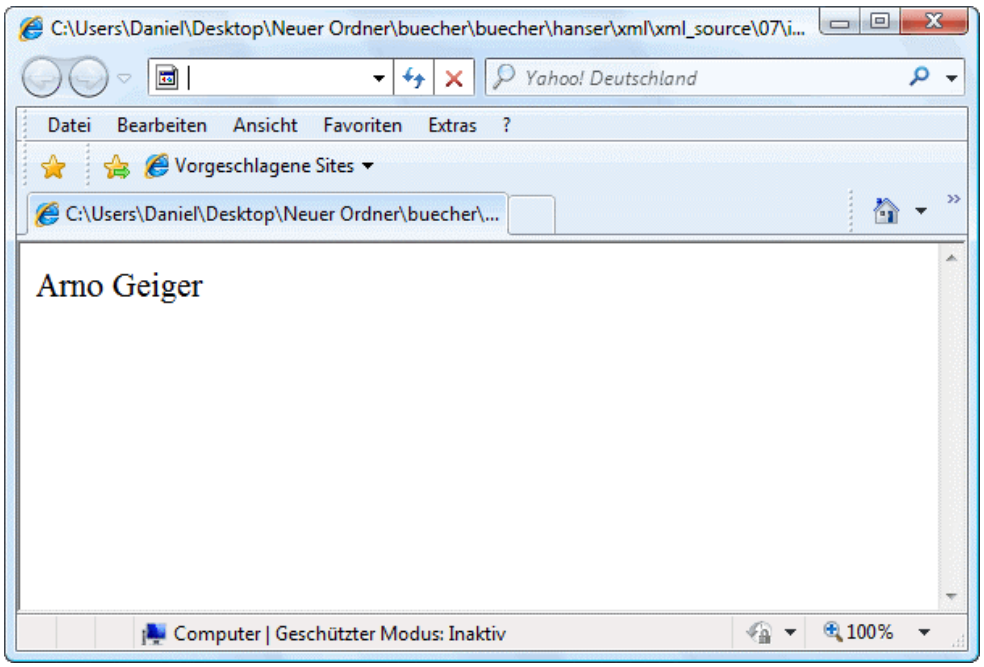

Abbildung 7.13 Die Stylesheets wurden korrekt importiert.

7.8.3 Stylesheets inkludieren

Das Element `xsl:include` wurde bereits kurz vorgestellt. Mit diesem Element kann ein Stylesheet ein anderes Stylesheet inkludieren. Auf diese Weise lassen sich mehrere Stylesheets miteinander kombinieren, ohne dass dabei die Semantik der einzelnen Stylesheets verändert wird.

Dem Element `xsl:include` wird das Attribut `href` zugewiesen. Diesem Attribut übergibt man als Wert den URI des zu inkludierenden Stylesheets. Die innerhalb des inkludierten Stylesheets stehenden Angaben werden in das aktuelle Dokument eingebunden. Das geschieht exakt an der Stelle, an der der Aufruf stattgefunden hat.

Beachten Sie, dass `xsl:include` ausschließlich als Element der obersten Ebene erlaubt ist. Es kann demzufolge auch nur innerhalb von `xsl:stylesheet` stehen, muss aber außerhalb von `xsl:template` definiert werden.

Auch hierzu wieder ein Beispiel:

Listing 7.39 Das ist das Ausgangsdokument.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="inkludieren.xsl" ?>
<bibliothek>
   <name>Arno Geiger</name>
   <foto>geiger.gif</foto>
</bibliothek>
```

Über das `href`-Attribut wird das Stylesheet `inkludieren.xsl` eingebunden. Die Syntax dieser Datei sieht folgendermaßen aus:

Listing 7.40 Die zweite Datei wird eingebunden.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:include href="inkludieren2.xsl" />

<xsl:template match="/">
   <html>
      <head>
      </head>
   <body>
      <xsl:apply-templates/>
   </body>
   </html>
</xsl:template>

<xsl:template match="foto">
   <img>
      <xsl:attribute name="src">
         <xsl:value-of select="." />
      </xsl:attribute>
   </img>
</xsl:template>
</xsl:stylesheet>
```

Innerhalb dieses Stylesheets wird die über das `href`-Attribut angegebene XSL-Datei `inkludieren2.xsl` als XML-Dokument inkludiert. Der Inhalt der `inkludieren2.xsl` sieht folgendermaßen aus:

Listing 7.41 So sieht das zweite Stylesheet aus.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="name">
   <p>
      <xsl:value-of select="." />
   </p>
</xsl:template>
</xsl:stylesheet>
```

Im aktuellen Beispiel wird dafür gesorgt, dass oberhalb des eingebundenen Fotos ein durch `p` gekennzeichneter Textabsatz generiert wird.

Abbildung 7.14 Das Bild wird angezeigt.

An dieser Stelle noch ein Hinweis auf eine geläufige Fehlerquelle. Stylesheets dürfen sich nicht selbst inkludieren. Versucht man dies dennoch, bekommt man eine entsprechende Fehlermeldung angezeigt. Im Internet Explorer sieht die aus wie in **Abbildung 7.15**.

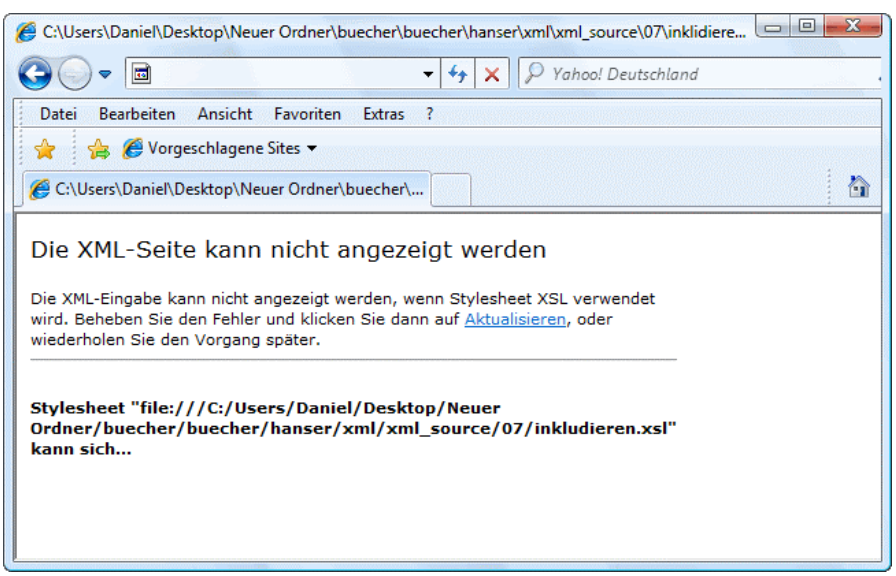

Abbildung 7.15 Hier stimmt etwas nicht.

7.9 Mit XSLT arbeiten

Auf den folgenden Seiten wird gezeigt, wie sich XSLT praktisch einsetzen lässt. Dabei werden Sie sehen, dass XSLT mehr kann, als Sie es von CSS oder anderen Stylesheet-Sprachen her gewohnt sind. Denn in der Tat kann man in XSLT mit Variablen arbeiten, Bedingungen definieren und Schleifen einsetzen. XSLT besitzt somit also viele Elemente einer klassischen Programmiersprache.

7.9.1 Variablen und Parameter einsetzen

Für CSS sind Variablen bereits seit Längerem im Gespräch, allerdings bis heute nicht verfügbar. XSLT-Entwickler haben es da besser. Denn XSLT unterstützt den Einsatz von Variablen und Parametern. Parameter sind dabei zusätzliche Kontextangaben, über die man die Template-Instanziierung beeinflussen kann. Zunächst erhalten Sie in diesem Kapitel allgemeine Informationen sowie ein einleitendes Beispiel zu diesem Thema. Danach werden die einzelnen Aspekte von Variablen und Parametern ausführlich betrachtet.

Damit man mit Parametern arbeiten kann, müssen diese innerhalb des zu instanziierenden Templates deklariert werden. Verwendet wird dafür innerhalb des Templates das Element `xsl:param`.

Parameter können an eine Template-Regel übergeben werden, wenn sie durch eines der folgenden Elemente aufgerufen werden:

- `xsl:apply-templates`
- `xsl:call-templates`

In beiden Fällen werden die Parameter jeweils mittels `xsl:with-param` festgelegt. Für die Deklaration des Parameters wird `xsl:param` verwendet. Dieses Element kennt zwei Attribute.

- `name` – Definiert den Namen der Parametervariablen. Über diesen Namen kann der Zugriff auf den Wert erfolgen.
- `select` – Bestimmt einen Standardwert für die Parametervariable. Wenn dieses Attribut angegeben wird, handelt es sich um einen Parameter, der dafür da ist, von einer Template-Definition an eine andere übergeben zu werden.

Um das parametrisierte Template zu instanziieren und den Parametern Werte zuweisen zu können, wird `xsl:with-param` verwendet. Auch dieses Element kennt zwei Attribute.

- `name` – Gibt den Namen des Parameters an. Unter diesem Namen muss der Parameter innerhalb der Template-Definition verfügbar sein. Damit er das ist, muss er mittels `xsl:param` definiert werden. (Wie das funktioniert, wurde zuvor gezeigt.)
- `select` – Hierüber wird dem Parameter ein Wert zugewiesen. Das kann z.B. ein Knoten aus einer XML-Datei oder statischer Text sein.

Elemente des Typs `xsl:with-param` dürfen ausschließlich innerhalb von `xsl:call-template` oder `xsl:apply-templates` stehen.

Interessant ist natürlich die Frage, wie sich Parameter in der Praxis einsetzen und nutzen lassen. Auch dazu wieder ein Beispiel.

Listing 7.42 Das ist das Ausgangsdokument.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="parameter.xsl" ?>
<bibliothek>
   <beginn>5</beginn>
   <ende>20</ende>
</bibliothek>
```

In diesem Beispiel wurden zwei Zahlenwerte definiert. Bei denen handelt es sich um einen Start- (`beginn`) und um einen Endwert (`ende`). Durch ein entsprechendes Stylesheet soll aus dem vorliegenden XML-Dokument eine HTML-Tabelle generiert werden. Innerhalb dieser Datei ist für alle Ganzzahlen zwischen Anfangs- und Endwert jeweils eine eigene Tabellenzelle enthalten. In jeder dieser Zellen werden die Ganzzahl und deren berechnetes Quadrat ausgegeben. Aussehen wird das Ganze im Ergebnis dann folgendermaßen:

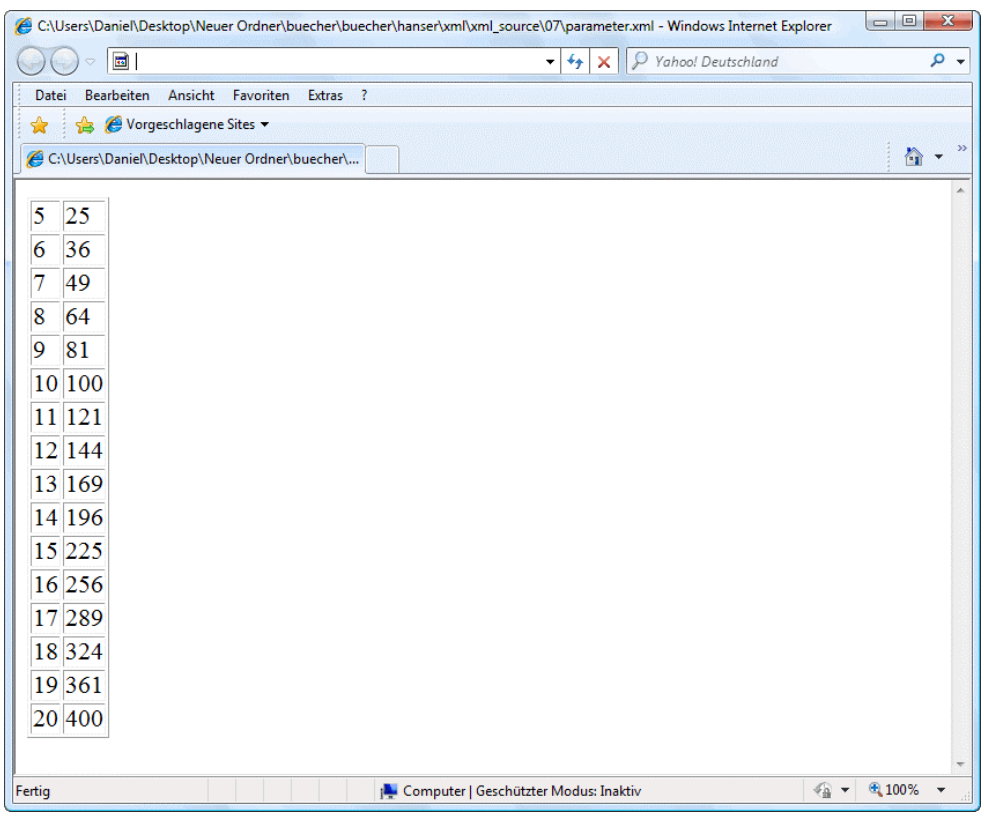

Abbildung 7.16 Die Ausgabe im Browser

Nachdem Sie das Ergebnis gesehen haben, geht es nun an die eigentliche XSLT-Syntax. Diese sieht folgendermaßen aus:

Listing 7.43 Zahlen werden in eine Tabelle geschrieben.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
   <html>
      <head>
      </head>
   <body>
      <table border="1">
         <xsl:call-template name="Durchlauf">
            <xsl:with-param name="Zahl" select="number
         (/bibliothek/beginn)" />
         </xsl:call-template>
      </table>
   </body>
   </html>
</xsl:template>

<xsl:template name="Durchlauf">
   <xsl:param name="Zahl" />
   <xsl:choose>
      <xsl:when test="$Zahl &lt;=number(/bibliothek/ende)">
         <tr>
            <td>
             <xsl:value-of select="$Zahl" />
            </td>
            <td>
              <xsl:value-of select="$Zahl * $Zahl" />
            </td>
         </tr>
         <xsl:call-template name="Durchlauf">
           <xsl:with-param name="Zahl" select="$Zahl + 1" />
         </xsl:call-template>
      </xsl:when>
      <xsl:otherwise>
         <xsl:call-template name="Ende" />
      </xsl:otherwise>
   </xsl:choose>
</xsl:template>

<xsl:template name="Ende">
   <xsl:comment>Fertig!</xsl:comment>
</xsl:template>

</xsl:stylesheet>
```

Innerhalb der Template-Definition für das Wurzelelement wird mittels `xsl:call-template` die Template-Definition `Durchlauf` aufgerufen. Der Parameter `Zahl` wird innerhalb von `xsl:call-template` definiert. Diesem Parameter wird als Wert des `select`-Attributs der Inhalt des `beginn`-Elements zugewiesen.

Dieses Beispiel macht noch einen weiteren Aspekt deutlich. Es zeigt, wie man überprüfen kann, ob es sich bei dem Inhalt eines Elements tatsächlich um eine Zahl handelt. Dafür kann die Funktion `number()` verwendet werden.

Innerhalb der aufgerufenen Template-Definition `Durchlauf` wird der `Zahl`-Parameter definiert. Das geschieht über `<xsl:param name="Zahl" />`.

Im nächsten Schritt muss überprüft werden, ob der Wert von `Zahl` kleiner oder gleich dem Wert ist, der in `ende` gespeichert ist. Realisiert wird das folgendermaßen:

```
<xsl:when test="$Zahl &lt;= number(/bibliothek/ende)">
```

7.9.1.1 Globale Parameter definieren

Es besteht auch die Möglichkeit, einem Stylesheet insgesamt Parameter zu übergeben. Das muss allerdings vom Prozessor unterstützt werden, was leider nicht immer gegeben ist. In diesem Fall werden aus den übergebenen Parametern Top-Level-Elemente, die als Kinder von `xsl:stylesheet` deklariert werden.

Wie sich globale Parameter nutzen lassen, zeigt das folgende Beispiel:

Listing 7.44 Parameter werden verwendet.

```
<xsl:output method="html"/>
   <xsl:param name="schriftgroesse">
      20pt
   </xsl:param>

   <xsl:template match="winery">
      <b><font size="{$ schriftgroesse}">
      <xsl:apply-templates/>
      <xsl:text> </xsl:text>
      <xsl:value-of select="../@grape"/></font></b><br/>
   </xsl:template>
```

Hier wird der Parameter `schriftgroesse` definiert. Zum Einsatz kommt für die Transformationen in diesem Beispiel der Saxon XSLT Processor. Diesem Prozessor kann ein Parameter folgendermaßen übergeben werden.

java com.icl.saxon.StyleSheet -x org.apache.xerces.parsers.SAXParser

-y org.apache.xerces.parsers.SAXParser xq338.xml

xq348.xsl schriftgroesse =8pt

Ganz ähnlich sähe es aus, wenn der XSLT-Prozessor Xalan verwendet wird.

java org.apache.xalan.xslt.Process – in eingabe.xml

-xsl ausgabe.xsl –param schriftgroesse 8pt

7.9.1.2 Parameter übergeben

Wie sich Parameter verwenden lassen, wurde auf den vorherigen Seiten gezeigt. Nun geht es darum, wie man diese an Templates übergeben kann. Das Element `xsl:with-param`, das dafür eingesetzt wird, haben Sie bereits kennengelernt. Und auch dessen mögliche Attribute `name` und `select` wurden vorgestellt.

Elemente des Typs `xsl:with-param` dürfen ausschließlich von Elementen der Typen `xsl:call-template` oder `xsl:apply-templates` verwendet werden.

Wurde der Wert des Parameters über `select` bestimmt, erhält man einen der folgenden vier Basistypen:

- String
- Zahl
- Boolean
- Node-Set

Das gilt allerdings nur, wenn keine erweiterte Funktion aufgerufen wird. In diesem Fall gibt es auch andere Ergebnistypen. Mit den erhaltenen Typen wird dann weitergearbeitet.

Wie sich Parameter übergeben lassen, zeigt folgendes Beispiel.

Listing 7.45 Die Definition eines Bildelements.

```
<xsl:template match="image">
   <xsl:param name="align" select="'left'" />
   <img src="images" align="{$align}" />
</xsl:template>
```

In dieser Syntax wurde das `img`-Element definiert. Über dessen `align`-Parameter wird dessen Ausrichtung bzw. die des Bildes bestimmt. Als Wert können diesem Attribut z.B. `left` (für eine linksbündige Ausrichtung) oder `right` (für eine rechtsbündige Ausrichtung) übergeben werden.

Listing 7.46 Das element wird rechtsbündig ausgerichtet.

```
<xsl:apply-templates select="images/image[1]">
   <xsl:with-param name="align"  select="'right'"/>
</xsl:apply-templates>
```

In diesem Beispiel wurde `right` gewählt, was eine rechtsbündige Ausrichtung des Bildes zur Folge hat.

7.9.2 Variablen verwenden

Wer den Umgang mit Variablen aus „echten“ Programmiersprachen gewohnt ist, wird in diesem Zusammenhang von den XSLT-Variablen zunächst enttäuscht sein. Denn in der Tat kennt XSLT Variablen nur in einem sehr begrenzten Rahmen.

Variablen kann man deklarieren und ihnen einen Wert zuweisen. Allerdings kann der zugewiesene Wert in XSLT nicht mehr verändert werden.[2]

Für die Deklaration einer Variablen wird das Element `xsl:variable` verwendet. Dieses Element lässt sich auf zwei Arten einsetzen.

- Als Top-Level-Element, um eine globale Konstante zu deklarieren.
- Lokal innerhalb eines Templates.

Ein Template kann keinen Variablenwert an ein anderes Template übergeben. Die einzelnen Template-Regeln haben somit keinerlei Nebeneffekte. Das ist so übrigens beim Entwurf der XSLT-Spezifikation gewünscht gewesen, um den Umgang mit Templates zu vereinfachen.

Zunächst wird in diesem Kapitel die allgemeine Verwendung von Variablen vorgestellt. Anschließend geht es mehr ins Detail. Die folgenden Aspekte lassen sich meistens auch auf Parameter anwenden.

[2] In der Tat gibt es aber dennoch eine Möglichkeit, genau das zu tun. Verwendet wird dafür eine Erweiterung für den Saxon-Prozessor. Mehr zu diesem Thema erfahren Sie im weiteren Verlauf dieses Kapitels.

7.9.2.1 Variablen verwenden

Die Syntax von `xsl:variable` entspricht weitestgehend der von `xsl:param`. Allerdings lassen sich bei `xsl:variable` keinerlei Standardwerte angeben. Zudem darf ein `xsl:variable`-Element nicht nur als erstes Kindelement innerhalb des Templates stehen, sondern kann überall dort verwendet werden, wo es benötigt wird.

Wird eine Variable innerhalb eines Templates eingefügt, muss auf den Geltungsbereich geachtet werden. So kann der Variablenwert nur von den nachfolgenden Geschwisterelementen und deren Nachkommen verwendet werden. Enthält das Variablenelement selbst keine Kindelemente, können sie sich nicht auf den Variablenwert beziehen.

Wie sich Variablen nutzen lassen, zeigt folgendes Beispiel:

Listing 7.47 Die Variable wird deklariert.

```
<xsl:variable name="aname">
   @Name="Nick Hornby"
</xsl:variable>
<xsl:value-of select="Autor[$aname]"/>
```

Deklariert wird hier die Variable `aname`. Der Zugriff auf die Variable erfolgt über das `$`-Zeichen.

Und noch ein Beispiel zum Gültigkeitsbereich von Variablen.

Listing 7.48 Die Variable gilt nur in dem einen Element und dessen Unterelementen.

```
<xsl:if test="position()=1">
   <xsl:variable name="aname" select="..."/>
</xsl:if>

<xsl:if test="position()=last()">
   <xsl:variable name="aname" select="..."/>
</xsl:if>
```

Die Variablen werden in diesem Beispiel innerhalb von `xsl:if` verwendet und gelten auch ausschließlich innerhalb dieses Elements und dessen Unterelementen.

7.9.2.2 Eindeutige Namen

Eine häufige Fehlerquelle ist die Verwendung nicht eindeutiger Namen. Achten Sie daher unbedingt darauf, dass Variablen- und Parameternamen in einem Template jeweils nur einmal verwendet werden dürfen.

Folgendes Template wäre demnach fehlerhaft:

Listing 7.49 Das ist nicht erlaubt.

```
<xsl:template name="image">
   <xsl:param name="align"/>
   <xsl:param name="align"/>
</xsl:template>
```

Es ist aber möglich, innerhalb eines Template eine globale Definition einer Variablen oder eines Parameters zu überschreiben. Dafür muss für die lokale Variable oder den lokalen Parameter derselbe Name verwendet werden.

7.9.2.3 Variablen und Parameterwerte

Variablenbindende Elemente können den Wert einer Variablen auf drei unterschiedliche Arten annehmen.

- Wenn das variablenbindende Element ein `select`-Attribut besitzt, muss es sich bei dem Attributwert um einen Ausdruck handeln. Der Wert der Variable ist dabei das Objekt, das das Resultat aus der Auswertung des Ausdrucks ist. In diesem Fall muss der Inhalt leer sein.
- Besitzt das variablenbindende Element kein `select`-Attribut und einen nicht leeren Inhalt, wird der Wert durch den Inhalt des variablenbindenden Elements angegeben. Bei dem Inhalt des variablenbindenden Elements handelt es sich um ein Template, das instanziiert wird, um den Variablenwert zurückzugeben. Das Ergebnis ist ein Ergebnisbaum-Fragment, das einer Knotenmenge mit nur einem Wurzelknoten entspricht, der als Kinder die Folge von Knoten enthält, die durch die Instanziierung des Templates entstehen. Bei dem Basis-URI der Knoten im Ergebnisbaum-Fragment handelt es sich um den Basis-URI des variablenbindenden Elements.
- Hat das variablenbindende Element einen leeren Inhalt und besitzt es kein `select`-Attribut, handelt es sich bei dem Variablenwert um eine leere Zeichenkette.

Der zuletzt genannte Punkt soll anhand eines Beispiels verdeutlicht werden. In der Tat ist es nämlich so, dass die Syntax

```
<xsl:variable name="x"/>
```

das Gleiche wie die folgende Syntax bewirkt:

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

Und dann ist noch auf ein paar Dinge hinsichtlich der Variablen zu achten. So sollte Folgendes vermieden werden:

Listing 7.50 Das ist nicht korrekt.

```
<xsl:variable name="n">
   2
</xsl:variable>

   ...

<xsl:value-of select="item[$n]"/>
```

Durch diese Syntax wird der Wert des ersten Elements ausgegeben. Denn in dieser Syntax ist die Variable `n` an ein Ergebnisbaum-Fragment gebunden, nicht aber an eine Zahl. Vermeiden lässt sich dieses Problem durch:

Listing 7.51 So stimmt es.

```
<xsl:variable name="n" select="2"/>

   ...

<xsl:value-of select="item[$n]"/>
```

oder durch Folgendes:

Listing 7.52 Auch das ist möglich.

```
<xsl:variable name="n">
   2
</xsl:variable>

   ...

<xsl:value-of select="item[position()=$n]"/>
```

Und noch ein Hinweis in diesem Zusammenhang. Oftmals benötigt man eine leere Knotenmenge als Standardwert eines Parameters. Diese kann man folgendermaßen angeben:

```
<xsl:param name="x" select="/.."/>
```

7.9.2.4 Variablen und Parametervariablen mit `xsl:copy-of`

`xsl:copy-of` wird normalerweise dafür genutzt, das gesamte von dem angegebenen Knoten abhängige Knoten-Set in den Ergebnisbaum zu kopieren. Interessant ist dieses Element aber auch im Zusammenhang mit Variablen und Parametervariablen. Denn mit `xsl:copy-of` kann man ein Ergebnisbaum-Fragment in einen Ergebnisbaum einfügen, ohne dass dieses zuvor in eine Zeichenkette umgewandelt werden muss, wie es bei `xsl:value-of` geschieht.

Dabei enthält das `select`-Attribut einen entsprechenden Ausdruck. Handelt es sich bei dem Ergebnis der Auswertung um ein Ergebnisbaum-Fragment, wird das gesamte Fragment in den Ergebnisbaum kopiert. Ist das Ergebnis eine Knotenmenge, werden alle Knoten dieser Menge in der Dokumentreihenfolge in den Ergebnisbaum kopiert.

Durch das Kopieren eines Elementknotens werden die folgenden Elemente kopiert:

- Attributknoten
- Namensraumknoten
- Kinder des Elementknotens
- der Elementknoten selbst

Ein Wurzelknoten wird kopiert, indem seine Kinder kopiert werden.

Handelt es sich beim Ergebnis weder um eine Knotenmenge noch um ein Ergebnisbaum-Fragment, wird das Ergebnis in eine Zeichenkette umgewandelt und anschließend in den Ergebnisbaum eingefügt.

7.9.2.5 Variablen und Parameter auf der obersten Ebene

Die beiden Elemente `xsl:variable` und `xsl:param` können auf der obersten Ebene genutzt werden, wobei als ein Element der obersten Ebene ein als Kind eines `xsl:stylesheet`-Elements vorkommendes Element bezeichnet wird. Ein variablenbindendes Element auf der obersten Ebene deklariert eine globale Variable, die überall verfügbar ist.

Wenn man `xsl:param` auf der obersten Ebene verwendet, wird ein Parameter für das Stylesheet vereinbart. XSLT stellt keinen Mechanismus zur Verfügung, mit dem Parameter an das Stylesheet übergeben werden können.

Sollte ein Stylesheet mehr als eine Bindung einer Variablen auf der obersten Ebene mit demselben Namen und derselben Importpriorität enthalten, ist das ein Fehler.

Auf der obersten Ebene wird der Ausdruck oder das Template, das den Variablenwert spezifiziert, mit demselben Kontext ausgewertet, der auch benutzt wird, um den Wurzelknoten des Dokuments zu verarbeiten.

Im folgenden Beispiel wird die globale Variable `para-font-size` deklariert, die in einem Attributwert-Template referenziert wird.

Listing 7.53 Hier wurde eine globale Variable deklariert.

```
<xsl:variable name="para-font-size">12pt</xsl:variable>
<xsl:template match="para">
   <xsl:block font-size="{$para-font-size}">
      <xsl:apply-templates/>
   </xsl:block>
</xsl:template>
```

7.9.2.6 Variablen und Parameter innerhalb von Templates

`xsl:variable` und `xsl:param` sind nicht nur auf der obersten Ebene erlaubt, man kann sie auch innerhalb von Templates einsetzen. Dabei lässt sich `xsl:variable` an allen Stellen innerhalb eines Templates nutzen, an denen auch eine Anweisung erlaubt wäre. In einem solchen Fall ist die Bindung dann für alle folgenden Geschwister und deren Nachfahren sichtbar. Die Bindung ist für das `xsl:variable`-Element selbst allerdings nicht sichtbar.

`xsl:param` kann als Kindelement am Anfang eines `xsl:template`-Elements genutzt werden. In diesem Fall ist die Bindung für alle folgenden Geschwister und deren Nachfahren sichtbar. Auch in diesem Zusammenhang gilt wieder, dass die Bindung für das `xsl:param`-Element selbst nicht sichtbar ist.

Noch einige erklärende Worte zum Thema Bindung: Eine Bindung verdeckt eine andere Bindung, wenn sie an einem Punkt auftritt, an dem die andere Bindung sichtbar ist und die Bindungen den gleichen Namen tragen. Ein Fehler ist es, wenn eine durch `xsl:variable` oder `xsl:param` hergestellte Bindung innerhalb eines Templates eine andere Bindung innerhalb des Templates verdeckt, die ebenfalls durch ein `xsl:variable`- oder `xsl:param`-Element hergestellt wurde.

Im Gegensatz dazu ist es aber kein Fehler, wenn eine Bindung, die durch ein `xsl:variable`- oder `xsl:param`-Element in einem Template hergestellt wurde, eine andere Bindung verdeckt, die durch ein `xsl:variable` oder ein `xsl:param` auf oberster Ebene hergestellt wurde.

Folgendes wäre demnach falsch:

Listing 7.54 Das ist falsch.

```
<xsl:template name="autoren">
   <xsl:param name="nname" select="1"/>
   <xsl:variable name="nname" select="2"/>
</xsl:template>
```

Im Gegenzug dazu ist Folgendes korrekt:

Listing 7.55 Diese Syntax stimmt.

```
<xsl:param name="nname" select="1"/>
<xsl:template name="autoren">
   <xsl:variable name="nname" select="2"/>
</xsl:template>
```

7.9.2.7 Parameter an Templates übergeben

Um einen Parameter an ein Template zu übergeben, wird das Element `xsl:with-param` verwendet. Über das `name`-Attribut wird der Parameter angegeben. `xsl:with-param` kann innerhalb von `xsl:call-template` und `xsl:apply-templates` definiert werden. Der Wert des Parameters wird genauso wie bei `xsl:variable` und `xsl:param` angegeben.

Im folgenden Beispiel wird ein benanntes Template für einen nummerierten Absatz mit einem Argument definiert, um auf diese Weise das Format der Nummerierung zu kontrollieren.

Listing 7.56 So wird die Nummerierung kontrolliert.

```
<xsl:template name="numbered-block">
  <xsl:param name="format">1. </xsl:param>
  <xsl:block>
    <xsl:number format="{$format}"/>
    <xsl:apply-templates/>
  </xsl:block>
</xsl:template>

<xsl:template match="ol//ol/li">
  <xsl:call-template name="numbered-block">
    <xsl:with-param name="format">a. </xsl:with-param>
  </xsl:call-template>
</xsl:template>
```

7.9.2.8 Änderungen bei Parametern und Variablen in XSLT 2.0

Gerade hinsichtlich der Parameter und Variablen hat sich in XSLT 2.0 viel getan. Eine der wesentlichsten Neuerungen besteht in der Typbindung der Werte, die man über das Attribut `AS` erzwingen kann.

In XSLT 2.0 entfällt darüber hinaus auch die automatische Umwandlung des Variableninhalts in ein sogenanntes Ergebnisbaum-Fragment. Stattdessen erzeugt ein Element des Typs Variable einen temporären Baum, auf den sich beliebige Funktionen und Template-Regeln anwenden lassen. Das gilt allerdings nur, wenn diese eine Knotensequenz als Eingabe erlauben.

Listing 7.57 So werden in XSLT 2.0 Variablen verwendet.

```
<xsl:transform version="2.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:param name="hanser">
   <a>
      <b>
         <c/>
      </b>
```

```
   </a>
</xsl:param>

<xsl:template match="/">
   <xsl:apply-templates select="$hanser/*"/>
</xsl:template>

<xsl:template match="*">
   <xsl:element name="{upper-case(local-name(.))}">
      <xsl:apply-templates/>
   </xsl:element>
</xsl:template>

</xsl:transform>
```

In der gezeigten Syntax wird ein einfaches Dokument erzeugt, das in der Variablen mit dem Namen `hanser` gespeichert wird. Dazu wird zunächst ein temporärer Baum mit den drei Elementknoten

- `a`,
- `b` und
- `c`

angelegt. Dieser Baum wird in einen Ergebnisbaum mit der gleichen Struktur transformiert. Dabei werden die Namen der Elementknoten mittels `upper-case` in Großbuchstaben umgewandelt. Wie üblich startet die Transformation mit der Auswertung der Template-Regel für den Wurzelknoten des Eingabedokuments, die dafür sorgt, dass die Regel auf das lokal erzeugte Dokument angewendet wird. Im gezeigten Beispiel wird das Eingabedokument ignoriert.

Die Transformation des temporären Baums beginnt im gezeigten Beispiel mit dem Dokumentelement, um auf diese Weise eine erneute Anwendung der Template-Regel für den Wurzelknoten vermeiden zu können. Wäre das nicht der Fall, würde die Regelanwendung rekursiv wiederholt werden, ohne dass das jemals zu einem Ergebnis führen würde.

Eine weitere Neuerung betrifft den Elementtyp `param`. Für den gibt es in XSLT 2.0 gleich drei neue Attribute.

- `as` – Dieses Attribut kann man verwenden, um den Datentyp des Parameters zu spezifizieren. Besitzt ein Parameter einen abweichenden Typ, der nicht in den angegebenen Typ gewandelt werden kann, beendet der XSLT-Prozessor die Transformation mit einer Fehlermeldung.
- `required` – Hierüber lässt sich bestimmen, ob die globalen Parameter des Stylesheets und die lokalen Parameter der Template-Regeln obligatorisch sind. Mit `yes` legt man fest, dass der Parameter angegeben werden muss. In diesem Fall darf die Spezifikation des formalen Parameters weder einen Sequenzkonstruktor noch ein `select`-Attribut enthalten.
- `tunnel` – Für den Einsatz dieses Parameters innerhalb einer Template-Regel muss dem `tunnel`-Parameter der Wert `yes` zugewiesen werden. `tunnel`-Parameter gehören dem Aufrufkontext an und sind innerhalb der gesamten Aufrufkette verschachtelter Template-Regeln sichtbar. Die Initialisierung funktioniert wie bei `xsl:with-param`, wobei hier auch `tunnel="yes"` angegeben werden muss.

Durch die `tunnel`-Parameter und die temporären Baume erhöht XSLT 2.0 die Fähigkeiten von XSLT hinsichtlich der Parameter und Variablen enorm.

7.9.3 Bedingte Anweisungen

Genauso wie in „echten“ Programmiersprachen ist es auch in XSLT möglich, bedingte Anweisungen zu definieren. Dabei kennt XSLT einfache und erweiterte Bedingungen. So kann innerhalb von Templates auf bestimmte Bedingungen reagiert werden.

Bei der einfachen Fallunterscheidung wird die `if`-Anweisung verwendet. Diese kennen Sie vielleicht aus anderen Sprachen wie zum Beispiel PHP oder JavaScript. Allerdings funktioniert die `if`-Anweisung in XSLT nur mit einem einzigen Fall. Alternativen wie `elseif` gibt es in XSLT leider nicht.

Wenn eine angegebene Bedingung erfüllt ist, werden die nachfolgenden Anweisungen ausgeführt. Die Syntax ist denkbar einfach. Die zu erfüllende Bedingung wird innerhalb des Elements `xsl:if` angegeben, und zwar über den Wert des Attributs `test`. Bei diesem Wert handelt es sich um einen Ausdruck, dessen Ergebnis `true` oder `false`, also wahr oder falsch ist. Hierzu ein einfaches Beispiel:

Listing 7.58 So werden Bedingungen definiert.

```
<xsl:if test="@name">
   Name: <xsl:value-of select="@name"/>
</xsl:if>
```

In diesem Beispiel wird der Name nur dann ausgegeben, wenn auch tatsächlich ein Name angegeben wurde. Eine „leere“ Ausgabe wie

```
Name:
```

wird somit also in jedem Fall vermieden.

Sollte kein boolescher Wert zur Verfügung stehen, wird der Ausdruck in einen booleschen Ausdruck konvertiert. Bei Pfadangaben wird `true` dann zurückgeliefert, wenn

- ein String zurückgeliefert wird, der nicht leer ist,
- eine Zahl zurückgeliefert wird, die nicht ungleich `0` ist,
- ein Knotentest zurückgeliefert wird, der nicht leer ist.

Auch die Funktionsweise der Bedingungen lässt sich wieder am besten anhand eines Beispiels zeigen. Zunächst das Ausgangsdokument:

Listing 7.59 Das ist das Ausgangsdokument.

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

<bibliothek>
   <autor>Ellis</autor>
   <autor>Hornby</autor>
   <autor>Bryson</autor>

</bibliothek>
```

Es handelt sich dabei zunächst einmal um eine sehr einfache XML-Struktur. Durch entsprechende XSLT-Syntax soll daraus folgende Ausgabe generiert werden:

```
Ellis, Hornby, Bryson!
```

Alle innerhalb des Dokuments vorkommenden Elemente sollen jeweils durch ein Komma voneinander getrennt werden. Abgeschlossen werden soll die Reihe durch ein Ausrufezeichen. Die Herausforderung besteht in diesem Fall nun darin, dass Kommata und Ausrufezeichen an der richtigen Stelle stehen.

Um diese Anforderungen umzusetzen, muss für jedes Element explizit überprüft werden, ob es sich um das letzte Element handelt. Realisiert wird das über die beiden folgenden XPath-Funktionen:

- `position()` – Ermittelt die Positionsnummer des aktuellen Knotens.
- `last()` – Ermittelt die Positionsnummer des letzten von mehreren Knoten eines Knoten-Sets.

Die entsprechende XSLT-Syntax, bei der unter anderem diese beiden Funktionen eingesetzt werden, sieht folgendermaßen aus:

Listing 7.60 In diesem Beispiel werden Funktionen genutzt.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
   <html>
      <head>
         <title>Bibliothek</title>
      </head>
   <body>
      <h2>Bibliothek</h2>
   <xsl:apply-templates/>
   </body>
</html>
</xsl:template>

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

<xsl:template match="autor">
   <xsl:if test="position()!=last()">
      <xsl:value-of select="."/>,
   </xsl:if>

   <xsl:if test="position()=last()">
      <xsl:value-of select="."/>!
   </xsl:if>
</xsl:template>

</xsl:stylesheet>
```

Zunächst muss die Bedingung definiert werden, dass es sich bei dem aktuellen Element nicht um das letzte Element handelt.

```
<xsl:if test="position()!=last()">
```

Ist diese Bedingung erfüllt ist, wird hinter dem Element ein Komma gesetzt. Das sieht dann folgendermaßen aus:

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

Über die nächste `if`-Abfrage wird dann ermittelt, ob es sich bei dem aktuellen Element um das letzte Element handelt.

```
<xsl:if test="position()=last()">
```

Ist diese Bedingung erfüllt, wird hinter dem Element kein Komma, sondern ein Ausrufezeichen eingefügt.

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

Wie dieses Beispiel gezeigt hat, erinnert die Definition von Bedingungen in XSLT durchaus an Möglichkeiten, die Sie vielleicht von anderen Programmiersprachen her kennen. Damit sind die Optionen, die XSLT auf diesem Sektor zu bieten hat, übrigens längst noch nicht ausgeschöpft. Es besteht nämlich auch die Möglichkeit, erweiterte Bedingungen zu definieren.

7.9.4 Erweiterte Bedingungen definieren

Wie sich einfache Bedingungen definieren lassen, wurde auf den vorherigen Seiten gezeigt. Irgendwann stößt man mit dieser einfachen Variante allerdings an seine Grenzen, bzw. die Syntax wird dann zu umständlich.

Wer sich mit anderen Programmiersprachen auskennt, dem sind von dort für die Definition mehrerer Bedingungen z.B. folgende Konstrukte bekannt:

- `if else`
- `if – else if`

Auf diese Weise lassen sich gleich mehrere Bedingungen definieren. Über den else-Zweig lässt sich dann festlegen, was passieren soll, wenn keine der zuvor definierten Bedingungen erfüllt wurde. Mit der normalen bzw. einfachen XSLT-Syntax ist so etwas nicht oder nur mit sehr viel Aufwand zu bewältigen.

Listing 7.61 So sieht eine normale Bedingung aus.

```
<xsl:if test="position()!=last()">
   <xsl:value-of select="."/>,
</xsl:if>
```

Zum Glück hält XSLT eine Alternative zu einer solch einfachen Definition parat. Die Syntax ist dabei denkbar einfach. Das Element

```
xsl:choose
```

bildet den Rahmen für eine beliebige Anzahl an Abfragen. Diese Abfragen werden über die beiden Elemente

```
xsl:when
```

und

```
xsl:otherwise
```

realisiert. Dabei kann die Abfrage beliebig viele `xsl:when`-Elemente enthalten, muss aber eine abschließende `xsl:otherwise`-Anweisung besitzen. Von den `xsl:when`-Abfragen wird diejenige ausgewählt, deren Bedingung als erste erfüllt wird.

Die allgemeine Syntax für eine erweiterte Bedingung sieht folgendermaßen aus:

Listing 7.62 Das ist die Grundstruktur.

```
<xsl:choose>
   <xsl:when test="…….">
      ……
   </xsl:when>
   <xsl:when test="…….">
      ……
   </xsl:when>
   <xsl:when test="…….">
      ……
   </xsl:when>
   <xsl:otherwise>
      ……
   </xsl:otherwise>
</xsl:choose>
```

Anhand dieser Struktur werden die Gemeinsamkeiten im Vergleich zu anderen Programmiersprachen deutlich. So kann man `xsl:when` mit `if`-Abfragen oder `switch` vergleichen, während `xsl:otherwise` mit der `else`-Klausel gleichzusetzen ist.

Nachdem die Grundlagen der erweiterten Bedingungen vorgestellt wurden, soll das Ganze nun anhand eines Beispiels gezeigt werden. Ausgangspunkt dafür ist das folgende Dokument, in dem mehrere `ziffer`-Elemente definiert wurden.

Listing 7.63 Das ist das Ausgangsdokument.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="variablen.xsl" ?>
<werte>
   <ziffer>1</ziffer>
   <ziffer>5</ziffer>
   <ziffer>13</ziffer>
   <ziffer>4</ziffer>
   <ziffer>10</ziffer>
</werte>
```

Mittels einer einfachen Syntax werden diese einzelnen `ziffer`-Elemente dahingehend überprüft, ob sie größer, kleiner oder gleich 10 sind. Je nachdem, welches Ergebnis ermittelt wurde, wird eine entsprechende Meldung ausgegeben.

Listing 7.64 Die Bedingungen werden definiert.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

```
<xsl:template match="/">
   <html>
      <head>
      </head>
    <body>
       <h1>Zahlenspiel</h1>
          <xsl:apply-templates />
    </body>
    </html>
</xsl:template>

<xsl:template match="ziffer">
   <div>
      <xsl:value-of select="." />
      <xsl:variable name="wert" select="." />

      <xsl:choose>
         <xsl:when test="$wert &lt; 10">
            <xsl:text> (unter 10)</xsl:text>


    </xsl:when>
          <xsl:when test="$wert = 10">
            <xsl:text> (Volltreffer)</xsl:text>
         </xsl:when>

         <xsl:otherwise>
            <xsl:text> (groesser 10)
            </xsl:text>
         </xsl:otherwise>

      </xsl:choose>

   </div>
</xsl:template>

</xsl:stylesheet>
```

Der Wert des jeweils durchlaufenen `ziffer`-Elements wird innerhalb der `$wert`-Variablen gespeichert. Über die erweiterte Bedingung wird nacheinander Folgendes überprüft:

- Ist der Wert kleiner als `10`, dann gib `(unter 10)` aus.
- Ist der Wert `10`, dann gib `(Volltreffer)` aus.
- Trifft keine dieser Bedingungen zu, gib `(groesser 10)` aus.

Ein abschließender Blick auf das Ergebnisdokument darf natürlich nicht fehlen.

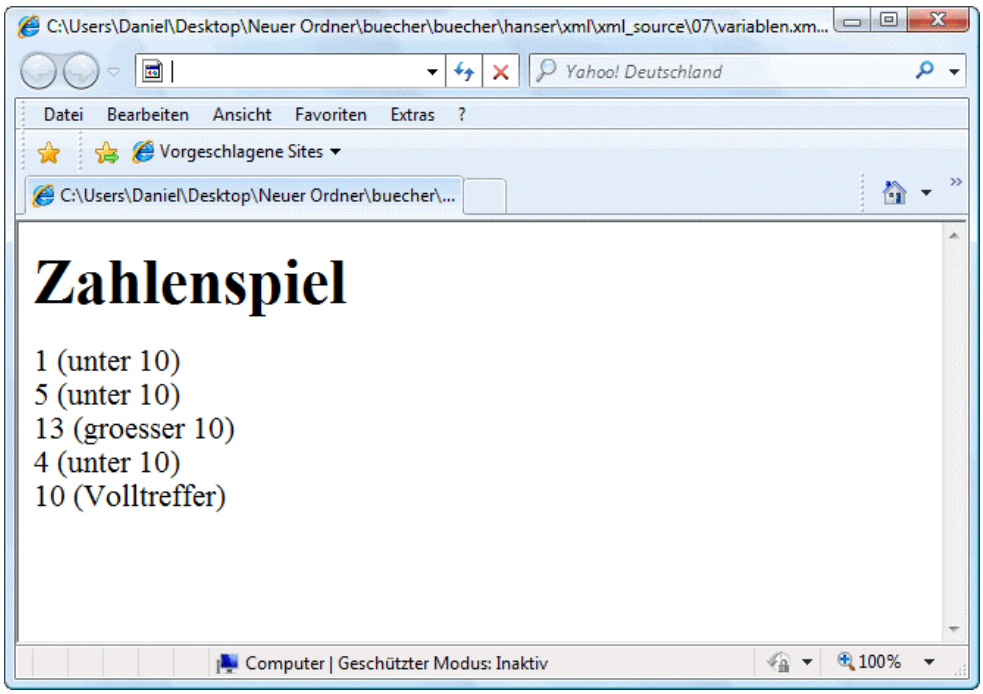

Abbildung 7.17 Die Ausgabe liefert das gewünschte Ergebnis.

Sie sehen also, dass auch in XSLT mit wenig Aufwand erweiterte Bedingungen definiert werden können.

7.9.5 Nummerierungen

In XSLT gibt es verschiedene Möglichkeiten, Daten nachträglich zu nummerieren. Das Element, um das sich dabei alles dreht, ist `xsl:number`. Durch zahlreiche Attribute lässt sich die Art der Nummerierung gezielt steuern.

7.9.5.1 Einfache Nummerierung

Im einfachsten Fall wird die Nummerierung ohne zusätzliche Attribute erzeugt. Dabei liefert das Element dann eine fortlaufende Nummerierung für alle Elemente der Knotenliste, zu der der Kontextknoten gehört.

Listing 7.65 Nummerierungen im Einsatz

```
<xsl:template match="autoren">
   <p>
      <xsl:number/>
      <xsl:text> </xsl:text>
      <xsl:value-of select="@name"/>
   </p>
</xsl:template>
```

Das würde dann zu einer einfachen nummerierten Liste nach folgendem Schema führen:

- `Hornby`
- `Parsons`
- `Ellis`
- `Roth`

Das Element `xsl:number` kennt eine Vielzahl an Attributen. Die folgenden Varianten sind möglich:

- `count` – Gibt das oder die Knoten-Sets bzw. Pfade an, bei denen der Zähler erhöht werden soll.
- `format` – Hierüber wird bestimmt, wie die Nummerierung aussehen soll. Die möglichen Werte finden Sie in der nachfolgenden Tabelle.
- `from` – Hierüber wird festgelegt, an welchem Knoten die Nummerierung beendet werden soll.
- `grouping-separator` – Damit legt man das Trennzeichen für Gruppierungen fest. Ein typisches Beispiel sind Tausender-Trennzeichen, die üblicherweise durch einen Punkt (1.000.000) gekennzeichnet werden.
- `grouping-size` – Über dieses Attribut wird die Anzahl der Ziffern festgelegt, bei der man ein Trennzeichen setzt. Im Fall der Tausender-Schritte wäre das die `3`.
- `lang` – Hierüber legt man die Sprache fest, nach deren Konventionen die Nummerierung erfolgen soll. Für Deutschland wählt man de, für England en usw.
- `letter-value` – Bei einigen Sprachen kann zusätzlich zu lang dieses Attribut mit einem der beiden Werte `alphabetical` oder `traditional` angegeben werden. Dazu muss die angegebene Sprache aber auch eine alphabetische bzw. traditionelle Zählweise unterstützen.
- `level` – Legt fest, ob die Aufzählung nur auf einer (`single`), mehreren (`multiple`) oder allen (`any`) Zweigen des Ausgangsdokuments fortgeführt werden soll.
- `value` – Ein Kalkulationsausdruck, über den die Nummerierung bestimmt wird.

7.9.5.2 Mehrstufige Nummerierung

Neben einfachen Nummerierungen, wie sie im vorherigen Abschnitt gezeigt wurden, sind auch mehrstufige möglich. Dabei klingt mehrstufig zunächst zugegebenermaßen etwas sperrig. Sie kennen solche mehrstufigen Nummerierungen allerdings von Textdokumenten her. Entscheidend dafür, wie der Prozessor beim Nummerieren vorgeht, ist das Attribut `level`, das bereits kurz im vorherigen Abschnitt gezeigt wurde. Dieses Attribut kennt die folgenden drei Werte:

- `any` – Hierüber lassen sich Knoten nummerieren, die auf unterschiedlichen Ebenen vorkommen können. Dabei kann es sich dann zum Beispiel um Grafiken, Tabellen oder Fußnoten handeln.
- `multiple` – Hierüber können zusammengesetzte Nummerierungen realisiert werden.

- `single` – Hier werden Geschwisterknoten gezählt, die auf einer Ebene liegen. Dabei handelt es sich um die Voreinstellung.

Mit dem Attribut `count` lässt sich die Übereinstimmung mit einem XPath-Muster überprüfen. Ebenso kann mit `from` ein Muster verwendet werden, über das ein anderer Startpunkt gesetzt wird.

In diesem Fall, muss der Prozessor bei jeder Nummer, die zu vergeben ist, vom aktuellen Kontextknoten aus die `ancestor-or-self`-Achse zurückgehen, um den ersten Knoten zu ermitteln, der dem expliziten oder vorgegebenen `count`-Muster entspricht. Besitzt `from` einen Wert, stoppt der Prozessor allerdings bereits an der dem `from`-Muster entsprechenden Position.

Für die Definition des Zahlenformats werden Format-Token verwendet, die den Wert des `format`-Attributs liefern.

Tabelle 7.2: Diese Werte stehen zur Auswahl.

Wert	Art der Nummerierung
`1`	`1, 2, 3, 4 …`
`01`	`01, 02, 03, 04 …`
`a`	`a, b, c, d …`
`A`	`A, B, C, D …`
`i`	`I, ii, iii, iv …`
`I`	`I, II, III, IV …`

7.9.6 Sortieren und Gruppieren

Normalerweise verarbeitet der XSLT-Prozessor eine mittels `select` ausgewählte Knotenliste in der Reihenfolge, die durch das Quelldokument vorgegeben ist. Nun ist das natürlich nicht immer gewünscht. Zum Glück stellt XSLT Möglichkeiten zur Verfügung, mit denen die Verarbeitungsreihenfolge explizit geändert werden kann. Auf den folgenden Seiten werden die verschiedenen Varianten zum Sortieren und Gruppieren vorgestellt.

Dabei wird Ihnen immer wieder der Begriff Schlüssel begegnen. Durch einen solchen Schlüssel wird der gezielte Zugriff auf Knoten möglich.

7.9.6.1 Daten sortieren

Will man mit mehreren Schlüsseln arbeiten, um Duplikate innerhalb des vorausgehenden Schlüssels nach einem zusätzlichen Kriterium zu sortieren, kann für jeden Schlüssel ein eigenes `xsl:sort`-Element verwendet werden. Die Zahl der Schlüssel ist dabei nicht begrenzt. Allerdings muss man die Reihenfolge von der größeren zur kleineren Knotenmenge beachten.

`xsl:sort` kennt die folgenden Attribute:

- `case-order` – Dieses Attribut bestimmt, ob in der Sortierung Großbuchstaben vor Kleinbuchstaben angezeigt werden. Mögliche Werte sind `upper-first` (Großbuchstaben zuerst) und `lower-first` (Kleinbuchstaben zuerst).
- `data-type` – Bestimmt, ob die Sortierung numerisch oder alphabetisch erfolgen soll. Durch `text` wird eine alphabetische Sortierung erreicht, `number` sorgt für eine numerische Sortierung. In zukünftigen XSLT-Versionen sollen auch noch andere Datentypen, die in der XML-Spezifikation definiert sind, angegeben werden können.
- `lang` – Hierüber wird die Sprache bestimmt, nach deren Konventionen die Sortierung erfolgen soll. Erwartet wird, wenn man dieses Attribut setzt, ein Sprachkürzel wie `en`, `fr` oder `de`. Lässt man das Attribut weg, verwendet der Prozessor automatisch die Sprache der Systemumgebung.
- `order` – Durch dieses Attribut legt man fest, ob die Sortierung aufsteigend (z.B. von `A` bis `Z`) oder absteigend (z.B. von `Z` bis `A`) erfolgen soll. Mögliche Werte sind `ascending` (aufsteigend = Voreinstellung) und `descending` (absteigend).
- `select` – Darüber bestimmt man, was eigentlich sortiert werden soll. Fehlt dieses Attribut, wird der Inhalt des betroffenen Elements sortiert.

Zwei eigentlich konforme XSLT-Prozessoren müssen nicht unbedingt gleich sortieren. So kann es durchaus sein, dass einige XSLT-Prozessoren bestimmte Sprachen nicht unterstützen.

Verwendet werden kann `xsl:sort` innerhalb der beiden Elemente `xsl:apply-templates` und `xsl:for-each`.

Listing 7.66 Das ist das Ausgangsdokument.

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

   <autor name="Parsons">
      <buch>
         <titel>One for my baby</titel>
         <isbn>3492261469</isbn>
         <verlag>Piper</verlag>
      </buch>
   </autor>

   <autor name="Bryson">
      <buch>
         <titel>Eine kurze Geschichte
                von fast allem</titel>
         <isbn>3442460719</isbn>
         <verlag>Goldmann</verlag>
      </buch>
   </autor>

   <autor name="Roth">
      <buch>
         <titel>Verschwörung
                gegen Amerika</titel>
         <isbn>3499240874</isbn>
         <verlag>Rowohlt</verlag>
      </buch>
   </autor>

</bibliothek>
```

In diesem Dokument werden einige bibliografische Angaben zu Büchern definiert. Enthalten sind die folgenden Werte:

- `Autor`
- `Titel`
- `ISBN`
- `Verlag`

Lässt man sich das Dokument auf herkömmliche Weise anzeigen, erscheinen die Bücher exakt in der Reihenfolge, in der sie im Dokument definiert wurden.

Listing 7.67 Das wäre die normale Variante.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
   <h1>Bibliothek</h1>
   <table border="1">
      <tr>
         <td>Autor</td>
       <td>Titel</td>
       <td>ISBN</td>
       <td>Verlag</td>
      </tr>
   <xsl:for-each select="bibliothek/autor">
      <tr>
         <td><xsl:value-of select="@name"/></td>
         <td><xsl:value-of select="buch/titel" /></td>
         <td><xsl:value-of select="buch/isbn" /></td>
         <td><xsl:value-of select="buch/verlag "/></td>
      </tr>
    </xsl:for-each>
    </table>
</xsl:template>

</xsl:stylesheet>
```

Das Ergebnis können Sie in **Abbildung 7.18** sehen.

Abbildung 7.18 Das ist die normale Ausgabe.

Da das zu unübersichtlich ist, soll die Ausgabe so geändert werden, dass eine alphabetische Sortierung nach Autorennamen vorgenommen wird. Zusätzlich werden die Angaben in eine HTML-Tabelle geschrieben. Theoretisch sollte die Tabelle dann folgendermaßen aussehen:

Tabelle 7.3: So sollte das Ergebnis aussehen.

Autor	Titel	ISBN	Verlag
Bryson	Eine kurze Geschichte von fast allem	3442460719	Goldmann
Parsons	One for my baby	3492261469	Piper
Roth	Verschwörung gegen Amerika	3499240874	Rowohlt

Für eine solche Ausgabe muss auf Schleifenkonstruktionen zurückgegriffen werden. Ausführliche Informationen dazu finden Sie im weiteren Verlauf dieses Kapitels.

Um nach dem Autorennamen zu sortieren, muss dem `select`-Attribut des `xsl:sort`-Elements das `name`-Element zugewiesen werden.

```
<xsl:sort select="@name"/>
```

Weitere Angaben – natürlich mit Ausnahme der Schleife – sind für das gewünschte Ergebnis nicht nötig.

Listing 7.68 Die Tabelle wird zusammengestellt.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
   <h1>Bibliothek</h1>
   <table border="1">
      <tr>
       <td>Autor</td>
      <td>Titel</td>
      <td>ISBN</td>
      <td>Verlag</td>
      </tr>
   <xsl:for-each select="bibliothek/autor">
   <xsl:sort select="@name"/>
      <tr>
         <td><xsl:value-of select="@name"/></td>
         <td><xsl:value-of select="buch/titel" /></td>
         <td><xsl:value-of select="buch/isbn" /></td>
         <td><xsl:value-of select="buch/verlag "/></td>
      </tr>
    </xsl:for-each>
    </table>
</xsl:template>

</xsl:stylesheet>
```

Die Hauptaufgabe übernimmt `xsl:for-each`. Hierüber werden die einzelnen Elemente des Dokuments mittels einer Schleife durchlaufen. Die Sortierung wird über `xsl:for-each` realisiert.

Lässt man sich das Ergebnis jetzt im Browser anzeigen, ergibt sich das gewünschte Bild.

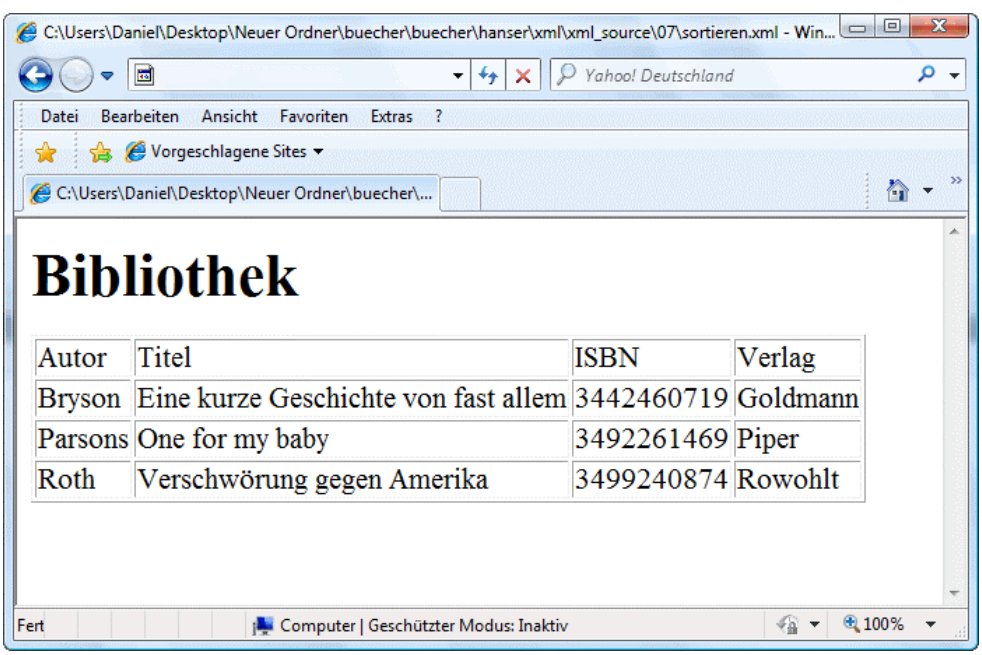

Abbildung 7.19 Das gewünschte Ergebnis wurde erzielt.

Dieses Beispiel zeigt einen großen Vorteil von XSLT gegenüber anderen Sprachen. Versucht man eine solche Sortierung und Tabellenausgabe nämlich in einer anderen Sprache, wird man dafür zumeist ein Vielfaches an Code benötigen. XSLT spielt hier seine Fähigkeit voll aus, dass man mit wenig Code eine Menge erreichen kann.

Einen zweiten Blick verdient noch einmal das Attribut `data-type`, über das der Datentyp des Sortierschlüssels angegeben wird. Wenn man

```
<xsl:sort select="zahl" datat-ype="number"/>
```

verwendet, wird der Sortierschlüssel dort, wo es möglich ist, in einen numerischen Wert konvertiert. Interessant ist diese Variante z.B., wenn nummerierte Inhalte ausgegeben werden sollen. So könnte in der gezeigten Anwendung beispielsweise noch eine zusätzliche Spalte vorhanden sein, in der der Verkaufsrang angegeben wird.

Tabelle 7.4: So soll das Ergebnis aussehen.

Rang	Autor	Titel	ISBN	Verlag
2	Bryson	Eine kurze Geschichte von fast allem	3442460719	Goldmann
3	Parsons	One for my baby	3492261469	Piper
1	Roth	Verschwörung gegen Amerika	3499240874	Rowohlt

Sinnvollerweise würde man eine solche Tabelle nach der `Rang`-Spalte sortieren. Dazu sind sowohl an der XML- wie auch an der XSLT-Datei einige Anpassungen nötig. Zunächst wieder das XML-Dokument:

Listing 7.69 Das ist das Ausgangsdokument.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="sortierung2.xsl" ?>

<bibliothek>

   <autor name="Parsons">
      <buch>
         <rang>2</rang>
         <titel>One for my baby</titel>
         <isbn>3492261469</isbn>
         <verlag>Piper</verlag>
      </buch>
   </autor>

   <autor name="Bryson">
      <buch>
            <rang>3</rang>
         <titel>Eine kurze Geschichte von fast allem</titel>
         <isbn>3442460719</isbn>
         <verlag>Goldmann</verlag>
      </buch>
   </autor>

   <autor name="Roth">
      <buch>
         <rang>1</rang>
         <titel>Verschwörung gegen Amerika</titel>
         <isbn>3499240874</isbn>
         <verlag>Rowohlt</verlag>
```

```
      </buch>
   </autor>

</bibliothek>
```

Hier wurde jeweils ein `rang`-Element eingefügt, dem ein entsprechender numerischer Wert zugewiesen wurde, der den Verkaufsrang widerspiegelt. Damit die Ausgabe wie gewünscht aussieht, muss auch das XSLT-Dokument angepasst werden.

Listing 7.70 Dieses Dokument sorgt für die richtige Ausgabe.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
   <h1>Bibliothek</h1>
   <table border="1">
      <tr>
         <td>Rang</td>
         <td>Autor</td>
         <td>Titel</td>
         <td>ISBN</td>
       <td>Verlag</td>
      </tr>
   <xsl:for-each select="bibliothek/autor">
   <xsl:sort select="buch/rang" data-type="number"/>
      <tr>
         <td><xsl:value-of select="buch/rang"/></td>
         <td><xsl:value-of select="@name"/></td>
         <td><xsl:value-of select="buch/titel" /></td>
         <td><xsl:value-of select="buch/isbn" /></td>
         <td><xsl:value-of select="buch/verlag "/></td>
      </tr>
    </xsl:for-each>
    </table>
</xsl:template>
</xsl:stylesheet>
```

Es hat sich an der Syntax nicht besonders viel geändert. Zunächst einmal wurde dafür gesorgt, dass das `rang`-Element auch tatsächlich abgefragt wird. Realisiert wird das über `<xsl:sort select="buch/rang" />`. Damit die Sortierung numerisch erfolgt, wurde `data-type="number"` verwendet. Der Parser sortiert die Elemente daraufhin nach der Nummer in der `Rang`-Spalte, und zwar aufsteigend. Würde man anstelle von `data-type="number"` den Wert `data-type="text"` angeben, fiele die Sortierung textbasiert aus.

Abbildung 7.20 Jetzt wird nach der Rang-Spalte sortiert.

Wollte man eine umgekehrte Reihenfolge erzwingen, dann sähe das folgendermaßen aus:

```
<xsl:sort select="buch/rang" data-type="number" order="descending"/>
```

Abbildung 7.21 zeigt die Auswirkungen des `order`-Attributs in Verbindung mit dem Wert `descending`.

Abbildung 7.21 Die Reihenfolge wurde umgekehrt.

7.9.6.2 Elemente gruppieren

Auch das Gruppieren von Elementen ist in XSLT möglich, stellt sich allerdings deutlich komplizierter als das Sortieren dar. Was es mit dem Gruppieren auf sich hat und was daran eigentlich so schwierig ist, lässt sich am besten anhand eines Beispiels zeigen.

Listing 7.71 Das ist das Ausgangsdokument.

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

   <autoren id="0001">
      <nname>Ellis</nname>
      <vname>Bret</vname>
   </autoren>

   <autoren id="0002">
      <nname>Hornby</nname>
      <vname>Ellis</vname>
   </autoren>

   <autoren id="0003">
      <nname>Ellis</nname>
      <vname>Michael</vname>
   </autoren>

</bibliothek>
```

Es handelt sich hierbei um ein einfaches XML-Dokument. Durch eine Gruppierung sollen die Datensätze nach dem Nachnamen gruppiert werden. Im Ergebnisdokument sollte das folgendermaßen aussehen:

Listing 7.72 So soll die Sortierung letztendlich aussehen.

```
Ellis,<br />
     Bret<br />
     Michael<br />

Hornby,<br />
   Nick<br />
```

In diesem Beispiel wurden die Vornamen den jeweiligen Nachnamen zugeordnet. Solche Anwendungen lassen sich in zwei Schritten umsetzen.

1. Man muss die Nachnamen ermitteln, die in der Rückgabe enthalten sind.
2. Sämtliche Einträge, die den gleichen Nachnamen haben, müssen ermittelt werden.

Für das Sortieren in XSLT bieten sich verschiedene Ansätze an. So gibt es z.B. eine Brute-Force-Variante. Dabei wird die Transformation in zwei Durchläufen ausgeführt, bei der man ein Zwischen-Stylesheet schreiben könnte, das die Namen sortiert und ein neues XML-Dokument erzeugt. Anschließend wird das Stylesheet benutzt, das man bereits geschrieben hat, da jetzt die Dokumentenreihenfolge und die sortierte Reihenfolge gleich sind. Das würde funktionieren, ist aber etwas umständlich. Das ist aber nicht der einzige Nachteil. Diese Variante ist auch noch extrem langsam, da in der Mitte angehalten, eine Datei auf die Festplatte geschrieben und anschließend die Datei erneut geschrieben werden muss.

Des Problems Lösung ist der `xsl:key`-Ansatz. Diese Variante wird oft als Muench-Methode, nach dem Oracle-XML-Experten Steve Muench, bezeichnet. Die Muench-Methode besteht aus drei Schritten.

1. Es wird zunächst ein Schlüssel für die Eigenschaft definiert, die zum Gruppieren genutzt werden soll.
2. Jetzt werden alle Knoten ausgewählt, die man gruppieren will. Dabei kann man z.B. auf die Funktionen `key()` und `generate-id()` zurückgreifen, um die eindeutigen Gruppierungswerte ermitteln zu können.
3. Um alle zutreffenden Knoten zu ermitteln, wird die `key()`-Funktion verwendet. Da diese Funktion eine Knotenmenge zurückliefert, könnte man mit der Knotenmenge, die auf einen bestimmten Gruppierungswert zutrifft, weitere Sortierungen durchführen.

Das sind also die allgemeinen Schritte aus theoretischer Sicht, jetzt geht es an die Praxis. Die Muench-Methode basiert auf dem Setzen eines Schlüssels. Ein solcher Schlüssel (Key) weist einem XML-Knoten einen Schlüsselwert zu. Auf Basis dieses Schlüssels wird der Zugriff auf den betreffenden Knoten möglich. Bei vielen Knoten mit dem gleichen Schlüsselwert werden die Knoten anhand dieses Werts ermittelt.

Im ersten Schritt muss man die Schlüsselfunktion anlegen. Im aktuellen Beispiel könnte diese folgendermaßen aussehen:

```
<xsl:key name="autor-name" match="autoren" use="nname" />
```

Hier definiert das `xsl:key`-Element einen neuen Index namens `autor-name`. `xsl:key` selbst kennt drei Attribute.

- `name` – Hierüber wird der Name für den Schlüssel festgelegt.
- `match` – Damit werden die Knoten ausgewählt, die den Schlüssel zugewiesen bekommen.
- `use` – Das ist der Wert des Schlüssels für jeden Knoten.

`xsl:key` kann innerhalb von `xsl:stylesheet` vorkommen, muss aber außerhalb von `xsl:template` stehen.

Nachdem der Schlüssel definiert wurde, lassen sich nun für einen Nachnamen alle Einträge ermitteln.

```
key('nname', 'ellis')
```

Diese Syntax liefert sämtliche Autoren, die den Nachnamen `ellis` haben. Damit ist Schritt 1 erledigt.

Im nächsten Schritt muss man alle Einträge ermitteln, die den gleichen Nachnamen haben.

```
<xsl:apply-templates select="key('autor-name', nname)" />
```

Zu diesem Zweck müssen alle verfügbaren Nachnamen, also der erste Eintrag zu einem bestimmten Nachnamen, ermittelt werden. Für diesen Zweck greift man ebenfalls auf den Einsatz von Schlüsselwerten zurück. Jeder Datensatz ist ein Teil der Knotenliste, die zurückgeliefert wird, wenn als Schlüsselwert der dazugehörende Nachname verwendet wird.

Will man herausfinden, ob es sich bei einem Eintrag um den ersten der durch einen Schlüssel zurückgegebenen Liste handelt, muss dieser Eintrag mit dem ersten Knoten der Liste verglichen werden. Dafür bieten sich verschiedene Möglichkeiten an. So kann man beispielsweise mit `generate-id()` einen eindeutigen Bezeichner für die beiden Knoten generieren lassen und diese miteinander vergleichen.

```
contact[generate-id() = generate-id(key('autor-name', nname)[1])]
```

Darüber hinaus gibt es auch noch eine andere Variante. Bei der wird überprüft, ob die Menge aus den zu vergleichenden Knoten exakt ein oder mehrere Elemente enthält. Da innerhalb einer Menge Knoten nicht mehrfach auftauchen dürfen, ist klar, dass, wenn ausschließlich ein Knoten enthalten ist, beide Knoten identisch sind.

```
contact[count(. | key('autor-name', nname)[1]) = 1]
```

Hier wurden die Gruppen ermittelt und lassen sich jetzt in die gewünschte Reihenfolge bringen. Für die gruppierte Ausgabe ist die folgende Syntax verantwortlich:

Listing 7.73 Die Elemente werden gruppiert.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:key name="autor-name" match="autoren" use="nname" />
<xsl:template match="bibliothek">
    <xsl:for-each select="autoren[count(. |
    key('autor-name', nname)[1]) = 1]">
      <xsl:sort select="nname" />
      <xsl:value-of select="nname" />,<br />
      <xsl:for-each select="key('autor-name', nname)">
         <xsl:sort select="vname" />
         <xsl:value-of select="vname" />
         <xsl:value-of select="title" />
          <br />
      </xsl:for-each>
   </xsl:for-each>
</xsl:template>

</xsl:stylesheet>
```

Noch ein Hinweis dazu, wo sich diese Syntax anwenden lässt: Sie funktioniert tatsächlich nur bei solchen XSLT-Prozessoren, die Keys unterstützen. Mittlerweile ist das aber bei den meisten Prozessoren der Fall.

7.9.7 Elemente und Attribute hinzufügen

Es gibt verschiedene Möglichkeiten, Elemente und Attribute zu erzeugen und diese im Ergebnisbaum ausgeben zu lassen. Interessant ist das in erster Linie, wenn in der Quelldatei bestimmte Informationen fehlen. XSLT stellt für diesen Zweck die folgenden Elemente zur Verfügung:

- `xsl:attribut`
- `xsl:element`
- `xsl:comment`

- `xsl:processing-instruction`
- `xsl:text`

Auf den folgenden Seiten lernen Sie den Umgang mit den einzelnen Elementen kennen.

7.9.7.1 Elemente erzeugen

Elemente dienen zunächst einer sauberen Syntax im Ergebnisbaum. Ebenso können sie aber auch Knoten erzeugen, die innerhalb des Ausgangsdokuments bereits schon mal verwendet wurden. Um ein Element zu erzeugen, wird `xsl:element` eingesetzt.

Angewendet wird `xsl:element` beispielsweise beim Einsatz verschiedener Attribut-Sets. Dann nämlich kann es sinnvoll sein, die Elemente im Ergebnisbaum nicht direkt zu notieren, sondern sie mittels `xsl:element` zu generieren.

Folgende Attribute sind verfügbar:

- `name` – Legt den Namen des zu generierenden Elements fest.
- `namespace` – Der Namensraum, dem das Element angehört.
- `use-attribute-sets` – Hierüber werden Attribut-Sets definiert, die innerhalb des einleitenden, zu erzeugenden Elements verwendet werden.

Durch die folgende Syntax

Listing 7.74 Das ist das Ausgangsdokument.

```
<xsl:element name="MeinElement">
   Mein Element
</xsl:element>
```

wird im Ergebnisbaum das folgende Element erzeugt:

Listing 7.75 So sieht das Element aus.

```
<MeinElement>
   Mein Element
</MeinElement>
```

7.9.7.2 Attribute hinzufügen

Ähnlich einfach wie Elemente lassen sich auch Attribute erzeugen. Interessant ist das zum Beispiel, wenn man einem HTML-Element ein zusätzliches Attribut zuweisen will. Denken Sie nur an einen `div`-Bereich, der rechtsbündig ausgerichtet werden soll. In diesem Fall könnte man das `align`-Attribut mit `xsl:attribute` dem `div`-Element zuweisen. `xsl:attribute` kennt die folgenden Attribute:

- `name` – Hierüber legt man den Namen des Attributs fest.
- `namespace` – Sollte für das Attribut der Namensraum bekannt sein, aus dem es stammt, kann man den entsprechenden URI hierüber angeben.

Wie sich eine solche Anwendung in der Praxis umsetzen lässt, zeigt das folgende Beispiel:

Listing 7.76 Die XML-Struktur wird angelegt.

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

   <text ausrichtung="links">
      Text steht links
   </text>

   <text ausrichtung="zentriert">
      Text ist zentriert
   </text>

   <text ausrichtung="rechts">
      Text steht rechts
   </text>

</dokument>
```

In diesem Dokument gibt es ein XML-Element `text`. Diesem Element wurde das Attribut `ausrichtung` zugewiesen. Das `ausrichtung`-Attribut kann einen der drei Werte `links`, `zentriert` oder `rechts` besitzen. Über das Stylesheet soll nun eine Ausgabe generiert werden, die für die „Übersetzung“ der Attribute in ihre HTML-Entsprechung sorgt.

Listing 7.77 Attribute werden zugewiesen.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
   <html>
      <head>
      </head>
      <body>
         <xsl:apply-templates />
      </body>
   </html>
</xsl:template>

<xsl:template match="text">
   <p>
   <xsl:choose>

   <xsl:when test="@ausrichtung='zentriert'">
      <xsl:attribute name="align">center</xsl:attribute>
   </xsl:when>

   <xsl:when test="@ausrichtung='rechts'">
      <xsl:attribute name="align">right</xsl:attribute>
     </xsl:when>

   <xsl:otherwise>
      <xsl:attribute name="align">left</xsl:attribute>
   </xsl:otherwise>

   </xsl:choose>

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

</xsl:stylesheet>
```

In diesem Beispiel werden Elemente vom Typ `text` in HTML-Elemente vom Typ `p` umgewandelt. Zusätzlich soll auch die Ausrichtung berücksichtigt werden. Dabei gilt die folgende Aufteilung:

- `links` = `left`
- `rechts` = `right`
- `zentriert` = `center`

Wird also eine für ein Element

```
ausrichtung="rechts"
```

ermittelt, soll daraus im Ergebnisdokument

```
<p align="right">
```

werden. Verwendet werden dafür `xsl:choose`, `xsl:when` und `xsl:otherwise`. Das Ergebnisdokument sieht folgendermaßen aus:

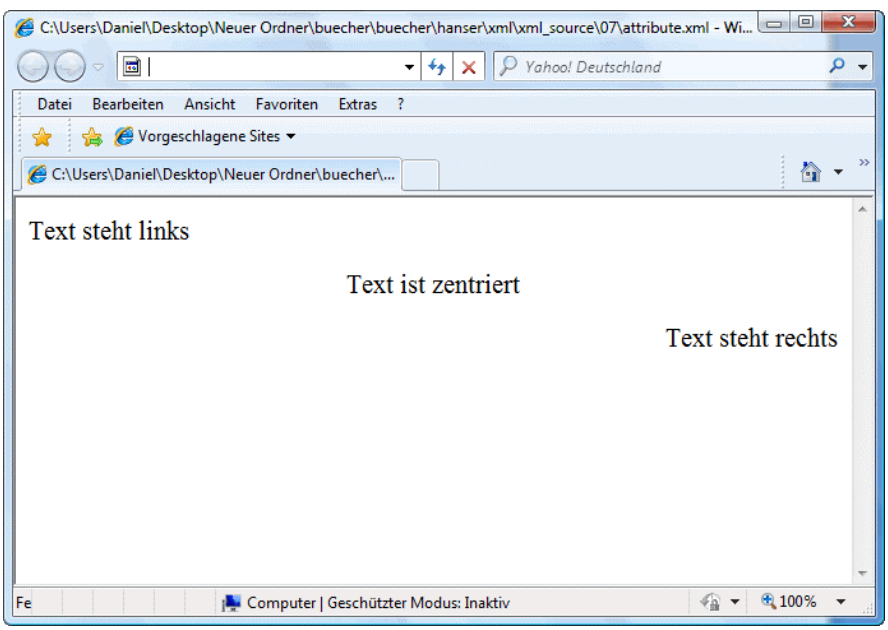

Abbildung 7.22 Die Elemente werden unterschiedlich ausgerichtet.

7.9.7.3 Attributlisten

Im vorherigen Abschnitt wurde gezeigt, wie Elementen Attribute zugewiesen werden können. Das funktioniert so lange, wie man nur ein Attribut einfügen will. Sollen es aber mehrere werden, bietet sich der Einsatz von `xsl:attribute-set` an. Hierüber kann man ein oder mehrere Attribute für den Ergebnisbaum definieren, und zwar separat. Interessant ist das beispielsweise, wenn man mehrere Attribute häufig verwenden will. So muss man die Attribute nur einmal definieren, kann sie dann aber dem gewünschten Element auf einmal zuweisen.

`xsl:attribute-set` kennt die folgenden Attribute:

- `name` – Weist dem Attribut-Set[3] einen Namen zu. Über diesen Namen wird das Attribut-Set anschließend verwendet.
- `use-attribute-sets` – Hierüber kann ein anderes Attribut-Set in das aktuelle Attribut-Set eingebunden werden. Um mehrere Attribut-Sets anzugeben, müssen die Namen durch Leerzeichen getrennt werden.

`xsl:attribute-set` kann innerhalb von `xsl:stylesheet` vorkommen, muss aber außerhalb von `xsl:template` stehen. Zudem kann es von `xsl:element` und `xsl:copy` über deren Attribut `use-attribute-sets` innerhalb eines Templates verwendet werden.

Wie sich `xsl:attribute-set` am besten einsetzen lässt, zeigt folgendes Beispiel. In diesem wird eine XML-Struktur in Tabellenform ausgegeben. Das Ausgangsdokument sieht folgendermaßen aus:

Listing 7.78 Das ist das XML-Dokument.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="attribute_set.xsl" ?>
<bibliothek>
   <autor>
      <vorname>Nick</vorname>
      <nachname>Hornby</nachname>
   </autor>

   <autor>
      <vorname>Tony</vorname>
      <nachname>Parsons</nachname>
   </autor>
</bibliothek>
```

Die Tabelle wird über verschiedene Attribute formatiert. In diesem Beispiel sind dies `border`, `cellpadding` und `cellspacing`. Um diese Attribute der Tabelle zuzuweisen, bietet sich der Einsatz von `xsl:attribute-set` an.

Listing 7.79 Die Attribut-Sets werden definiert.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:attribute-set name="tabellen_definition">

   <xsl:attribute name="border">
      1
   </xsl:attribute>

   <xsl:attribute name="cellpadding">
      3
   </xsl:attribute>

   <xsl:attribute name="cellspacing">
      5
   </xsl:attribute>

</xsl:attribute-set>
```

[3] Attribut-Set mit – in diesem Zusammenhang – einem Satz an Attributen.

```
<xsl:template match="/">
   <html>
      <head>
      </head>
   <body>
   <xsl:element name="table" use-attribute-sets="tabellen_definition">
      <tr
         <td><b>Vorname</b></td>
         <td><b>Nachname</b></td>
      </tr>
      <xsl:for-each select="bibliothek/autor">
      <tr>
         <td valign="top"><xsl:value-of select="vorname" /></td>
         <td valign="top"><xsl:value-of select="nachname" /></td>
      </tr>
    </xsl:for-each>
</xsl:element>
 </body>
 </html>
 </xsl:template>

</xsl:stylesheet>
```

Noch vor der eigentlichen Template-Definition wird das Attribut-Set mit dem Namen `tabellen_definition` definiert. Innerhalb des `xsl:attribute-set name`-Elements werden über `xsl:attribute` die jeweiligen Attribute bestimmt. Um das Attribut-Set später auf das `table`-Element anwenden zu können, wird die Anweisung `<xsl:element name="table" use-attribute-sets="tabellen_definition">` verwendet. Hier wird über `name` das entsprechende Element (`table`) und über `use-attribute-sets` das zu verwendende Attribut-Set angegeben.

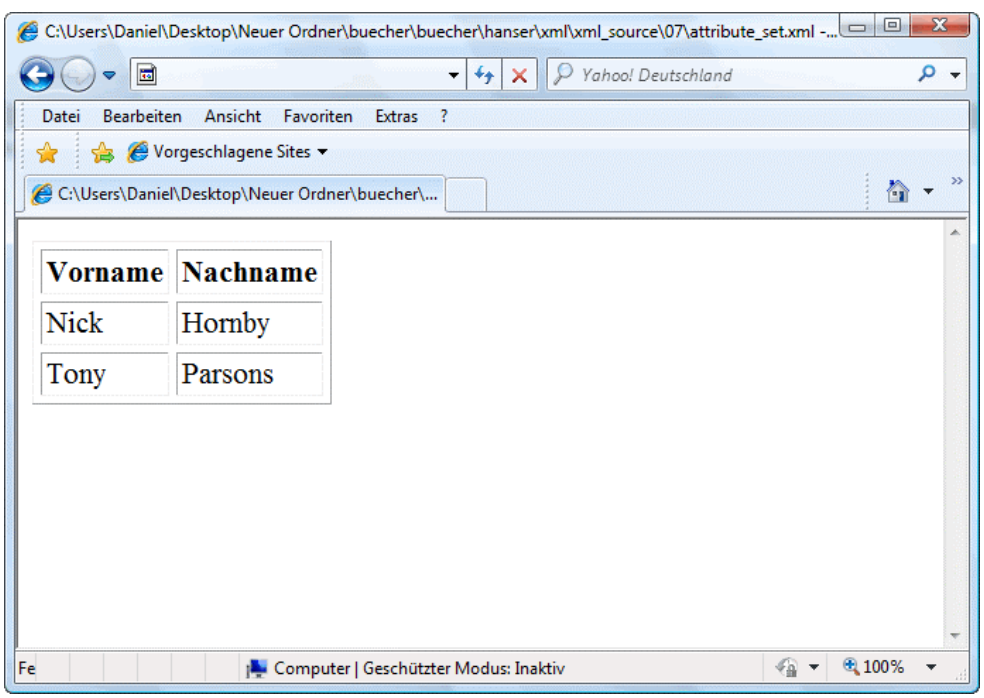

Abbildung 7.23 So sieht das Ergebnis aus.

7.9.7.4 Zeicheninhalt ausgeben

Um einfachen statischen Text im Ergebnisdokument auszugeben, wird `xsl:text` verwendet. Auf diese Weise lassen sich über das XSLT-Stylesheet Texte definieren, die im Ergebnisdokument ausgegeben werden. `xsl:text` darf innerhalb von `xsl:template` vorkommen.

Listing 7.80 Das ist das Ausgangsdokument.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="xsltext.xsl" ?>
<gruss>
   <welt>Hallo, Welt!</welt>
</gruss>
```

Innerhalb des Ausgangsdokuments wurde ein einfacher Willkommensgruß definiert. Über das Stylesheet soll dieser Gruß nun in eine entsprechende HTML-Syntax überführt werden. Das Besondere an diesem Beispiel ist nun allerdings, dass unterhalb des Grußes ein zusätzlicher Text eingefügt wird.

Listing 7.81 Hier wird mit JavaScript gearbeitet.

```
<?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>
      </head>
   <body>
      <xsl:apply-templates />
   </body>
   </html>
</xsl:template>

<xsl:template match="welt">
   <p>
      <xsl:value-of select="." />
   </p>

   <script type="text/javascript">
      <xsl:text>
         &lt;!--
         document.write(&quot;&lt;p&gt;&quot; +
         document.lastModified + &quot;&lt;/p&gt;&quot;);
         //--&gt;
      </xsl:text>
   </script>
</xsl:template>

</xsl:stylesheet>
```

Um den zusätzlichen Text auszugeben, wird mit `xsl:text` gearbeitet. Bei diesem Text handelt es sich um ein JavaScript, über das der Zeitpunkt der letzten Änderung des Dokuments angezeigt wird. Das Beispiel zeigt so ganz nebenbei übrigens noch etwas, nämlich wie HTML-eigene Zeichen maskiert werden.[4]

[4] So werden im vorliegenden Beispiel über &quot Anführungszeichen definiert.

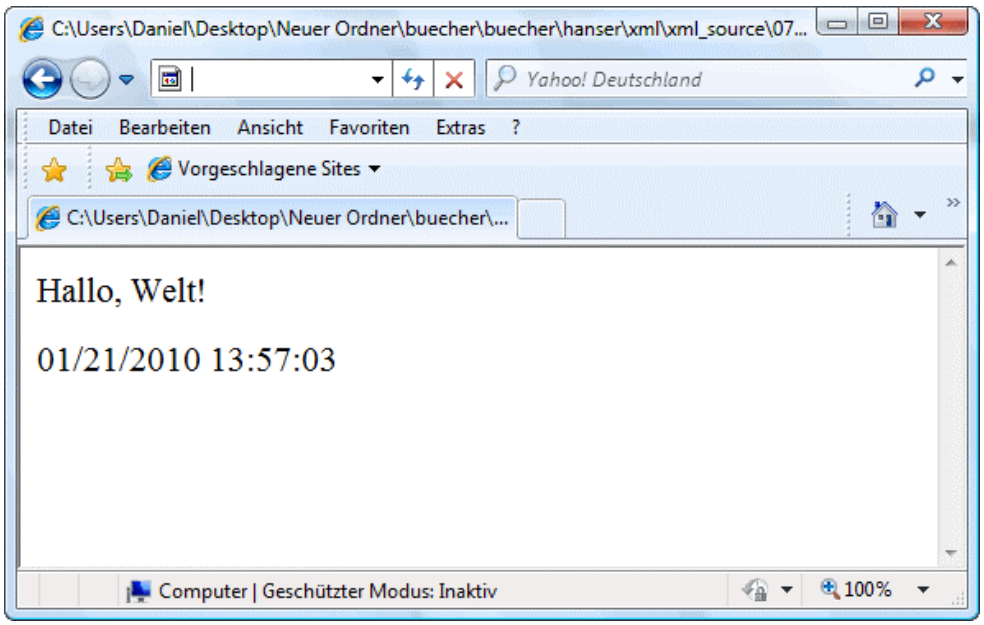

Abbildung 7.24 Auch das Änderungsdatum des Dokuments wird ausgegeben.

7.9.7.5 Kommentare einfügen

Oftmals ist es sinnvoll, im Ergebnisbaum Kommentare einzufügen. Damit diese XML- und/oder HTML-gerecht sind, gibt es `xsl:comment`. Durch dieses Element wird der eigentliche Inhalt nicht sichtbar. Attribute besitzt das Element nicht. Vorkommen darf es innerhalb von `xsl:template`.

Listing 7.82 Kommentare werden eingefügt.

```
<xsl:template match="/">
   <html>
      <head>
         <script type="text/javascript">
            <xsl:comment>
               alert("Hallo, Welt!");
            </xsl:comment>
         </script>
      </head>
   <body>
      Hallo Welt
   </body>
   </html>
</xsl:template>
```

In dieser Syntax wird bei der HTML-Ausgabe unter anderem ein JavaScript-Bereich notiert. Der Inhalt dieses Bereichs soll nun auskommentiert werden. Auf diese Weise wird verhindert, dass solche Browser, die JavaScript nicht interpretieren können, keine Fehlermeldung ausgeben.

7.9.7.6 Leerräume

XML behandelt von Hause aus bereits Leerräume, also den sogenannten Whitespace. Wem diese Whitespace-Behandlung nicht genügt, der kann auf entsprechende XSLT-Elemente zurückgreifen. Denn tatsächlich stellt XSLT zwei Elemente bereit, über die man explizit die Behandlung von Leerräumen durch den Parser bestimmen kann. Die beiden Elemente sind

- `xsl:strip-space` und
- `xsl:preserve-space`.

Über beide Elemente wird festgelegt, was mit den Leerräumen, die innerhalb des Quelldokuments enthalten sind, bei der Ausgabe geschehen soll. Hier zunächst die verfügbaren Leerraumzeichen:

- `#x20` – einfaches Leerzeichen
- `#x9` – Tabulator-Zeichen
- `#xD` – Wagenrücklaufzeichen
- `#xA` – Zeilenvorschub-Zeichen

Um Missverständnissen vorzubeugen: Es geht an dieser Stelle um die Formatierung der Elemente im Ergebnisbaum. Die Leerräume, die innerhalb der Elemente vorkommen, bleiben davon unberührt.

Ein Beispiel soll den Einsatz der Leerzeichen-Behandlung zeigen. Zunächst das Quelldokument.

Listing 7.83 Achten Sie auf die Leerzeile.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="ausgabe.xsl" ?>
<autor>
   <name>Parsons</name>

   <name>Ellis</name>
</autor>
```

Zwischen den beiden `name`-Elementen wurde eine Leerzeile eingefügt, die so auch im Ergebnisdokument erhalten bleiben soll. Dafür wird das Element `xsl:preserve-space` verwendet.

Listing 7.84 Die Leerzeile bleibt erhalten.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:preserve-space elements="autor" />

<xsl:template match="/">
   <html>
      <head>
      </head>
   <body>
      <pre>
         <xsl:apply-templates />
       </pre>
   </body>
   </html>
</xsl:template>
```

```
<xsl:template match="name">
   <b>
      <xsl:value-of select="." />
   </b>
</xsl:template>
</xsl:stylesheet>
```

Lässt man sich das Ergebnisdokument anzeigen, wird deutlich, dass die Leerzeile tatsächlich enthalten ist. Beachten Sie, dass das momentan weder im Internet Explorer noch im Firefox funktioniert.

```
Parsons


Ellis
```

Würde man das Stylesheet ohne `xsl:preserve-space` auf das Dokument anwenden, dann ergäbe sich folgendes Bild:

```
Parsons
Ellis
```

Die Leerzeile würde automatisch entfernt werden. Den beiden Elementen `xsl:preserve-space` und `xsl:strip-space` wird eine Liste von durch Leerzeichen getrennten Elementnamen übergeben. Wird `xsl:strip-space` verwendet, werden die Leerzeichen, die zwischen den Elementen innerhalb des Quelldokuments enthalten sind, entfernt.

Will man die Leerzeichen hingegen erhalten, muss man `xsl:preserve-space` einsetzen. Da es sich dabei allerdings ohnehin um das Standardverhalten handelt, kann man normalerweise auf die Angabe von `xsl:preserve-space` verzichten.

Anstelle der Elementnamen können Sie übrigens auch das Wildcard-Zeichen (`*`) verwenden. Dadurch werden alle Elemente mit der angegebenen Eigenschaft behandelt.

7.9.8 Mit Funktionen arbeiten

Innerhalb von XPath-Ausdrücken, die man in XSLT an den verschiedenen Stellen verwenden kann, lassen sich die bereits vorgestellten XPath-Funktionen nutzen. Darüber hinaus stellt XSLT aber auch noch zusätzliche Funktionen zur Verfügung, die auf den folgenden Seiten vorgestellt werden.

Zunächst ein einführendes Beispiel. Die folgende Syntax überprüft, ob die Funktion `normalize-space` im aktuell verwendeten XSLT-Prozessor verfügbar ist.

Listing 7.85 Ein erstes Beispiel für den Einsatz einer Funktion

```
<xsl:choose>
  <xsl:when test="function-available('normalize-space')">
      <xsl:value-of select="normalize-space(.)" />
   </xsl:when>

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

</xsl:choose>
```

Verwendet wird für diesen Zweck die XSLT-Funktion `function-available()`. Als Wert wird ihr der Name der gesuchten Funktion übergeben.

Die folgende Übersicht zeigt die verfügbaren XSLT-Funktionen:

- `current()` – Stellt den Bezug zum aktuellen Knoten her. Als Ergebnis wird das Knotenset geliefert, das den aktuellen Knoten enthält.
- `document(String)` – Ermöglicht es, XML-Ausgangsdaten aus anderen XML-Dateien, als der in das Stylesheet eingebundenen, in den Ergebnisbaum zu übernehmen.
- `element-available(String)` – Ermittelt, ob ein XSLT-Element im verwendeten Parser verfügbar ist.
- `format-number(Zahl, String)` – Konvertiert eine Zahl in einen String unter Verwendung des im zweiten Argument angegebenen Musters. Ein Beispiel und die in der Funktion verfügbaren Formatregeln finden Sie im Anschluss an diese Übersicht.
- `function-available(String)` – Ist dem Prozessor die Funktion mit dem angegebenen Namen bekannt, wird `true` zurückgeliefert.
- `generate-id(String)` – Generiert einen String, der als Wert eines ID-Attributs verwendet wird.
- `key(String, String)` – Alle Knoten, die einen Schlüssel mit dem ersten Argument als Namen und dem zweiten Argument als Wert besitzen, werden als Ergebnis-Knotenmenge erstellt.
- `system-property()` – Ermittelt Informationen über den verwendeten XSLT-Prozessor. XSLT-Prozessoren müssen wenigstens die drei Eigenschaften `xsl:version`, `xsl:vendor` und `xsl:vendor-url` kennen.
- `unparsed-entity-uri(String)` – Ermöglicht den Zugriff auf die DTD-Einträge, die vom Parser analysiert wurden. Gibt es keinen entsprechenden Eintrag, wird ein Leer-String generiert.

Eine genauere Betrachtung verdient die Funktion `format-number()`. Über diese Funktion kann man explizit die Art und Weise angeben, in der Zahlen formatiert werden sollen.

Listing 7.86 Die gewünschte Formatierung wird festgelegt.

```
<xsl:template match="/">
<html>
   <body>
      <xsl:value-of select='format-number(500100, "#.00")' />
      <br />

      <xsl:value-of select='format-number(500100, "#.0")' />
      <br />

      <xsl:value-of select='format-number(500100, "###,###.00")' />
      <br />

      <xsl:value-of select='format-number(0.23456, "##%")' />
      <br />

      <xsl:value-of select='format-number(500100, "#######")' />

   </body>
</html>

</xsl:template>
```

Die Ausgabe im Ergebnisdokument würde folgendermaßen aussehen:

```
500100.00
500100.0
500,100.00
23%
500100
```

Über die Funktion `format-number()` wird also eine Zahl `x` in einen String konvertiert. Die Art der Konvertierung wird dabei über das zweite Argument bestimmt.

```
<xsl:value-of select='format-number(500100, "#.0")' />
```

Als Argument können die folgenden Werte angegeben werden:

- `0` – eine Ziffer
- `#` – eine Ziffer, bei der allerdings führende oder abschließende Nullen nicht angezeigt werden.
- `.` – Platzhalter für Dezimalpunkt-Trennzeichen
- `,` – Platzhalter für Gruppierungs-Trennzeichen
- `;` – mehrere Formate trennen
- `-` – negatives Vorzeichen als Voreinstellung
- `%` – Wert mit 100 multiplizieren und als Prozentwert anzeigen
- `‰` – Wert mit 1000 multiplizieren und als Promillewert anzeigen

Das dritte Argument ist ein optionaler Name des Dezimalformats. Das ermöglicht den Einsatz eines beliebigen Zeichens innerhalb der Formatierungsmuster-Zeichenfolge. Dem Zeichen wird eine Rolle in `xsl:decimal-format` zugewiesen. Ein typisches Beispiel dafür ist `european`. Hierüber wird die Rolle des Kommas und des Punkts aus dem Standardwert umgekehrt.

7.9.8.1 Auf mehrere Quelldokumente zugreifen

Eine weitere interessante Funktion ist `document()`. Mit dieser Funktion kann man mittels XSLT auf mehrere Dokumente gleichzeitig zugreifen. Auch dieser Aspekt lässt sich wieder am besten anhand eines Beispiel zeigen.

Angenommen, es existieren verschiedene Dokumente, in denen jeweils ein Name und eine E-Mail-Adresse enthalten sind. Ein typisches Dokument würde folgendermaßen aussehen:

Listing 7.87 Eines von mehreren Dokumenten

```
<eintrag>
   <name>Michael Mayer</name>
   <addresse>mayer@hanser.de</addresse>
</eintrag>
```

Es wird davon ausgegangen, dass es neben dem gezeigten Dokument noch zwei weitere Dateien gibt. Um alle drei Dateien in einem Dokument zusammenzuführen, muss jeweils eine Referenz auf sie erstellt werden.

Listing 7.88 Die einzelnen Dokumente werden gesammelt.

```
<dateien>
   <datei name="mail1.xml" />
   <datei name="mail2.xml" />
   <datei name="mail3.xml" />
</dateien>
```

Was jetzt noch fehlt, ist ein Stylesheet, das die zuvor gezeigte Datei einliest und die in dieser Datei definierten Dateien einbindet.

Listing 7.89 Die Dokumente werden ausgewertet.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="text"/>

<xsl:template match="/">
   <xsl:for-each select="/dateien/datei">
      <xsl:apply-templates select="document(@name)/eintrag" />
   </xsl:for-each>
</xsl:template>

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

<xsl:template match="name">
   Name: <xsl:value-of select="." />
</xsl:template>

<xsl:template match="addresse">
   E-Mail-Adresse: <xsl:value-of select="." />
</xsl:template>

</xsl:stylesheet>
```

Das Stylesheet wertet das Hauptdokument aus, in dem es alle Elemente mittels einer `for-each`-Schleife abarbeitet. Dabei wird jedes Mal mittels der `document()`-Funktion in die einzelnen Dokumente geblickt. Das erste Argument der Funktion liefert den Dateinamen, das zweite eine Knotenliste innerhalb dieser Datei.

Listing 7.90 Das Ergebnis sieht dann folgendermaßen aus:

```
Name: Michael Mayer
E-Mail-Adresse: mayer@hanser.de


Name: Jonny Müller
E-Mail-Adresse: mueller@hanser.de


Name: Marina Gonzalez
E-Mail-Adresse: gonzalez@hanser.de
```

Dieses Beispiel hat gezeigt, wie sich problemlos mehrere Dateien zu einer zusammenfügen lassen.

7.9.8.2 Eigene Funktionen definieren

In XSLT 1.0 gab es bereits die Möglichkeit, den Sprachumfang um eigene benutzerspezifische Funktionen zu erweitern. Das Problem dabei bestand allerdings darin, dass der Programmcode zusammen mit dem Stylesheet ausgeliefert werden musste, das den entsprechenden Code verwendet hat. Da man viele Dinge allerdings auch mit Bordmitteln von XPath und XSLT lösen kann, wurde in XSLT 2.0 der neue Elementtyp `function` eingeführt. Über den lassen sich nutzerspezifische Funktionen definieren. `xsl:function` deklariert den Namen, die Parameter und die Implementierung der Funktion.

Eine auf diese Weise definierte Funktion kann in XPath unter ihrem vollqualifizierten Namen verwendet werden, der über das `name`-Attribut spezifiziert wurde. Mögliche Parameter werden über `xsl:param` eingebunden. Zusätzlich wird `xsl:sequence` eingesetzt, um das Funktionsergebnis zu spezifizieren.

Listing 7.91 Eine Funktion wurde definiert.

```
<xsl:function name="a:nextNumber">
     <xsl:param name="number" as="xs:integer"/>
     <xsl:choose>
             <xsl:when test="$number = 1">
                     <xsl:sequence select="string('Eins')"/>
             </xsl:when>
             <xsl:when test="$number = 2">
                     <xsl:sequence select="string('Zwei')"/>
             </xsl:when>
                  <xsl:when test="$number = 3">
                    <xsl:sequence select="string('Drei')"/>
             </xsl:when>
     </xsl:choose>
</xsl:function>
```

Solche Funktionen sind innerhalb des definierten Stylesheets sichtbar.

```
<xsl:value-of select=" a:nextNumber( 1 )"/>
```

Zusätzlich stehen sie in allen Stylesheets zur Verfügung, von denen das Stylesheet importiert wird. Befindet sich innerhalb eines Stylesheets eine Funktionsdefinition, die bereits in einem importierten Stylesheet definiert wurde, überschreibt die neue Definition die importierte Variante. In diesem Zusammenhang wird von gleichen Funktionen gesprochen, wenn die Funktionsnamen und die Anzahl der Parameter übereinstimmen. Keine Rolle spielen hingegen der Ergebnistyp und die Typen der Parameter.

Um eine Funktion nur dann sichtbar zu machen, wenn noch keine Funktion mit dem gleichen Namen und der gleichen Parameteranzahl existiert, kann dem einleitenden `xsl:function`-Element das Attribut `override` zugewiesen werden. Folgende Werte sind möglich:

- `yes` – Überschreiben der Funktion ist erwünscht.
- `no` – Überschreiben der Funktion ist nicht erwünscht.

Auf diese Weise kann man die genannten Import-Probleme umgehen.

7.9.9 HTML- und XML-Ausgaben

Die Art der Ausgabe können Sie üblicherweise dem Prozessor überlassen. Was aber, wenn er PDF generiert, Sie aber ein HTML-Dokument benötigen? Dann kommt `xsl:output` zum Einsatz. Dieses Element kann dazu verwendet werden festzulegen, wie der bei der Abarbeitung des Stylesheets generierte Ergebnisbaum in das Ergebnisdokument übertragen wird.

`xsl:output` kennt verschiedene Parameter, über die sich die Ausgabe steuern lässt. Das Element kann innerhalb von `xsl:stylesheet` vorkommen, muss aber außerhalb von `xsl:template` stehen.

Zunächst ein Blick auf die Attribute, die bei `xsl:output` möglich sind:

- `method` – Legt die Art fest, in der der Ergebnisbaum erzeugt werden soll. Mögliche Werte sind `xml`, `html` (Standard), `text` oder öffentliche bzw. eigene Namensräume.
- `cdata-section-elements` – Gibt bei `method="xml"` die Elemente an, deren Inhalte beim Anlegen des Ergebnisbaums in CDATA-Abschnitte geschrieben werden sollen.
- `encoding` – Hierüber wird die Kodierung bestimmt, in der der Ergebnisbaum kodiert wird. Standardmäßig wird UTF-8 angenommen, Sie können aber z.B. auch ISO-8859-1 festlegen.
- `indent` – Verwendet man den Wert `yes`, werden im Ergebnisbaum untergeordnete Elemente eingerückt dargestellt. Diese Angabe ist von Vorteil, wenn man den Quelltext besser lesbar gestalten möchte. Auf die Darstellung des Dokuments hat diese Angabe keinerlei Auswirkungen.
- `doctype-public` – Sollte sich die Gültigkeitsprüfung des Ergebnisbaums auf eine öffentliche DTD beziehen, wird diesem Attribut bei `method="xml"` und `method="html"` die Zeichenkette des öffentlichen Bezeichners zugewiesen.
- `doctype-system` – Wenn sich die Gültigkeitsprüfung des Ergebnisbaums auf eine adressierte DTD bezieht, wird diesem Attribut bei `method="xml"` und `method="html"` die Zeichenkette der System-ID zugewiesen.
- `media-type` – Hierüber gibt man den MIME-Typ des Ergebnisbaums an. Bei `method="html"` ist der beispielsweise `text/html`, bei `method="xml"` ist er üblicherweise `text/xml`, und bei `method="text"` lautet er meistens `text/plain`.
- `omit-xml-declaration` – Wurde als Ausgabemethode `method="xml"` angegeben, legt man hierüber fest, ob im Ergebnisbaum eine XML-Deklaration (`<?xml ...?>`) ausgegeben werden soll. Mögliche Werte sind `yes` (ohne XML-Deklaration) und `no` (mit XML-Deklaration). Das ist übrigens kein Schreibfehler, denn `omit` steht hier für weglassen. `yes` bedeutet somit also, dass die XML-Deklaration weggelassen werden soll.
- `version` – Wurde `omit-xml-declaration="no"` gesetzt, legt man hierüber die XML-Versionsangabe fest.

- `standalone` – Hat man `omit-xml-declaration="no"` angegeben, gibt man bekannt, ob sich die DTD-Deklarationen innerhalb der aktuellen Datei befinden. Letztendlich wird also festgelegt, ob in der XML-Deklaration `standalone="yes"` oder `standalone="no"` stehen soll.

Die Funktionsweise von `xsl:output` lässt sich am besten anhand eines Beispiels zeigen.

Listing 7.92 Das ist das Ausgangsdokument.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="xml_ausgabe.xsl" ?>
<autoren>
   <buch>
      <autor>
         Bill Bryson
      </autor>
   </buch>
</autoren>
```

In diesem XML-Dokument sind die drei Elemente `autoren`, `buch` und `autor` enthalten. Es handelt sich dabei offensichtlich um deutschsprachige Ausdrücke. Um das Dokument internationaler zu gestalten, sollen diese Elemente ins Englische übersetzt werden. Nun könnte man die Übersetzung natürlich direkt in der XML-Datei vornehmen. Besser ist es jedoch, wenn die Übersetzung über das XSLT-Stylesheet gesteuert wird. Und genau für solche Zwecke ist `xsl:output` ideal.

Listing 7.93 Die entsprechende XSLT-Syntax sieht folgendermaßen aus:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:output method="xml" indent="yes" encoding="ISO-8859-1" omit-xml-
declaration="yes" />

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

<xsl:template match="buch">
   <book>
      <xsl:apply-templates />
   </book>
</xsl:template>

<xsl:template match="autor">
   <author>
      <xsl:value-of select="." />
   </author>
</xsl:template>

</xsl:stylesheet>
```

In diesem Beispiel werden `xsl:output` die beiden Attribute `method` und `indent` zugewiesen. Als Werte erhalten diese `xml` bzw. `yes`. `method="xml"` sorgt dafür, dass es sich bei dem Ergebnisdokument um XML handelt. Durch `indent="yes"` werden ins Ergebnisdokument Einrückungen eingefügt.

Ein abschließender Blick in das Ergebnisdokument liefert das gewünschte Ergebnis.

Listing 7.94 Das Dokument wurde übersetzt.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="ausgabe.xsl" ?>
<authors>
   <book>
      <author>
         Bill Bryson
      </author>
   </book>
</authors>
```

Die Elemente sind jetzt auf Englisch, und es wurden sogar Einrückungen berücksichtigt.

7.9.9.1 Die XML-Ausgabe

Ein erstes Beispiel zur XML-Ausgabe wurde im vorherigen Abschnitt gezeigt. Jetzt geht es noch etwas mehr ins Detail. Die XML-Ausgabemethode gibt den Ergebnisbaum als extern und allgemein analysierte Entity in wohlgeformtem XML aus. Sollte der Wurzelknoten des Ergebnisbaums ein einzelnes Elementknoten-Kind sein und keine Textknoten-Kinder besitzen, sollte die Entity ebenfalls eine wohlgeformte XML-Dokument-Entity sein.

Entscheidend sind für die XML-Ausgabe die folgenden Attribute:

- `version`
- `encoding`
- `indent`
- `cdata-section-elements`
- `media-type`

Eine typische XML-`output`-Definition könnte demnach also folgendermaßen aussehen:

```
<xsl:output method="xml" indent="yes" encoding="ISO-8859-1"
omit-xml-declaration="yes" />
```

De XML-Ausgabemethode sollte eine XML-Deklaration generieren. Das gilt jedoch nicht, wenn das Attribut `omit-xml-declaration` den Wert `yes` besitzt. Die XML-Deklaration sollte sowohl Angaben zur XML-Version wie auch zur Kodierung enthalten.

Wurde das `standalone`-Attribut angegeben, sollte es eine eigenständige Dokumenttypdeklaration mit dem gleichen Wert wie der Wert des `standalone`-Attributs einfügen. Im anderen Fall sollte es keine eigene Dokumenttypdeklaration einfügen, um so sicherzustellen, dass es sowohl eine XML-Deklaration als auch eine Text-Deklaration ist.

Wenn das `doctype-system`-Attribut angegeben wurde, sollte die XML-Ausgabemethode eine Dokumenttypdeklaration unmittelbar vor dem ersten Element ausgeben. Bei dem Namen, der auf `<!DOCTYPE` folgt, sollte es sich um den Namen des ersten Elements handeln.

Wurde `doctype-public` angegeben, sollte die XML-Ausgabemethode `PUBLIC` ausgeben werden, gefolgt von einem Identifikator und dem System-Identifikator. Im anderen Fall sollte `SYSTEM`, gefolgt vom System-Identifikator, ausgegeben werden.

7.9.9.2 Die HTML-Ausgabe

Neben der XML-Ausgabe gibt es auch noch die HTML-Variante. Bei der wird der Ergebnisbaum als HTML ausgegeben.

Listing 7.95 Die HTML-Ausgabe wird definiert.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:output method="html"/>

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

...

</xsl:stylesheet>
```

Mit `method="html"` wird die HTML-Ausgabemethode definiert. Auch bei dieser Variante sind, ähnlich wie bei XML, verschiedene Attribute verfügbar.

Über das `version`-Attribut wird die verwendete HTML-Version angegeben. Standardwert ist hier `4.0`.

Wurde dem `indent`-Attribut der Wert `yes` zugewiesen, darf die HTML-Ausgabemethode bei der Ausgabe des Ergebnisbaums Leerraum löschen und hinzufügen. Allerdings nur so lange, wie sich das nicht auf die Darstellung im Endgerät (Browser) auswirkt.

Ein Problem bei der HTML-Ausgabemethode sind die Inhalte der `script`- und der `style`-Elemente. Diese sollten nämlich nicht geschützt werden. Ein literales Ergebnisdokument in einem Stylesheet wie

```
<script>if (a &lt; b) foo()</script>
```

oder

```
<script><![CDATA[if (a < b) foo()]]></script>
```

sollte als

```
<script>if (a < b) foo()</script>
```

ausgegeben werden. Die HTML-Ausgabemethode sollte `<`-Zeichen, die in Attributwerten vorkommen, nicht schützen.

Boolesche Attribute – also solche Attribute, die nur einen einzigen erlaubten Wert haben, der mit dem Namen des Attributs übereinstimmt – sollten in minimierter Form ausgegeben werden. So sollte aus

```
<option selected="selected">
```

Folgendes gemacht werden:

```
<option selected>
```

Über das `encoding`-Attribut wird die bevorzugte Kodierung angegeben. Ist ein `head`-Element vorhanden, sollte die HTML-Ausgabemethode ein `meta`-Element direkt hinter dem Start-Tag des `head`-Elements hinzufügen, in dem die Zeichenkodierung ausgegeben wird.

Wurden die `doctype-public`- oder `doctype-system`-Attribute angegeben, sollte die HTML-Ausgabemethode eine Dokumenttypdeklaration unmittelbar vor dem ersten Element ausgeben. Der Name, der auf `<!DOCTYPE` folgt, sollte `html` oder `HTML` sein. Falls das `doctype-public`-Attribut notiert wurde, sollte die `PUBLIC`-Ausgabemethode, gefolgt von dem angegebenen öffentlichen Bezeichner, ausgegeben werden. Wurde zusätzlich das `doctype-system`-Attribut angegeben, sollten auch die angegebenen Systembezeichner hinter dem öffentlichen Bezeichner ausgegeben werden.

7.10 Text formatiert ausgeben

Bislang wurden die Texte im Ergebnisdokument unformatiert ausgegeben. Das mag für den Anfang genügen, irgendwann soll sich das aber natürlich ändern. Auch für solche Zwecke hält XSLT die passenden Elemente parat. Genau genommen handelt es sich dabei um eine Kombination aus XSLT und CSS.

Wer sich etwas mit CSS auskennt, kann so ganz einfach seine Dokumente formatieren. Wie sich solche Anwendungen umsetzen lassen, kann man gut anhand eines Beispiels zeigen. Das Ergebnis wird die (leicht) formatierte Ausgabe eines XML-Dokuments sein.

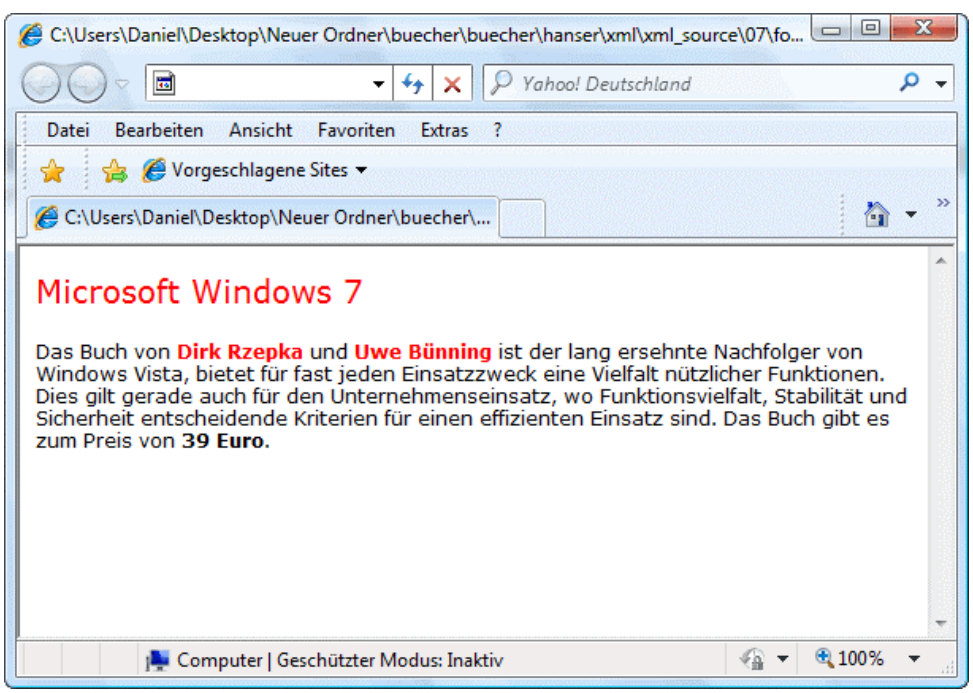

Abbildung 7.25 Das Dokument wurde formatiert.

Das XML-Dokument sieht folgendermaßen aus:

Listing 7.96 Das ist die Buchbeschreibung.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="formatierung.xsl" ?>
<buch>Microsoft Windows 7
     <beschreibung>Das Buch von <autor>Dirk Rzepka</autor> und
    <autor>Uwe Bünning</autor> ist der lang ersehnte Nachfolger
    von Windows Vista, bietet für fast jeden Einsatzzweck
    eine Vielfalt nützlicher Funktionen. Dies gilt gerade auch
    für den Unternehmenseinsatz, wo Funktionsvielfalt, Stabilität
    und Sicherheit entscheidende Kriterien für einen
    effizienten Einsatz sind. Das Buch gibt es zum Preis von
    <preis>39 Euro</preis>.
    </beschreibung>
</buch>
```

Das Dokument besteht aus nur wenigen Elementen. Für die drei Elemente `buch`, `beschreibung` und `preis` sollen mittels CSS verschiedene Formatierungen umgesetzt werden. Die entsprechende XSLT-Datei sieht folgendermaßen aus:

Listing 7.97 Auf diese Weise wird die formatierte Ausgabe erreicht.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
   <html>
     <head>
     </head>
   <body style="font-family:Verdana; font-size:20px; color:red">
      <xsl:apply-templates />
   </body>
   </html>
</xsl:template>

<xsl:template match="beschreibung">
   <p style="font-family:Verdana; font-size:12px; color:black">
      <xsl:apply-templates />
   </p>
</xsl:template>

<xsl:template match="autor">
   <span style="font-weight:bold; color:red">
      <xsl:value-of select="." />
   </span>
</xsl:template>

<xsl:template match="preis">
   <b>
      <xsl:value-of select="." />
   </b>
</xsl:template>

</xsl:stylesheet>
```

Um die HTML-Ausgabe optisch aufzupeppen, werden hier Inline-Stylesheets verwendet.

7.10.1 Hyperlinks und Grafiken

In den bisherigen Beispielen wurden ausnahmslos Texte ausgegeben. Das ist zwar auch in der Praxis sehr oft der Fall, aber eben nicht immer. Oftmals will man auch Hyperlinks definieren und Grafiken anzeigen. Genau darum geht es auf den folgenden Seiten.

Den Anfang macht die Definition von Hyperlinks. Zunächst das entsprechende Ausgangsdokument:

Listing 7.98 Hier wird der Link definiert.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="links.xsl" ?>
<bibliothek>
    <link>http://www.hanser.de/</link>
</bibliothek>
```

Hier wurde ein `link`-Element definiert. Dieses Element soll im Ergebnisdokument als echter Hyperlink fungieren, also anklickbar sein. Das entsprechende Stylesheet, das genau dafür sorgt, sieht folgendermaßen aus:

Listing 7.99 Jetzt wird der Link generiert.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
   <html>
      <head>
      </head>
   <body>
      <xsl:apply-templates />
   </body>
   </html>
</xsl:template>

<xsl:template match="link">
   <a>
      <xsl:attribute name="href">
         <xsl:value-of select="." />
      </xsl:attribute>
         <xsl:value-of select="." />
   </a>
</xsl:template>

</xsl:stylesheet>
```

Dem a-Element wird das `href`-Attribut zugewiesen. Als Wert erhält dieses Attribut den Inhalt des `link`-Elements.

Das Ergebnisdokument sieht auf wie auf **Abbildung 7.26**.

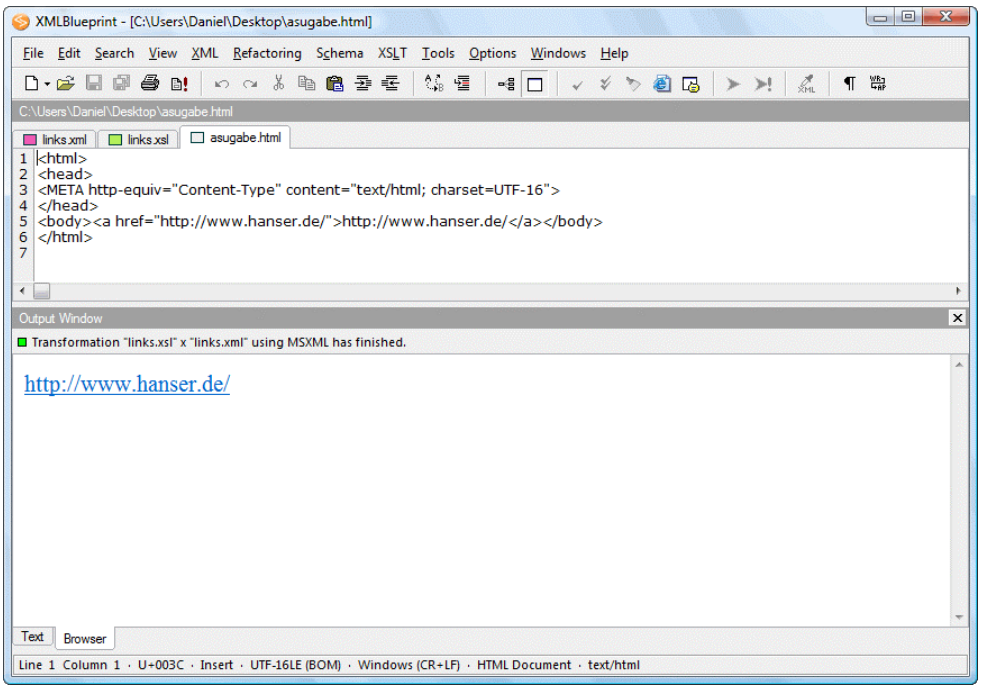

Abbildung 7.26 Es ist ein echter Hyperlink.

7.10.1.1 Grafiken einfügen

Das gleiche Prinzip funktioniert auch bei der Integration von Grafiken. Wie das genau geht, wird ebenfalls wieder anhand eines Beispiels gezeigt. Zunächst das entsprechende Ausgangsdokument.

Listing 7.100 Das ist das Ausgangsdokument.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="grafiken.xsl" ?>
<bibliothek>
   <grafik>images/windows.jpg</grafik>
</bibliothek>
```

Hier wurde das Element `grafik` definiert. Als Inhalt werden diesem Element der Pfad und der Dateiname der anzuzeigenden Grafik zugewiesen. Diese Informationen genügen, um die Grafik anzuzeigen. Das entsprechende Stylesheet sieht folgendermaßen aus:

Listing 7.101 Die Grafik wird eingebunden.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

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

```
      <head>
      </head>
   <body>
      <xsl:apply-templates />
   </body>
   </html>
</xsl:template>

<xsl:template match="grafik">
   <img>
      <xsl:attribute name="src">
         <xsl:value-of select="." />
      </xsl:attribute>
   </img>
</xsl:template>

</xsl:stylesheet>
```

Für die Anzeige der Grafik wird das `img`-Element verwendet. Als Attribut weist man diesem `src` zu. Der Wert dieses Attributs ist der Inhalt des `grafik`-Elements, also Pfad und Name des Bildes. Mehr muss für die Anzeige von Grafiken nicht getan werden. Das beweist auch ein Blick in den Browser.

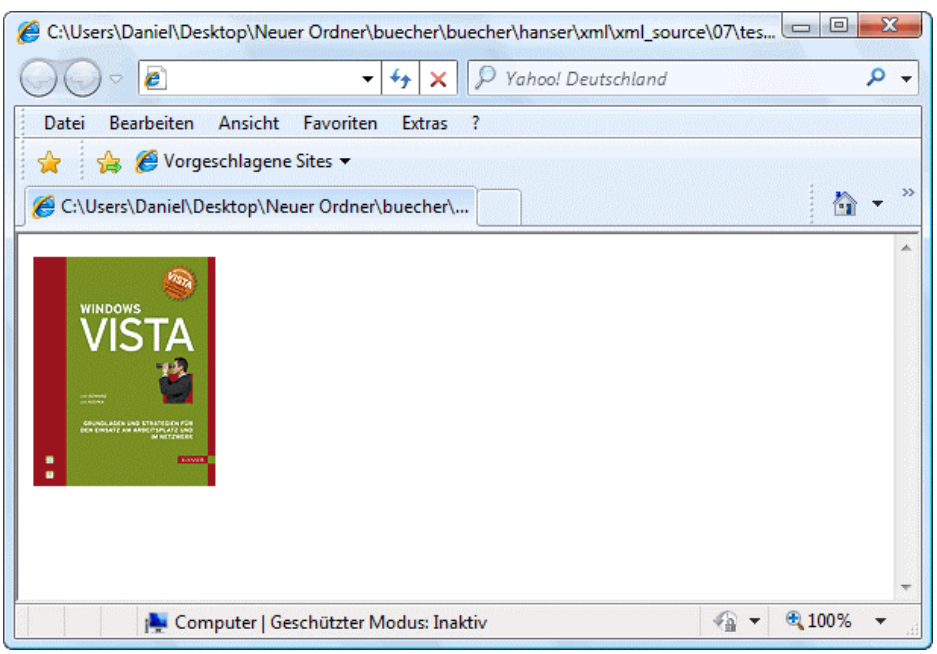

Abbildung 7.27 Das Bild wurde eingefügt.

7.11 Weitere Neuerungen in XSLT 2.0

Im Laufe dieses Kapitels wurden bereits einige Neuerungen vorgestellt, die XSLT 2.0 mit sich bringt. Dazu gehören beispielsweise die gezeigten neuen Funktionen. Neben den Funktionen wurden aber auch neue Elemente eingeführt, die in diesem Buch natürlich nicht unerwähnt bleiben sollen.

7.11.1 Neue Elemente

In XSLT 2.0 wird eine Vielzahl neuer Elemente eingeführt. Diese beziehen sich hauptsächlich auf Sequenzen. Auf den folgenden Seiten werden die neuen Elemente kurz vorgestellt. So bekommen Sie gleich einen Überblick, was sich so alles im Bereich der Elemente in XSLT 2.0 getan hat.

An dieser Stelle darf nicht der Hinweis fehlen, dass ein Blick in die offizielle Spezifikation immer empfehlenswert ist. Zu finden ist diese im Fall von XSLT 2.0 unter *http://www.w3.org/TR/xslt20/*.

7.11.1.1 xsl:analyze-string

Dieses Element wendet auf einen String, der von einem `select`-Attribut geliefert wird, den regulären Ausdruck an, der über `regex` angegeben wurde.

Listing 7.102 So kann man mit regulären Ausdrücken arbeiten.

```
<xsl:analyze-string select="abstract" regex="\n">

   <xsl:matching-substring>
      <br/>
   </xsl:matching-substring>

   <xsl:non-matching-substring>
      <xsl:value-of select="."/>
   </xsl:non-matching-substring>

</xsl:analyze-string>
```

7.11.1.2 xsl:character-map

Wird verwendet, um während der Serialisierung ganz gezielt bestimmte Zeichen durch andere zu ersetzen.

Listing 7.103 So wird die Serialisierung erreicht.

```
<xsl:character-map name="chars">
  <!-- &#x5397E; -->
  <xsl:output-character character="„" string="&lt;quote>" />
  <!-- &#x5397C; -->
  <xsl:output-character character="‟" string="&lt;/quote>" />
</xsl:character-map>
```

7.11.1.3 xsl:document

Darüber lassen sich neue Dokumentknoten erzeugen. Die allgemeine Syntax für dieses Element sieht folgendermaßen aus:

Listing 7.104 Ein Beispiel für `xsl:document`

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

 <xsl:template match="/">
    <xsl:document href="news.html">
        <html><body>
```

```
            <xsl:apply-templates/>
          <hr/></body></html>
      </xsl:document>
     <a href="news.hml">Neuigkeiten</a>
  </xsl:template>

  </xsl:stylesheet>
```

7.11.1.4 xsl:for-each-group

Hierüber lassen sich aus einer Sequenz von Knoten oder Einzelwerten Gruppen bilden.

Listing 7.105 So wird `for-each-group` eingesetzt.

```
<xsl:for-each-group select="cities/city" group-by="@country">
    <tr>
      <td><xsl:value-of select="position()"/></td>
      <td><xsl:value-of select="@country"/></td>
      <td>
        <xsl:value-of select="current-group()/@name" separator=", "/>
      </td>
      <td><xsl:value-of select="sum(current-group()/@pop)"/></td>
    </tr>
  </xsl:for-each-group>
```

7.11.1.5 xsl:function

Hierüber lassen sich benutzerdefinierte XSLT-Funktionen generieren. Diese Funktionen wiederum lassen sich dann von XPath 2.0-Ausdrücken aufrufen. Ein Beispiel für die Definition einer solchen Funktion:

Listing 7.106 Eine Funktion wird definiert.

```
<xsl:function name="str:reverse" as="xs:string">
  <xsl:param name="sentence" as="xs:string"/>
  <xsl:sequence select="if (contains($sentence, ' '))
     then concat(str:reverse(substring-after($sentence, ' ')),
     ' ',
     substring-before($sentence, ' '))
       else $sentence"/>
</xsl:function>
```

Verwenden lässt sich diese Funktion dann folgendermaßen:

Listing 7.107 Und so nutzt man sie weiter.

```
<xsl:template match="/">
<output>
  <xsl:value-of select="str:reverse('Hallo, Welt')"/>
</output>
</xsl:template>
```

7.11.1.6 xsl:import-schema

Dieses Element wird verwendet, um darüber XML-Schema-Komponenten zu identifizieren, die vorhanden sein müssen, bevor Quelldokumente verfügbar sind. Wie so etwas aussehen kann, zeigt das folgende Beispiel:

Listing 7.108 Ein Inline-Schema-Dokument in Aktion

```
<xsl:import-schema>

  <xs:schema targetNamespace=http://localhost/ns/yes-no
             xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:simpleType name="local:yes-no">
      <xs:restriction base="xs:string">
        <xs:enumeration value="yes"/>
        <xs:enumeration value="no"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:schema>
</xsl:import-schema>

<xsl:variable name="condition" select="'yes'" as="local:yes-no"/>
```

Dieses Beispiel zeigt ein Inline-Schema-Dokument. Dort wird ein einfacher Typ `local:yes-no` deklariert. Diesen Typ kann das Stylesheet dann innerhalb einer Variablen nutzen. In dem Beispiel wird von der Namensraum-Deklaration `xmlns:local=http://localhost/ns/yes-no` ausgegangen.

7.11.1.7 xsl:matching-substring

Dieses Element wird innerhalb von `xsl:analyze-string` verwendet, um die vorgegebene Aktion festzulegen, die bei denjenigen Teil-Strings ausgeführt werden soll, auf die der reguläre Ausdruck passt. Wie dieses Element eingesetzt werden kann, zeigt das folgende Beispiel. Zunächst das Ausgangsdokument.

Listing 7.109 Das Element `xsl:matching-substring` wird verwendet.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="ausgabe.xsl" ?>
<autoren>
   <autor>Hornby Geiger Ellis</autor>
</autoren>
```

Das Stylesheet sieht folgendermaßen aus:

Listing 7.110 Das ist das Stylesheet.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:output  method="xml"  indent="yes" omit-xml-declaration="yes"/>

<xsl:template  match="/autoren">
   <xsl:analyze-string  select="autor" regex="\S+">
      <xsl:matching-substring>
         <autoren>
            <xsl:value-of  select="."/>
         </autoren>
      </xsl:matching-substring>
   </xsl:analyze-string>
</xsl:template>

</xsl:stylesheet>
```

Und jetzt noch die Ausgabe.

Listing 7.111 Das Ergebnis sieht so aus:

```
<autoren>Hornby</autoren>
<autoren>Geiger</autoren>
<autoren>Ellis</autoren>
```

In den aktuellen Browser-Versionen wird diese Funktion noch nicht funktionieren.

7.11.1.8 xsl:namespace

Hierüber lässt sich ein Namensraumknoten erzeugen. Auch hierzu wieder ein Beispiel.

Listing 7.112 Der Namensraumknoten wird erzeugt.

```
<data xsi:type="xs:integer"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<xsl:namespace name="xs" select="'http://www.w3.org/2001/XMLSchema'"/>
<xsl:text>42</xsl:text>
</data>
```

Diese Syntax sorgt für die Ausgabe eines Dokuments, in dem der Namensraumknoten verwendet werden kann.

Listing 7.113 So sieht das Ergebnis aus.

```
<data xsi:type="xs:integer"
     xmlns:xs=http://www.w3.org/2001/XMLSchema
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">42</data>
```

7.11.1.9 xsl:next-match

Dieses Element wird verwendet, um die Template-Regel anzugeben, die als nächstes verwendet werden soll. So lässt sich auf einfache Weise die Template-Regel überschreiben. (Vergleichbar ist das mit dem bereits vorgestellten Element `xsl:apply-imports`.) Die allgemeine Syntax sieht folgendermaßen aus:

S**Listing 7.114** So wird die nächste Template-Regel angegeben.

```
<xsl:next-match>
  <!-- Inhalt: (xsl:with-param | xsl:fallback)* -->
</xsl:next-match>
```

7.11.1.10 xsl:non-matching-substring

Dieses Element wird innerhalb von `xsl:analyze-string` verwendet, um auf diese Weise die vorgegebene Aktion festzulegen, die bei den Teil-Strings ausgeführt werden soll, auf die der reguläre Ausdruck nicht passt.

Listing 7.115 Die Aktion wird festgelegt.

```
<xsl:non-matching-substring>
   <addrLine2>
      <xsl:value-of select="$elValue"/>
   </addrLine2>
</xsl:non-matching-substring>
```

7.11.1.11 xsl:output-character

Wird verwendet, um einen einzelnen Eintrag innerhalb von `xsl:character-map` festzulegen. Im folgenden Beispiel wird `xsl:output-character` eingesetzt, um die beiden Zeichen „ und ‟ auf Start- und Endtags für den Elementtyp `quote` abzubilden.

Listing 7.116 Die Zeichen werden wie gewünscht abgebildet.

```
<xsl:character-map name="chars">
  <!-- &#x5397E; -->
  <xsl:output-character character="„" string="&lt;quote>" />
  <!-- &#x5397C; -->
  <xsl:output-character character="‟" string="&lt;/quote>" />
</xsl:character-map>
```

7.11.1.12 xsl:perform-sort

Dieses Element wird verwendet, um eine Sequenz zu sortieren. Auch hierzu wieder ein Beispiel:

Listing 7.117 Die Elemente werden sortiert.

```
<xsl:perform-sort select="/">
    <xsl:sort select="autor/nname"/>
    <xsl:sort select="authr/vname"/>
</xsl:perform-sort>
```

7.11.1.13 xsl:result-document

Durch dieses Element wird das in XSLT 1.0 verwendete `xsl:output`-Element um zusätzliche Möglichkeiten erweitert. Auf diese Weise können die Eigenschaften des Ergebnisbaums noch detaillierter beschrieben werden.

Listing 7.118 Erweiterte Möglichkeiten bei der Ausgabe

```
<xsl:template match="\">
    <xsl:result-document href="buecher.html">
        <html><body bgcolor="#000000">
            <xsl:apply-templates/>
        <hr/></body></html>
    </xsl:result-document>
    <a href="buecher.html" target="_blank">Zeige das Dokument</a>
</xsl:template>
```

7.11.1.14 xsl:sequence

Dieses Element wird innerhalb eines Sequenzkonstruktors verwendet, um eine Sequenz von Knoten oder Einzelwerten zu bilden.

Listing 7.119 So werden Sequenzen gebildet.

```
<xsl:variable name="values" as="xs:integer*">
   <xsl:sequence select="(1,2,3,4)"/>
   <xsl:sequence select="@preis"/>
</xsl:variable>

<xsl:value-of select="sum($values)"/>
```

7.11.2 Rückwärtskompatibiliät

Bei all den Neuerungen, die XSLT 2.0 mit sich bringt, stellt sich unweigerlich die Frage nach der Rückwärtskompatibilität. Um diese sicherzustellen, wurde das Attribut `version` eingeführt. Dieses Attribut gibt es für alle Elementtypen, außer für `xsl:output`.

Weist man dem einleitenden Starttag eines Elements die Anweisung `version="1.0"` zu, können für jedes Element die dazugehörenden Attribute und die untergeordneten Elemente die Verarbeitung gemäß XSLT 1.0 erzwungen werden.

Um die Rückwärtskompatibilität explizit zu deaktivieren, weist man dem `version`-Attribut den Wert `2.0` zu. Auf diese Weise lässt sich in untergeordneten Elementen die Rückwärtskompatibilität überschreiben.

Listing 7.120 Die Rückwärtskompatibilität wird hergestellt.

```
<xsl:transform version="2.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <!-- ... -->

  <xsl:template match="bibliothek" version="1.0">
    <xsl:for-each-group  version="2.0"
                         select=".//author" group-by="nnamae">
      <xsl:sort select="current-grouping-key()"/>
      <xsl:apply-templates select="current-group()[1]" />
      <xsl:apply-templates
          select="current-group()/ancestor::isbn" />
    </xsl:for-each-group>
  </xsl:template>

  <!-- ... -->

</xsl:transform>
```

Im gezeigten Beispiel wird über die Angabe `version="2.0"` im `xsl:transform`-Element signalisiert, dass sich die Syntax bei der Transformationsspezifikation an XSLT 2.0 orientiert. Ausnahme bildet hier allerdings die Template-Regel für den Elementtyp `bibliothek`. Diese soll nämlich so behandelt werden, wie es in XSLT 1.0 üblich ist. Ein Problem gibt es dabei allerdings: Denn innerhalb des `bibliothek`-Elements taucht ein Elementtyp auf, den es so nicht in XSLT 1.0 gab. Damit dieser dennoch wie gewünscht vom XSLT-Prozessor verarbeitet werden kann, wird mit `version="2.0"` die Angabe des `version`-Attributs des übergeordneten Elements überschrieben.

Eine vollständige Rückwärtskompatibilität ist allerdings nicht möglich. Denn alle Unterschiede zwischen XSLT 1.0 und XSLT 2.0 lassen sich dadurch nicht ausgleichen. Vor allem die Ergebnisse von XPath-Ausdrücken unterscheiden sich in aller Regel. Das gilt zumindest dann, wenn eine Schemaspezifikation der Auswertung zugrunde liegt.

Aber nicht nur semantische Unterschiede gibt es. Auch bei der Syntax verhalten sich XSLT 1.0 und XSLT 2.0 verschieden. So wird in XSLT 2.0 an verschiedenen Stellen eine explizite Klammerung von Ausdrücken gefordert. Bei XSLT 1.0 war diese noch optional. Weiterer Unterschied: Im Gegensatz zu XSLT 1.0 sind bei XSLT 2.0 Leerzeichen nach Bezeichnern obligatorisch. Auch die Regeln für das Umwandeln von Zeichenketten in

Zahlenwerte wurden erweitert. Jetzt werden auch Werte mit positivem Vorzeichen, Gleitkommawerte und die Werte positiv unendlich (`INF`) und negativ unendlich (`-INF`) unterstützt.

8

8 Formatierungen mit XSL-FO

Dieses Kapitel widmet sich den XSL Formatting Objects (XSL-FO). Bei XSL-FO handelt es sich um ein XML-Vokabular, das Teil der XSL-Spezifikation ist und mit dem sich seitenorientierte Ausgabeformate definieren lassen. Mit XSL-FO kann man z.B. festlegen, wie Grafiken und Text auf einer Seite angeordnet werden sollen. Die Fähigkeiten von XSL-FO reichen allerdings noch viel weiter. Die gehen sogar so weit, dass sich mit XSL-FO hochwertige Druckerzeugnisse generieren lassen. Neben dem Druck ist XSL-FO aber auch für die Anzeige am Bildschirm und sogar für Sprachsysteme interessant.

Nachdem in diesem Kapitel anfangs die Grundidee und der eigentliche Verarbeitungsprozess im Vordergrund stehen, lernen Sie im weiteren Verlauf die grundlegenden Formatierungsobjekte von XSL-FO kennen.

8.1 Die Idee hinter XSL-FO

Im Verlauf dieses Buchs ist eines besonders deutlich geworden: XML ermöglicht die Trennung von Inhalt und Layout. Durch diese Trennung kann man denselben Inhalt unterschiedlich darstellen. Einige dieser Darstellungsvarianten wurden in diesem Buch bereits vorgestellt. Denken Sie dabei nur an SVG o.Ä.

In diesem Kapitel geht es um XSL-FO, eine Sprache, die für die Steuerung von Formatierern da ist. Dabei besteht jedes XSL-FO-Dokument ausschließlich aus Formatierungsinformationen und reinem Text.

XSL-FO hat es sich zum Ziel gesetzt, XML-Dokumente als Seiten auszugeben. Dabei spielt es zunächst keine Rolle, ob die Dokumente für den Druck oder für die Anzeige auf elektronischen Geräten mittels entsprechender Software gedacht sind. Dafür muss einerseits ein Format für die Beschreibung der Seiten vorhanden sein, die erzeugt werden sollen. Andererseits braucht man aber auch Regeln für die Transformation der XML-Dokumente in dieses Format. Und genau für diesen Zweck wurde eben XSL entwickelt. XSTL für die Transformation wurde bereits im vorherigen Kapitel vorgestellt. Mit XSL-FO werden nun die Zieldaten beschrieben.

Bei XSL-FO handelt es sich um ein XML-Vokabular. Das hat den Vorteil, dass man sich vollständig im XML-Umfeld bewegen kann. So können Sie Ihren XML-Editor verwenden, und auch das Erlernen von XSL-FO fällt – wenn XML-Kenntnisse vorhanden sind – deutlich leichter.

8.1.1 Eigenschaften der Sprache

Bei XSL-FO handelt es sich um eine Seitenbeschreibungssprache, die aus dem Dunstkreis von DSSSL (Document Style Semantics and Specification Language) stammt. Bei DSSSL handelt es sich um eine Formatierungssprache für SGML-Dokumente. Zudem wurde XSL-FO zeitweise parallel zu CSS entwickelt. Allerdings unterscheiden sich beide Stylesheet-Ansätze deutlich.

Tabelle 8.1: Unterschiede zwischen XSL-FO und CSS

	XSL-FO	**CSS**
Seitenmodell	Bereiche	Boxen
Prozessmodell	Erst XSLT, dann XSL-FO	Ad-hoc-Formatierung mit CSS-Renderern

In XSL-FO sind Elemente und Attribute u.a. für folgende Dinge enthalten:

- Regionen einer Seite
- Seitenränder
- Bereiche einer Seite
- Abfolge von Seiten
- Seitenzahlen
- Rahmen
- Blöcke
- Spalten
- Tabellen
- Absätze
- Listen
- Grafiken
- Linien
- Textformatierungen

Diese – übrigens bei Weitem nicht vollständige – Liste zeigt, wie mächtig XSL-FO ist. Im weiteren Verlauf dieses Kapitel sehen Sie, wie sich die gezeigten Elemente einsetzen lassen.

8.1.2 Einsatzgebiete von XSL-FO

Die Aufgabe von XSL-FO ist die gleiche, wie sie Satzprogramme im Bereich des DTP (Desktop Publishing) übernehmen. XSLT-FO beschreibt Layout und Typografie der Seiten. Satzprogramme wie InDesign oder FrameMaker besitzen eine grafische Benutzeroberfläche. Zudem enthalten die meisten dieser Anwendungen bereits eine entsprechende XML-Schnittstelle. Dank dieser lassen sich XML-Dokumente importieren und abspeichern. Allerdings setzen die Satzprogramme nach wie vor auf proprietäre Lösungen. XSL-FO kommt dort zumeist noch nicht zum Zug.

Für XSL-FO gibt es momentan leider noch keine übersichtlichen grafischen Benutzeroberflächen. Daher bleibt nur der Weg über den Quelltext. Das ist für XSL-FO natürlich ein Handicap. Denn kaum ein Setzer oder Layouter möchte oder kann sich mit XSL-FO-Quelltexten auseinandersetzen.

Bislang ist es bei XSL-FO noch nicht möglich, das Ergebnis der Formatierung mit einem XSL-FO-Stylesheet interaktiv zu beeinflussen. Somit kann ein Layouter nicht in Einzelfällen Dinge definieren, die vom XSL-FO-Standard abweichen. Ohnehin ist XSL-FO für den Einsatz im Bereich des Massensatzes gedacht. Aber auch die gleichzeitige Ausgabe desselben Inhalts in verschiedenen Formaten ist für XSL-FO ein ideales Einsatzgebiet.

Größter Vorteil von XSL-FO ist aber zweifellos, dass die Sprache selbst aus dem XML-Universum stammt. Dadurch lassen sich Komponenten für die Verarbeitung von XSL-FO vergleichsweise einfach in DTD-Systeme u.Ä. integrieren, bei denen Daten ohnehin in XML-Formaten vorliegen oder die Daten in XML umwandeln können.

Tatsächlich sind die denkbaren Einsatzszenarien für XSL-FO enorm. Sie erstrecken sich von einfachen E-Book-Anwendungen bis hin zu großindustriellen Massenpublikationen, die einen riesigen Umfang besitzen und in verschiedenen Sprachen erscheinen. Typische Branchen, bei denen XSL-FO eingesetzt sind:

- Banken
- Verlage
- öffentliche Verwaltungen
- technische Dokumentationen
- die IT-Industrie

Die Vorteile von XSL-FO sind bereits jetzt von vielen Entscheidern erkannt worden.

8.1.3 Die Spezifikation des Vokabulars

XSL-FO ist Teil von XSL. Die erste Version der Spezifikation zu XSL-FO wurde nach langem Hin und Her im Oktober 2001 verabschiedet. Interessant ist innerhalb der XSL-Spezifikation hinsichtlich XSL-FO der Abschnitt zu den Formatting Objects (*http://www.w3.org/TR/xsl11/#fo-section*).

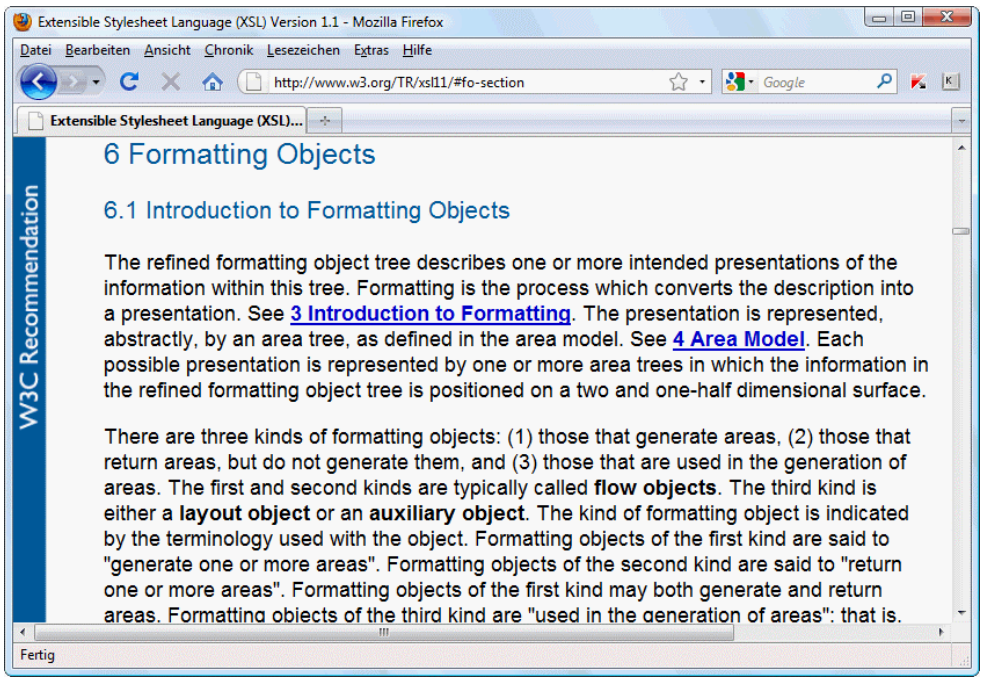

Abbildung 8.1 Hier gibt es Informationen zu den Formatting Objects.

Innerhalb von Fomatting Objects sind die Elementtypen und Attribute definiert, mit denen sich die Struktur von Seiten über die Formatierungsobjekte beschreiben lässt.

Die Formatierungsobjekte stellen in dem XSL zugrunde liegenden Modell eine Baumstruktur dar. Dabei bildet z.B. eine Webseite die Wurzel, von der aus zu verschiedenen Seitenelementen (Tabellen, Textabschnitte usw.) Äste verzweigen. Bei den Blättern wiederum handelt es sich um die einzelnen Zeichen oder beispielsweise auch Bilder, die sich nicht weiter aufteilen lassen.

Formatierungsobjekte sind Instanzen von abstrakten Klassen, über die typografische Elemente wie Absätzen, Zeilen, Seiten usw. bezeichnet werden. Jedes dieser Objekte lässt sich über einen Satz von Eigenschaften detailliert beschreiben.

Die Objekte werden als Elemente definiert, die entsprechenden Eigenschaften sind Attribute dieser Elemente. Der Elementinhalt, der aus den Elementen des Quelldokuments übernommen wird, erscheint als Inhalt des betreffenden Formatierungsobjekts. Alle Formatierungsobjekte und die Formatierungseigenschaften gehören zu dem Namensraum mit folgendem URI:

http://www.w3.org/1999/XSL/Format

Als Präfix wurde `fo` zugewiesen. Aus dem bisher Beschriebenen wird deutlich, dass typische XSL-FO-Elemente folgendermaßen aussehen:

Listing 8.1 So sieht XSL-FO-Syntax aus.

```
<fo:block>

   <fo:block text-align="center">
      <fo:external-graphic src="'url(hornby.jpg)'"/>
   </fo:block>

   <fo:block space-before="3pt" text-align="center"
   start-indent="10mm" end-indent="10mm">
      Nickt Hornby
   </fo:block>

</fo:block>
```

Die XSL-FO-Spezifikation richtet sich in erster Linie an diejenigen, die die Sprache implementieren müssen. Das sind vor allem all diejenigen, die Software für die Darstellung von XSL-FO-Dokumenten entwickeln. Explizit nicht wendet sich die Spezifikation an Entwickler von Stylesheets. Als alleinige Dokumentationsquelle eignet sich die Spezifikation daher nur bedingt.

Wichtig waren den Entwicklern der Spezifikation vor allem Aspekte wie Zugänglichkeit (Accessibility) und Internationalisierung. So ist eines der wichtigsten Ziele, dass sich ein und dasselbe Stylesheet für verschiedene Schriftsysteme nutzen lässt. Ebenso ist es auch möglich, Dokumente zu formatieren, in denen unterschiedliche Schriftsysteme kombiniert werden.

- Schrift läuft von rechts nach links.
- Schrift läuft von links nach rechts.
- Schrift läuft von oben nach unten.

Ein weiteres Feld von XSL-FO betreffen Sprachausgabesysteme. Denn in der Tat soll sich XSL-FO auch dort einsetzen lassen. Durch akustische Stylesheets lassen sich u.a. Lautstärke, Art der Stimme und Sprachgeschwindigkeit definieren.

8.1.4 Der Verarbeitungsprozess

Die Verarbeitung von XML mittels XSL-FO verläuft in mehreren Schritten, von denen jeder einzelne von einer speziellen Software bzw. einem speziellen Software-Modul abgearbeitet wird.

Zunächst werden die von der Transformation gelieferten Objekte in die Formatierungsobjekte umgewandelt, und der Baum der Formatierungsobjekte wird erzeugt. In dieser Phase werden die einzelnen Zeichen, die den Inhalt der Quelldaten liefern, in die entsprechenden `fo:`-Zeichenknoten umgewandelt.

Anschließend wird der jetzt noch vergleichsweise grob daherkommende Baum der Formatierungsobjekte weiter verfeinert. So werden beispielsweise nicht benötigte Leerzeichen und Duplikate entfernt.

Im dritten Schritt wird ein Baum aus Bereichen aufgebaut. Über diesen Baum wird die gesamte Ausgabe definiert. Die den Formatierungsobjekten zugewiesenen Eigenschaften

müssen dabei in die endgültigen Merkmale umgewandelt werden, die vom Formatter umgesetzt werden können.

Bei den erwähnten Eigenschaften handelt es sich um sogenannte *Traits*. Diese entsprechen normalerweise den angegebenen Eigenschaften. Es gibt aber auch Situationen, in denen sie zunächst berechnet werden müssen. Das ist z.B. der Fall, wenn Schriftgrößen relativ zu anderen definiert wurden.

Den eigentlichen Prozess für die Verarbeitung und Layoutierung zeigt Abbildung 8.2.

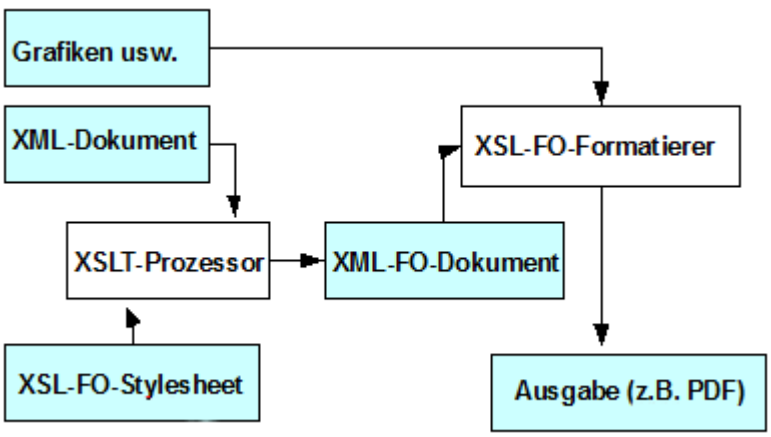

Abbildung 8.2 So sieht der Verarbeitungsprozess aus.

Die XML-Dateien und das XSL-FO-Stylesheet werden vom XSLT-Prozessor eingelesen. Aus Dokument und Stylesheet generiert der Prozessor mittels der Stylesheet-Anweisungen, die das Namensraum-Präfix `xsl` haben, das XML-FO-Ergebnisdokument.

Das XML-FO-Dokument wird vom Formatierer eingelesen. Die innerhalb dieses Dokuments referenzierten Dateien werden vom Formatierer an der Stelle ins Dokument eingebunden, an denen die betreffende Referenz steht.

Was der FO-Formatierer letztendlich ausgibt bzw. welches Ausgabeformat dabei genutzt wird, ist nicht vorgeschrieben. Sehr oft – und das werden Sie bei Ihren ersten Schritten mit XLS-FO wahrscheinlich ebenfalls tun – wird allerdings PDF verwendet.

8.1.5 Diese XSL-FO-Werkzeuge gibt es

Im Zusammenhang mit XSL-FO kommen verschiedenste Werkzeuge zum Einsatz. Am wichtigsten sind aber natürlich die FO-Formatierer, deren Funktionsweise Sie im vorherigen Abschnitt kennengelernt haben. Auf den folgenden Seiten werden die interessantesten Formatierer kurz vorgestellt.

In diesem Kapitel wird mit FOP gearbeitet, dem am häufigsten eingesetzten Formatierer. Letztendlich spielt es aber keine Rolle, welcher Formatierer hier eingesetzt wird. Denn es

geht lediglich darum, die Funktionsweise von XSL-FO zu zeigen. Und dazu können Sie dann natürlich den Formatierer Ihrer Wahl einsetzen.

FOP hat allerdings den Vorteil, dass er Open Source ist und somit kostenlos eingesetzt werden kann.

Zuvor noch ein paar allgemeine Hinweise zu den Formatierern. Leider gibt es Probleme zwischen der XSL-FO-Spezifikation und den tatsächlichen Anforderungen an die Stylesheet-Sprache. Denn die FO-Formatierer müssen proprietäre Funktionen unterstützen, um überhaupt praxistauglich zu sein. So gibt es beispielsweise Funktionalitäten, die von der XSL-FO-Spezifikation nicht exakt beschrieben werden. Ein typisches Beispiel dafür ist die Silbentrennung.

Da die Formatierer also proprietäre Erweiterungen integriert haben, unterscheiden sich die Produkte auch in ihrer Leistungsfähigkeit. Daher muss man exakt abwägen, welches Tool den eigenen Ansprüchen gerecht wird.

Die folgende Liste stellt einige der bekanntesten XSL-FO-Tools vor. Welches Sie dann letztendlich einsetzen, hängt davon ab, was Sie erreichen wollen bzw. was das Werkzeug leisten können muss.

8.1.5.1 FOP

Der Formatierer FOP (Formatting Objects Processor) ist sicherlich das bekannteste Tool seiner Art. Entwickelt wird diese Open-Source-Software von der Apache Software Foundation. Während die erste Version vor allem durch eingeschränkte Funktionalität und fehlende Standardkompatibilität auffiel, hat sich das mit der aktuellen Version deutlich geändert. Allerdings gibt es immer noch Funktionseinschränkungen, die den Einsatz von FOP nicht überall empfehlenswert machen. Hervorzuheben sind hier vor allem die mangelhaften Möglichkeiten bei der Umsetzung von Tabellen und die fehlende Unterstützung von Online-Containern.

Beachten Sie außerdem, dass FOP von den in XSL-FO 1.1 neu spezifizierten Eigenschaften lediglich die PDF-Lesezeichen unterstützt. Hier hat das Tool also noch deutlichen Nachholbedarf.

Da es sich bei FOP um eine Java-Anwendung handelt, ist der Formatierer plattformunabhängig. Ausführliche Informationen zu FOP und entsprechende Download-Hinweise finden Sie auf der Projektseite *http://xmlgraphics.apache.org/fop/index.html*.

8.1.5.2 XEP

Im Gegensatz zu FOP handelt es sich bei XEP, das Tool stammt übrigens aus dem Hause RenderX (*http://www.renderx.com/*), um ein kommerzielles Produkt. XEP unterstützt fast den vollständigen XSL-FO-Sprachstandard, scheitert allerdings dort, wo eine „falsche“ Schreibrichtung verwendet werden soll. Einschränkungen gibt es darüber hinaus auch hinsichtlich der in XSL-FO 1.1 neu hinzugekommenen Eigenschaften. Diese werden ebenfalls nicht unterstützt.

Neben dem XSL-FO-Standard bietet das Tool auch noch zahlreiche proprietäre Funktionen. Interessant sind hier vor allem die neuen Layout-Möglichkeiten.

XEP ist vergleichsweise teuer. So gehen die Preise für Server-Lizenzen bei ca. 2.000 US-Dollar los und steigern sich bis zu 32.000 Euro. Die entsprechenden Einzelplatz-Lizenzen sind dann natürlich deutlich günstiger. Informationen zum Preismodell finden Sie unter *http://www.renderx.com/documents/RenderX_Published_Price_List.pdf*.

8.1.5.3 Ecrion XF Rendering Server

Ecrion XF Rendering Server (*http://www.ecrion.com/*) ist ausschließlich für Windows verfügbar. Das Tool unterstützt XSL-FO 1.0 fast vollständig. Lediglich die Schreibrichtungen von rechts nach links und von oben nach unten sind derzeit noch nicht implementiert. Ebenfalls passen muss dieses Tool bei den neuen Funktionen von XSL-FO 1.1.

Dieses Manko macht der Ecrion XF Rendering Server allerdings durch einige interessante proprietäre Erweiterungen wett. Interessant ist vor allem, dass als Ausgabeformat unter anderem XPS, das neue Microsoft-Format, unterstützt wird.

Den Ecrion XF Rendering Server gibt es in verschiedenen Versionen. Er wird somit auch zu unterschiedlichen Preisen angeboten. Ausführliche Informationen zur Preisgestaltung finden Sie unter *http://www.ecrion.com/*.

8.1.5.4 XSL Formatter

Der XSL Formatter stammt aus dem Hause Antenna House (*http://www.antennahouse.com/*). Das Tool gibt es für Windows, Linux und Macintosh.

Positiv hervorzuheben ist zunächst einmal, dass der XSL Formatter XSL-FO 1.0 und 1.1 fast vollständig unterstützt. Darüber hinaus hat das Tool auch noch zahlreiche proprietäre Erweiterungen zu bieten, wodurch es auch für komplexe Anwendungen nutzbar wird. Interessant sind hier vor allem die Dinge, die der XSL Formatter hinsichtlich der Grafikverarbeitung ermöglicht. Die folgenden Formate werden unterstützt:

- BMP
- JPEG
- PNG
- TIFF
- GIF
- WMF
- EMF
- EPS
- SVG
- CGM

Zusätzlich können Sie aber auch MathML-Elemente und sogar Excel-Sheets integrieren. Das folgende Beispiel zeigt, wie sich MathML-Syntax verwenden lässt:

Listing 8.2 Das ist eine typische MathML-Anwendung.

```
<fo:instream-foreign-object>
   <math xmlns="http://www.w3.org/1998/Math/MathML">
      <mrow>
         <mi>y</mi><mo>=</mo><mi>x</mi><mo>&#x2212;</mo><mn>1</mn>
      </mrow>
   </math>
</fo:instream-foreign-object>
```

Neben den genannten Funktionen hat XSL Formatter in der Windows-Version eine gut durchdachte Benutzeroberfläche zu bieten, die vor allem bei der Stylesheet-Entwicklung hilfreich ist.

Einzige (wenn auch kleiner) Wehrmutstropfen sind das etwas unübersichtliche Lizenzmodell und die hohen Preise. Eine Alternative bietet hier die eingeschränkte Light-Version des XSL Formatters. Die hat zwar nicht alle Funktionen der Vollversion zu bieten, ist dafür aber deutlich günstiger.

8.1.5.5 XML2PDF

XML2PDF basiert auf dem .Net Framework, und ist dadurch ausschließlich für Windows verfügbar. Das Tool arbeitet äußerst standardkonform und setzt einen Großteil von XSL-FO 1.1 um. Mit XML2PDF bekommen Sie ein erstklassiges Werkzeug an die Hand, das in unterschiedlichen Versionen angeboten wird. Während einige Anwendungen aus der Produktfamilie wie XML2PDF Workstation kostenlos sind, werden andere ausschließlich als Kaufversionen angeboten. So kostet der XML2PDF Server immerhin bis zu 2.495 Euro.

Ausführliche Informationen über die Produkte und das Preismodell finden Sie auf der Seite *http://alt-soft.com/Products.aspx*.

8.1.6 Prozessoren im Einsatz

In diesem Kapitel wird auf den bereits vorgestellten Prozessor FOP gesetzt. Um FOP nutzen zu können, wird ein Java Runtime Environment (JRE) benötigt. Damit das JRE überhaupt einsetzbar ist, muss das Verzeichnis, in dem sich das JRE befindet, innerhalb der Umgebungsvariablen *JAVA_HOME* angegeben werden. Anschließend kann der Prozessor über die Kommandozeile oder ein Skript aufgerufen werden.

JAVA_HOME=/usr/java/j2re /usr/bin/fop welt.fo ausgabe.pdf

Neben dem Shell-Skript enthält das FOP-Paket auch noch eine Batch-Datei für Windows-Systeme, die ebenfalls direkt für den Aufruf verwendet werden kann. Dazu wechseln Sie in das FOP-Verzeichnis und wählen z.B. folgende Syntax innerhalb der Kommandozeile.

fop welt.fo ausgabe.pdf

Wie sich FOP in nutzen lässt, lässt sich am besten anhand eines Beispiels zeigen. Als Ausgangspunkt dient folgendes XSL-FO-Dokument:

Listing 8.3 Ein typisches XSL-FO-Dokument

```
<?xml version="1.0" encoding="UTF-8"?>
<fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">

 <fo:layout-master-set>
  <fo:simple-page-master master-name="HelloWorld">
    <fo:region-body/>
  </fo:simple-page-master>
 </fo:layout-master-set>

 <fo:page-sequence master-reference="HelloWorld">
   <fo:flow flow-name="xsl-region-body">
   <fo:block>Hallo, Welt!</fo:block>
  </fo:flow>
 </fo:page-sequence>
</fo:root>
```

An dieser Stelle soll nicht auf die Syntax des XSL-FO-Dokuments eingegangen werden.

Speichern Sie die Datei im gleichen Verzeichnis, in dem FOP liegt. Zudem wird davon ausgegangen, dass die XSL-FO-Datei unter dem Namen *welt.fo* abgespeichert wurde.

Öffnen Sie nun – wenn Sie unter Windows arbeiten – die Kommandozeile/Eingabeaufforderung, und wechseln Sie dort in das Verzeichnis, in dem FOP liegt. Aussehen könnte dieser Verzeichniswechsel z.B. folgendermaßen:

chdir c:\fop

Wobei hier davon ausgegangen wird, dass sich FOP unter *C:\fop* befindet. Um sich die möglichen Parameter anzeigen zu lassen, die FOP zu bieten hat, tragen Sie *fop* ein und bestätigen dies mit *[Enter]*.

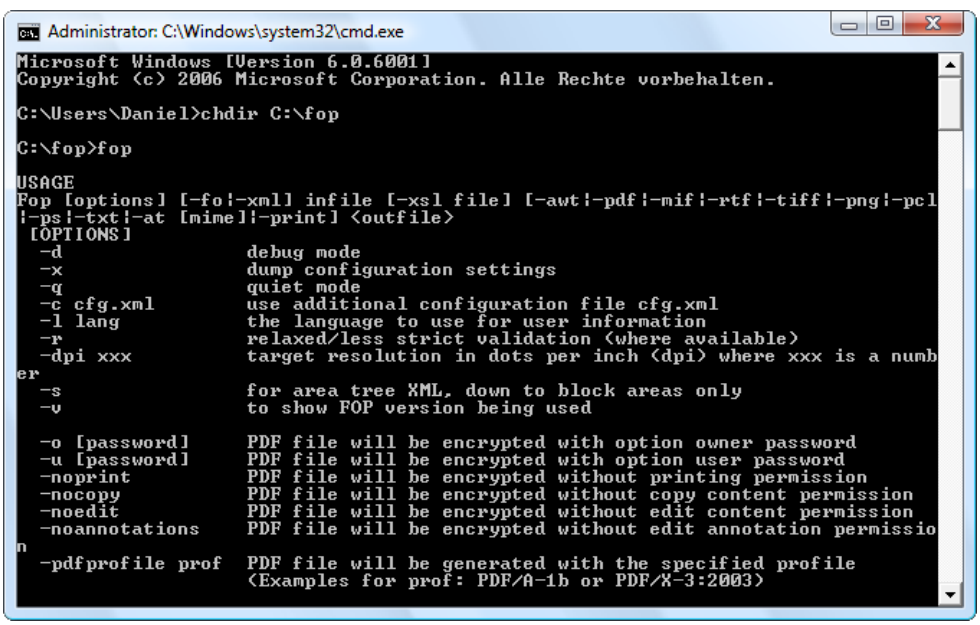

Abbildung 8.3 Diese Optionen stehen zur Verfügung.

FOP listet die verfügbaren Optionen auf und zeigt die notwendigen Syntaxvarianten an.

Im folgenden Beispiel soll aus der eingangs gezeigten XSL-FO-Syntax ein PDF-Dokument erzeugt werden. Rufen Sie FOP dazu folgendermaßen auf:[1]

fop welt.fo ausgabe.pdf

Hieraufhin wird die PDF-Datei generiert, die anschließend im gleichen Verzeichnis wie die welt.fo zu finden ist. Wenn Sie diese PDF-Datei öffnen, ergibt sich folgendes Bild:

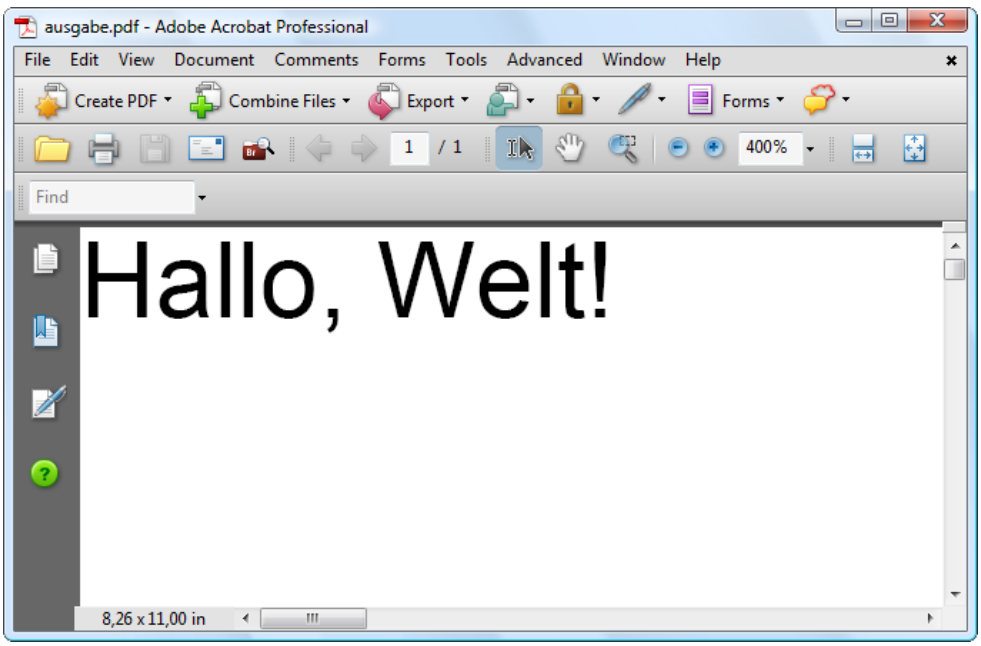

Abbildung 8.4 Das PDF-Dokument wurde generiert.

Dieses einfache Beispiel hat gezeigt, mit wie wenig Aufwand sich Prozessoren wie FOP nutzen lassen.

Noch ein Hinweis zur möglichen Fehlermeldungen. Schwerwiegende Syntaxfehler werden sofort geahndet. Eine Generierung des Zieldokuments findet in solchen Fällen nicht statt.

[1] Dazu muss die gezeigte Syntax in der Datei *welt.fo* abgespeichert werden, die im gleichen Verzeichnis wie FOP liegt.

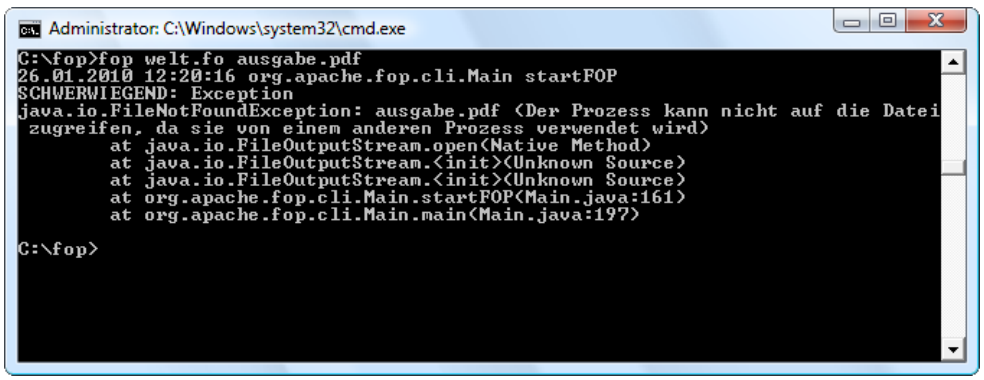

Abbildung 8.5 Ein Syntaxfehler ist aufgetreten.

Bekommen Sie also eine Exception angezeigt, wie sie auf **Abbildung 8.5** zu sehen ist, liegt das an einem Syntaxfehler.

Eine weitere Fehlermeldung, mit der man es immer mal wieder zu tun bekommt, ist die, die auf **Abbildung 8.6**zu sehen ist.

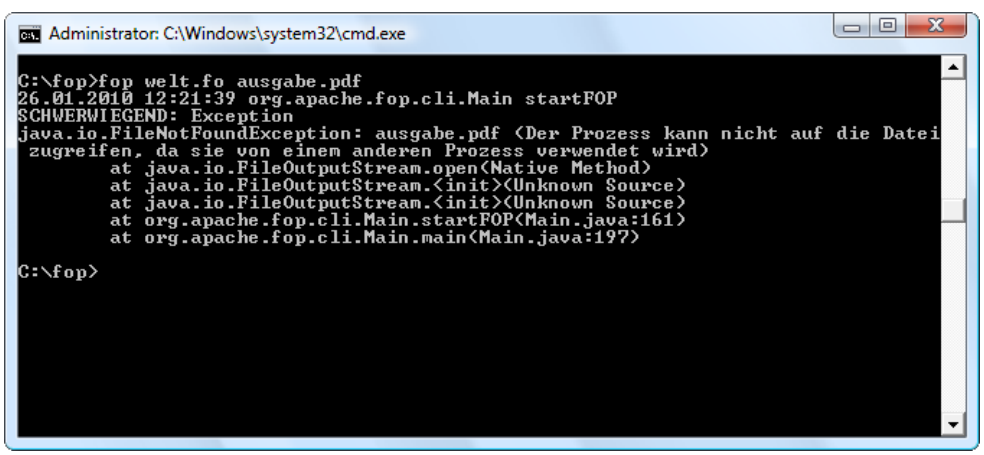

Abbildung 8.6 Das Zieldokument war bereits geöffnet.

Hier wurde versucht, ein Dokument zu generieren, das bereits geöffnet war. Schließen Sie also immer erst das PDF-Dokument, bevor Sie es neu generieren.

Und zu guter Letzt noch ein Aspekt, der Ihnen hin und wieder begegnen wird. Denn die Formatierer unterstützen normalerweise nicht alle verfügbaren Attribute und Eigenschaften aus der XSL-FO-Spezifikation. In solchen Fällen wird das Dokument zwar im Allgemeinen generiert, ein Warnhinweis wird aber dennoch ausgegeben.

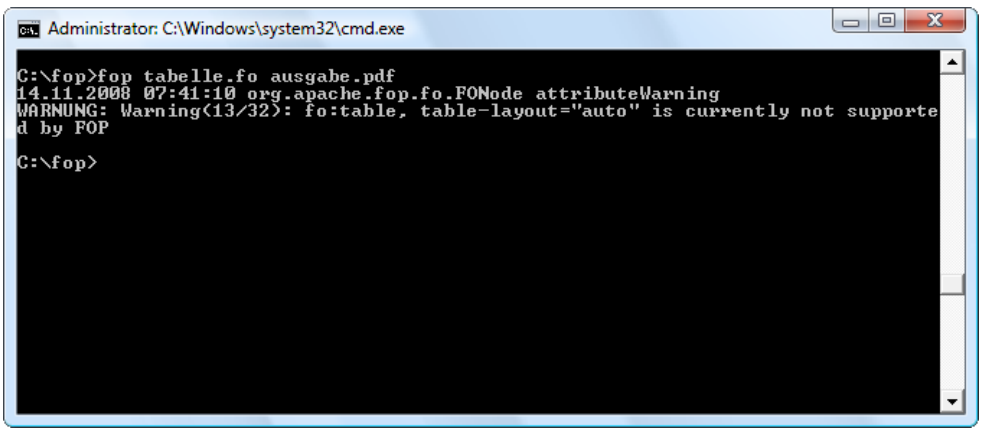

Abbildung 8.7 Nicht immer werden die verwendeten Eigenschaften interpretiert.

Im gezeigten Beispiel konnte `table-layout="auto"` vom Formatierer nicht interpretiert werden.

8.1.7 Prozessoren und die Standards

Die einzelnen Prozessoren und Formatierer machen nicht immer das Gleiche. So kann es durchaus passieren, dass ein Dokument unter FOP anders aussieht, als wenn es mit dem XSL Formatter erstellt wurde. Der Grund dafür: Die Software-Hersteller halten sich nicht immer an bestehende Standards bzw. interpretieren Standards unterschiedlich. Das soll übrigens kein Vorwurf an die Entwickler der Formatierer sein. Denn in der Tat reicht der XSL-FO-Standard in vielen Fällen nicht aus. Die Entwickler sind daher dazu gezwungen, proprietäre Lösungen in ihre Produkte zu integrieren.

Einige Beispiele für notwendige proprietäre Funktionalitäten:

- Algorithmen für Zeilenumbrüche und Silbentrennung
- Generierung des Ausgabeformats
- Fonts einbinden

Diese und andere Funktionen sind zunächst einmal fester Bestandteil des jeweiligen Formatierers. Ebenso können sie aber auch durch Sie als den Stylesheet-Entwickler gesteuert werden. Die Steuerung geschieht entweder mittels der entsprechenden Spezifikation innerhalb der produktspezifischen Konfigurationsdateien oder durch die Spezifikation der Eigenschaften im Stylesheet und das Hinzufügen eines entsprechenden Namensraums. Auf diese Weise bleibt das XSL-FO-Stylesheet immer standardkonform. Wird das Stylesheet unter einem anderen Formatierer verwendet, ignoriert dieser solche Eigenschaften, die mit einem für ihn fremden Namensraum-Präfix gekennzeichnet sind.

Da die Formatierer unterschiedlich leistungsfähig sind, stellt sich natürlich die Frage, welcher am besten eingesetzt werden sollte. Eine endgültige Antwort darauf kann hier nicht

gegeben werden. Ganz alleingelassen werden sollen Sie aber bei der Formatierer-Wahl natürlich nicht. Denn die letztendlich ist es entscheidend, welcher Formatierer eigentlich welche XSL-FO-Eigenschaften und Objekte unterstützt. Eine vollständige Liste der unterstützten Eigenschaften würde den Rahmen dieses Buches sprengen. Es gibt aber auf den Seiten einiger Hersteller entsprechende Informationen. Positiv fällt hier z.B. die zu FOP gehörende Webseite auf, die unter *http://xmlgraphics.apache.org/fop/compliance.html* zu finden ist.

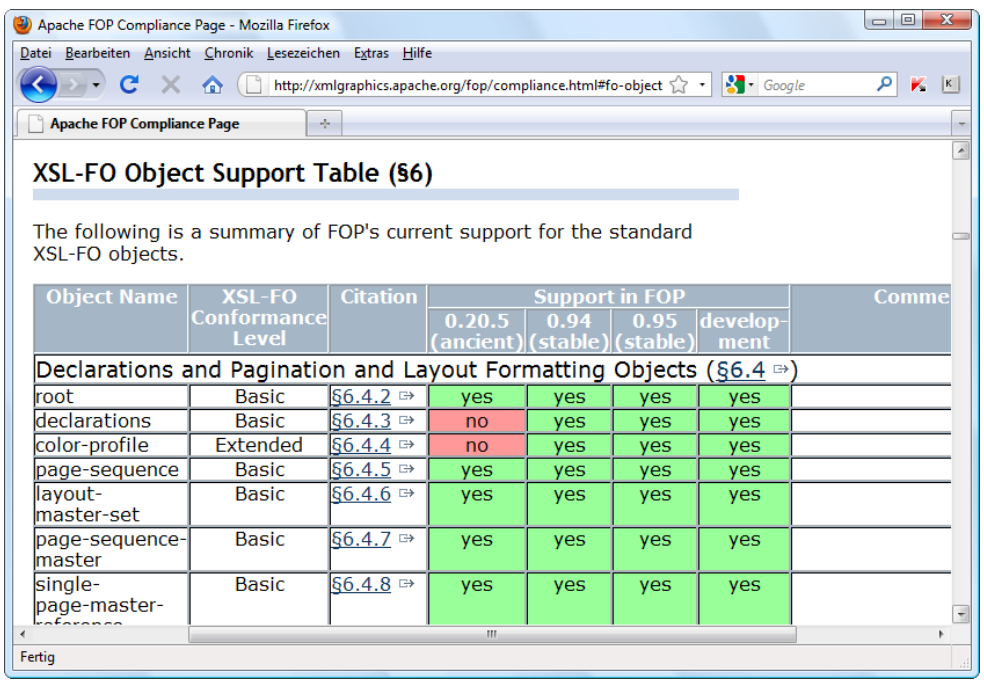

Abbildung 8.8 Welche Features unterstützt werden

Hier finden Sie eine Übersicht darüber, welche Elemente von FOP tatsächlich unterstützt werden und auf welche Sie besser verzichten sollten.

Beachten Sie, dass längst nicht alle Hersteller entsprechende Informationen auf ihren Seiten bereitstellen. Mittlerweile gibt es aber durchaus einige Webseiten, die diese Aufgabe übernommen und die nötigen Informationen zusammengetragen haben.

- XSL Formatter – *http://www.antennahouse.com/xslfo/axf4fo.htm*
- XEP – *http://www.renderx.net/Content/support/xep/reference.html*
- XF Rendering Server – *http://www.ecrion.com/Support/Resources/Matrix.aspx*

Es gibt eigentlich zu allen Produkten relevante Informationen. Als Suchbegriff für Google & Co. bietet sich *xsl-fo Compliance* an. Wenn Sie als weiteren Suchbegriff noch den Namen Ihres Formatierers angeben, finden Sie die gesuchten Daten in aller Regel.

8.2 Stylesheet-Design

Auf den vorherigen Seiten wurde gezeigt, was XSL-FO ist und wie Formatierer arbeiten. Weiter geht es nun mit dem Anlegen eigener Stylesheets.

De Formatierung mittels XSL-FO funktioniert ähnlich wie die bei CSS. Zumindest ist das dahinter liegende Konzept das gleiche. Denn genauso wie CSS formatiert XSL-FO Dokumente mithilfe rechteckiger Bereiche. Jeder dieser Bereiche hat eine bestimmte Position auf einer Seite. Darüber hinaus wird für jeden Bereich explizit festgelegt, was in ihm angezeigt werden soll. Bei den Bereichen handelt es sich um Container, in die letztendlich die Inhalte aus der XML-Quelle übernommen werden.

Die Bereiche selbst sind keine Formatierungsobjekte. Stattdessen werden die Bereiche durch die Formatierungsobjekte erzeugt. Das geschieht dadurch, dass im Ausgabedokument der entsprechende Platz belegt wird.

Die einzelnen Bereiche der Seite müssen hierarchisch angeordnet sein. Erst dadurch lassen sich die einzelnen Bereiche explizit steuern. Das funktioniert dann ähnlich, wie Sie es im Zusammenhang mit XPath bereits kennengelernt haben.

Innerhalb eines Bereichs ist ein Inhaltsrechteck enthalten. Dabei handelt es sich um eine Fläche, in der entweder Text, Grafiken usw. direkt stehen können oder in dem Kind-Bereiche definiert sind. Dieses Inhaltsrechteck kann von einem Padding-Bereich umgeben sein. Dieser Bereich sorgt für den entsprechenden Innenabstand. Um das Inhaltsrechteck und den möglicherweise definierten Innenabstand kann sich noch ein entsprechender Rahmen (`border`) ziehen.

8.2.1 Seitenlayout und Bildschirmdesign

Die meisten Entwickler dürften ihren ersten Kontakt zu Stylesheets im Zusammenhang mit der Präsentation der Daten am Bildschirm herstellen. Typischerweise formatiert man (X)HTML-Dokumente mittels CSS. Und diese Seiten werden normalerweise am Bildschirm angezeigt. Ist das Dokument länger, kann man sich mittels Scrollen durch das Dokument bewegen. So weit ist das auch völlig in Ordnung. Problematisch wird es aber, wenn man (XH)HTML-Dokumente ausdruckt. Denn das Layout von einer Darstellung für den Bildschirm unterscheidet sich von dem eines seitenorientierten Layouts. So müssen bei seitenorientierten Layouts z.B. viel mehr Dinge hinsichtlich der Paginierung u.Ä. berücksichtigt werden.

An diesem Punkt muss man bei XSLO-FO umdenken. Während man in (X)HTML „einfach“ mit der CSS-Formatierung beginnen kann, kommen bei XSL-FO zahlreiche Aspekte hinzu, die anfangs ungewohnt sind.

- Seitengrößen
- Kolumnen
- Seitenzahlen

- Randspalten
- Kopf- und Fußzeilen
- Inhaltsverzeichnis
- Register
- Titelei

Nun könnte man einwerfen, dass es für viele dieser Dinge mittlerweile sehr wohl auch für die Bildschirmdarstellung entsprechende Alternativen gibt. Das mag sein, wirklich stabil funktionieren diese aber nicht.

Hat man das Prinzip des seitenbasierten Layouts verinnerlicht, fällt XSL-FO auch nicht schwerer als CSS. Zudem darf nicht übersehen werden, dass es sehr viele Gemeinsamkeiten zwischen XSL-FO und CSS gibt. Das gilt vor allem hinsichtlich CSS Level 2 und 3. Das werden Sie z.B. feststellen, wenn es um die Definition von Rahmen, Schrift und Innenabständen geht. Spätestens da ist dann eine verwandtschaftliche Nähe nicht mehr zu übersehen.

8.2.2 Das Wurzelelement

Da es sich bei einem XSL-FO-Stylesheet um ein wohlgeformtes XML-Dokument handelt, wird zunächst die XML-Deklaration definiert. Aber Achtung: Die DOCTYPE-Deklaration fehlt hier!

Das Wurzelelement des XSL-FO-Stylesheets ist identisch mit dem eines XSLT-Stylesheets. Unterscheiden tun sich die beiden Varianten lediglich dadurch, dass innerhalb von XSL-FO-Dokumenten zusätzlich der XSL-FO-Namensraum angegeben werden muss.

Listing 8.4 Der Namensraum wurde angegeben.

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

Sollte der verwendete XSL-FO-Prozessor eigene, also proprietäre Erweiterungen anbieten und Sie wollen diese verwenden, müssen Sie zusätzlich den Namensraum des entsprechenden Prozessors angeben.

Auch wenn die XSL-Empfehlung vom W3C erst 2001 herausgegeben wurde, wird im Namensraum für die Formatierungsobjekte die Jahreszahl 1999 verwendet. Denn der URI für diesen Namensraum wurde bereits in diesem Jahr zugeordnet.

8.2.3 Stylesheets aufbauen

Für XSL-FO-Stylesheets gibt es bei der Gestaltung einige Einschränkungen bzw. Vorgaben, die es zu beachten gilt. Diese betreffen vor allem die für die Layout-Definitionen notwendigen Angaben für die Seitengestaltung, die innerhalb des Wurzelelements stehen müssen.

In diesem Abschnitt geht es zunächst darum, Ihnen einen ersten Eindruck davon zu verschaffen, wie Stylesheets aufgebaut sind. Die einzelnen Elemente werden dann im weiteren Verlauf dieses Kapitels ausführlich vorgestellt.

XSL-FO-Dokumente sind XML-Dokumente. Daher steht am Anfang immer eine entsprechende XML-Deklaration.

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

Im ersten Schritt wird zunächst eine Transformationsregel definiert, in welcher der Inhalt der Formatierungsobjekte bereitgestellt wird.

Listing 8.5 Das ist der Grundaufbau.

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

In diese Transformationsregel lässt sich nun der Baum der Formatierungsobjekte einfügen. Als erstes Formatierungsobjekt, das als Kind von `xsl:stylesheet` zugelassen ist, kann `fo:root` angegeben werden. Bei diesem Element handelt es sich um die Baumwurzel der Formatierungsobjekte.

Listing 8.6 Ein weiteres Element wurde eingefügt.

```
<xsl:template match="/">
   <fo:root>
      …
   <fo:root>
</xsl:template>
```

Innerhalb der ersten Template-Regel wird also das Wurzelelement für das XSL-FO-Dokument angelegt. Als erstes Element innerhalb dieses Wurzelelements muss das Layout-Masterset stehen.

Listing 8.7 So sieht das Grundlayout aus.

```
<xsl:template match="/">
   <fo:root>
      <fo:layout-master-set>
         …
      </fo:layout-master-set>
   <fo:root>
</xsl:template>
```

In diesem Layout-Masterset ist die Definition aller benötigten Layoutvorlagen enthalten. Dazu gehören u.a. die verschiedenen Seitenvorlagen für Titelseiten sowie linke und rechte Seiten.

Als Nächstes folgt die Beschreibung der eigentlichen Publikation. Für jedes Publikationsteil muss dabei eine eigene Seitenfolge (`fo:page-sequence`) vorhanden sein.

Von den Layoutvorlagen müssen zwei im Layout-Masterset festgelegt werden. Das sind einmal die Seitenvorlagen (`fo:simple-page-master`), durch die jeweils eine einzelne Layoutseite beschrieben wird. Und dann gibt es ggf. noch Seitenvorlagen (`fo:page-`

sequence.master), die mehrere Seitenvorlagen zu einer Folge zusammenfügen. Diese Folge lässt sich als einzelne Vorlage adressieren.

Für eine Publikation können u.U. mehrere Seitenvorlagen, eine oder mehrere Seitenfolgevorlagen und eine aufwendige Publikationsabfolge aus mehreren Seitenfolgen definiert werden. Das hängt letztendlich aber davon ab, wie komplex die Publikation ist.

Listing 8.8 Das Grundgerüst wurde weiter ausgebaut.

```
<xsl:template match="/">
   <fo:root>
      <fo:layout-master-set>
         <fo:simple-page-master…>
            …
         </fo:simple-page-master>
            …
      </fo:layout-master-set>
   <fo:root>
</xsl:template>
```

Nachdem diese allgemeinen Einstellungen vorgenommen wurden, geht es an die Platzverteilung für die einzelnen Regionen.

Listing 8.9 Die Regionen werden angelegt.

```
<xsl:template match="/">
   <fo:root>
      <fo:layout-master-set>
         <fo:simple-page-master
            …>
            <fo:region-before …/>
            <fo:region-body …/>
         </fo:simple-page-master>
        …
      </fo:layout-master-set>
   <fo:root>
</xsl:template>
```

Auf den folgenden Seiten werden die benötigten Elemente nach und nach vorgestellt.

8.2.4 Attributsätze

Sämtliche Eigenschaften für Absatz- und Überschriftentypen werden jeweils mit einzelnen Gestaltungsattributen beschrieben. Wenn die gleichen Gestaltungsattribute innerhalb eines Stylesheets an mehreren Stellen verwendet werden sollen, kann und sollte man Attributsätze definieren. Durch diese Attributsätze lässt sich die gesamte Attributkombination im Dokument beliebig oft verwenden.

Attributsätze werden über `xsl:attribute-set` definiert. In diesem Element werden dann die einzelnen Attribute über `xsl:attribute` angegeben. Hierzu ein Beispiel:

Listing 8.10 Attribute werden definiert.

```
<xsl:attribute-set name="content">
  <xsl:attribute name="font-family">Arial</xsl:attribute>
   <xsl:attribute name="font-size">12pt</xsl:attribute>
   <xsl:attribute name="font-color">red</xsl:attribute>
</xsl:attribute>
```

Um einen auf diese Weise definierten Attributsatz zu verwenden, wird `xsl:use-attribute-set` eingesetzt.

Listing 8.11 Der Attributsatz wird definiert.

```
<xsl:template match="inhalt">
   <fo:block xsl:use-attribute-sets="content">
      ...
   </fo:block>
</xsl:template>
```

Als Wert von `xsl:use-attribute-set` geben Sie den Namen des gewünschten Attributsets an.

Interessant sind Attributsätze vor allem auch im Zusammenhang mit externen Stylesheets. Denn werden Attributsätze ausgelagert, kann man diese leicht durch alternative Dateien ersetzen, in denen anders gestaltete Attributsätze enthalten sind. So können Sie ohne viel Aufwand völlig unterschiedlich gestaltete Dokumente generieren.

8.2.5 Parameter und Variablen

Sie kennen Parameter und Variablen vielleicht bereits aus anderen Sprachen wie PHP oder JavaScript. Beide Elemente stehen so auch im Zusammenhang mit XSLT zur Verfügung und sind natürlich auch bei XSL-FO interessant.

Zunächst stellt sich die Frage, wann Parameter und wann Variablen eingesetzt werden sollten. Parameter kommen immer dann ins Spiel, wenn es sich um einen feststehenden Wert handelt. Gekennzeichnet werden Parameter durch `xsl:param`. Die Definition eines Parameters sieht also folgendermaßen aus:

```
<xsl:param name="raender">10mm 50mm 50mm 30mm</xsl:param>
```

Um den Parameter im Stylesheet zu nutzen, wird die folgende Syntax verwendet.

```
<fo:simple-page-master margin="{$raender}">
```

Prinzipiell – und das gilt für Variablen und Parameter gleichermaßen – muss der Name einschließlich des führenden Dollarzeichens in geschweiften Klammern stehen. Lediglich bei der XSLT-Anweisung `value-of` kann auf die geschweiften Klammern verzichtet werde, da es sich bei den geschweiften Klammern um die Abkürzung von `xsl:value-of` handelt.

Ganz ähnlich sieht es bei der Definition von Variablen aus. Verwendet wird für deren Definition das Element `xsl:variable`.

```
<xsl:variable name="raender">10mm 50mm 50mm 30mm</xsl:variable>
```

Und auch der Zugriff auf die Variable funktioniert wie bei den Parametern.

```
<fo:simple-page-master margin="{$raender}">
```

Normalerweise stellt man global wirkende Variablen und Parameter an den Anfang des Stylesheets, noch vor die erste Template-Regel. So kann man sie ganz gezielt ändern.

8.3 Seitenlayouts festlegen

Auf den folgenden Seiten wird die Definition von XSL-FO Seitenlayouts gezeigt. Zuvor aber noch ein Blick auf die verfügbaren Maßeinheiten. Denn über die werden schlussendlich sämtliche Größenangaben festgelegt.

8.3.1 Maßeinheiten in XSL-FO

Um effektiv mit XSL-FO arbeiten zu können, müssen Sie die einsetzbaren Maßeinheiten kennen. In XSL-FO sind die folgenden verfügbar:

Tabelle 8.2: Mögliche Maßeinheiten in XSL-FO

Maßeinheit	Erklärung
`cm`	Zentimeter
`mm`	Millimeter
`in`	Inch (1 in = 2,54 cm)
`pt`	DTP-Punkt (1 pt = 1/72 in = 0,353 mm)
`pc`	Picas (1 pc = 12 pt)
`px`	Pixel
`%`	Prozent
`em`	1em entspricht der Schrifthöhe

Welche der gezeigten Maßeinheiten vom jeweiligen Formatierer unterstützt werden, lässt sich pauschal nicht sagen. Das muss im Einzelfall getestet werden.

8.3.2 Das Seitenlayout definieren

XSL-FO-Dokumente folgen immer dem gleichen Grundgerüst. Das folgende Beispieldokument zeigt diejenigen Elemente, die ein XSL-FO-Dokument haben muss, damit es von einem Formatierer verarbeitet werden kann.

Listing 8.12 Das Seitenlayout wird angelegt.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">
  <fo:layout-master-set>
    <fo:simple-page-master master-name="buch"
                           page-height="12cm"
                           page-width="10cm">
      <fo:region-body region-name="inhalt"/>
    </fo:simple-page-master>
  </fo:layout-master-set>

  <fo:page-sequence master-reference="buch">
    <fo:flow flow-name="inhalt">
      <fo:block>Hier steht der Inhalt</fo:block>
```

```
    </fo:flow>
  </fo:page-sequence>
</fo:root>
```

Das Wurzelelement heißt `fo:root`, das im aktuellen Beispiel den Namensraum enthält. Die Namensraum-Deklaration erscheint normalerweise zu Beginn des XSL-Stylesheets, welches das XSL-FO-Dokument erzeugt, im `stylesheet`-Element. Wenn Sie mit einem entsprechenden Formatierer arbeiten, können Sie auch – zumindest dort, wo das möglich ist – den Namensraum des Formatierers verwenden. Auf diese Weise lassen sich proprietäre Erweiterungen des jeweiligen Formatierers nutzen.

Im Element `fo:layout-master-set` werden die Vorlagen für die Seitengestaltung erzeugt. Darin legt man die Seitenreihenfolge fest und definiert die Ränder und Regionen der Seiten. In einem XSL-FO-Dokument gibt es immer genau ein `fo:layout-master-set`. Dieses Element dient als Container für eine Anzahl von Elementen, über die das Grundlayout erzeugt wird.

Das Aussehen einer einzelnen Seite wird über `fo:simple-page-master` gesteuert. Mit `fo:simple-page-master` definiert man eine einfache Musterseite.

Listing 8.13 Das Seitenlayout wird definiert.

```
<fo:simple-page-master master-name="buch"
   page-height="10cm"
   page-width="10cm"
   margin-top="0.5cm"
   margin-bottom="0.5cm"
   margin-left="1cm"
   margin-right="0.5cm">
        <fo:region-body margin-top="3cm" />
</fo:simple-page-master>
```

Bei der Definition der tatsächlichen Maße für die Seitenvorlage muss Folgendes beachtet werden: XSL-FO geht davon aus, dass auf ein Blatt gedruckt wird und nicht von der Rolle.

Für die Definition von Seiten werden in XSL-FO die folgenden Angaben erwartet:

- Seitenbreite
- Seitenhöhe
- Ränder

Die Randangabe untereilt sich wiederum in vier Bereiche:

- Rand oben
- Rand unten
- Rand rechts
- Rand links

Im Abzugsverfahren wird Folgendes festgelegt:

- Die Seitengröße wird über Maßangaben für alle vier Seiten bestimmt.
- Für den Bereich, in den der Inhalt des zu verarbeitenden XML-Dokuments einfließen soll, wird die Ausdehnung für die vier äußeren Seitenbereiche abgezogen.

Das Blatt muss also folgendermaßen aufgeteilt werden:

- Ein Randbereich, der nicht bedruckt werden darf und ggf. weggeschnitten wird.
- Fünf Seitenbereiche, in die gedruckt werden kann.

Wie die Seitengröße und die Ränder definiert werden, zeigt noch einmal folgendes Beispiel:

Listing 8.14 Die Seitendefinition wird festgelegt.

```
<fo:simple-page-master master-name="buch"
   page-height="297mm"
   page-width="210mm"
   margin-left="12mm"
   margin-right="12mm"
   margin-top="20mm"
   margin-bottom="20mm">
```

Wobei auch hier die Nähe zu CSS deutlich wird. So definiert man mit `margin-top` den Randabstand nach oben, während `margin-bottom` den Randabstand nach unten festlegt.

In den bisherigen Beispielen wurden lediglich einzelne Seiten definiert. Üblicherweise werden Dokumente mit einem Titelblatt versehen. Zudem wird zwischen linken und rechten Seiten unterschieden.

Wie sich so etwas umsetzen lässt, zeigt das folgende Beispiel:

Listing 8.15 Verschiedene Seiten wurden angelegt.

```
 <fo:layout-master-set>
   <fo:simple-page-master master-name="titel"
        page-height="10cm"
        page-width="10cm"
        margin-top="0.5cm"
        margin-bottom="0.5cm"
        margin-left="1cm"
        margin-right="0.5cm">
        <fo:region-body
        margin-top="3cm" />
    </fo:simple-page-master>

    <fo:simple-page-master master-name="leftPage"
        page-height="10cm"
        page-width="10cm"
        margin-left="3cm"
        margin-right="1.5cm"
        margin-top="0.5cm"
        margin-bottom="0.5cm">

        <fo:region-before extent="1cm"/>
            <fo:region-after extent="1cm"/>
            <fo:region-body
                margin-top="1.1cm"
                margin-bottom="1.1cm" />
    </fo:simple-page-master>

    <fo:simple-page-master master-name="rightPage"
        page-height="10cm"
        page-width="10cm"
        margin-left="1.5cm"
        margin-right="3cm"
        margin-top="0.5cm"
        margin-bottom="0.5cm">
```

```
        <fo:region-before extent="1cm"/>
            <fo:region-after extent="1cm"/>
            <fo:region-body
                margin-top="1.1cm"
                margin-bottom="1.1cm" />
    </fo:simple-page-master>
</fo:layout-master-set>

...

</fo:root>
```

In diesem Beispiel werden verschiedene `simple-page-master` angelegt. Diese unterscheiden sich zunächst einmal durch den jeweiligen Namen. Auf diese Weise lassen sich dann verschiedene Seiten definieren.

8.3.3 Seitenfolgen-Vorlagen definieren

Das Element `fo:simple-page-master` haben Sie bereits kennengelernt. In diesem Abschnitt geht es nun um das `fo:page-sequence-master`-Element. Bei dem handelt es sich um das zweite Kindelement von `fo:layout-master-set`. Über `fo:page-sequence-master` legt man fest, in welcher Reihenfolge die Seitengestaltungen oder Sequenzen von Seitengestaltungen in das betreffende Dokument eingefügt werden sollen.

Bei den genannten Seitenfolgen handelt es sich um eine Ansammlung von einzelnen Seiten oder ganzen Sequenzen. Um Missverständnisse zu vermeiden: Wo die Seiten letztendlich im Dokument erscheinen, wird hierüber nicht bestimmt. Dafür ist das Element `fo:page-sequence` zuständig.

Seitenfolgen werden immer dann benötigt, wenn ein inhaltlich einzugrenzender Teil des XML-Dokuments mehrere Seiten umfasst. Typischerweise handelt es sich dabei um Kapitel, für die verschiedene Seitenfolgen verwendet werden sollen. Das könnte z.B. in den folgenden Fällen sein:

- Die erste Seite des Kapitels wird als rechte Seite ohne Pagina definiert.
- Die zweite und jede andere Seite als linke Seite mit Pagina.
- Die dritte und jede andere Seite als rechte Seite mit Pagina.

Um unterschiedliche Seitenvorlagen zu einer Seitenvorlage zu kombinieren, müssen die einzelnen Seitenvorlagen mit einem eindeutigen Namen versehen werden. Dieser Name wird dem Attribut `master-name` des Elements `fo:page-sequence-master` zugewiesen. Hier ein typisches Beispiel für eine solche Definition:

Listing 8.16 So werden die Seiten zusammengefasst.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">
  <fo:layout-master-set>
    <fo:page-sequence-master master-name="contents">
      <fo:repeatable-page-master-alternatives>
           <fo:conditional-page-master-reference
               master-reference="leftPage"
               odd-or-even="even"/>
           <fo:conditional-page-master-reference
               master-reference="rightPage"
```

```
                odd-or-even="odd"/>
          </fo:repeatable-page-master-alternatives>
       </fo:page-sequence-master>
    </fo:layout-master-set>
</fo:root>
```

Mit dem Element vom Typ `page-sequence-master` definiert man eine Seitenfolge. Über dieses Element wird angegeben, welche Ordnung die Seiten in der Reihenfolge innerhalb des Dokuments einhalten sollen. Für diese Ordnung gibt es drei Attribute:

- `single-page-master-reference` – Die Seite kommt lediglich einmal innerhalb eines über `layout-master-set` definierten Grundlayouts vor. Interessant ist das z.B. für den Titel oder das Impressum.
- `repeatable-page-master-reference` – Die Seite kann beliebig oft mit verschiedenen Inhalten in der Ausgabe erscheinen. Dabei wird immer das gleiche Seitenlayout verwendet.
- `repeatable-page-master-alternatives` – Die Seitentypen wechseln sich regelmäßig ab. Typischerweise wird das für linke und rechte Buchseiten verwendet, die unterschiedlich sind.

Interessant ist auch das Element `fo:conditional-page-master-reference`. Durch das lassen sich Bedingungen definieren. Typische Bedingungen sind, ob es sich um eine linke oder rechte Seite handelt. Einige in der Praxis oft benötigte Anwendungen folgen im weiteren Verlauf dieses Abschnitts. Zunächst aber ein Blick auf die möglichen Attribute bzw. Bedingungen dieses Elements.

- `odd-or-even` – Bestimmt, ob es sich um eine gerade oder ungerade Seitennummer handelt. Mögliche Werte sind `odd` (ungerade), `even` (gerade), `any` (initial) oder `inherit` (Standard).
- `blank-or-not-blank` – Damit gibt man an, ob es mich um eine leere (`blank`) oder eine nicht leere (`not-blank`) Seite handelt.
- `page-position` – Hierüber wird die Position der Seite innerhalb der Seitenfolge bestimmt. Mögliche Werte sind `first` (erste Seite), `last` (letzte Seite), `rest` (weder erste noch letzte Seite eines Seitenverlaufs) und `any` (irgendeine Seite des Seitenverlaufs).

Die Werte lassen sich dabei nicht nur einzeln aufrufen, sondern auch untereinander kombinieren. Dadurch werden sechs verschiedene Varianten möglich, was ein gehöriges Maß an Flexibilität erlaubt.

Listing 8.17 Die verschiedenen Elemente in Aktion

```
<fo:page-sequence-master master-name="Mein-Buch">
<fo:repeatable-page-master-alternatives>

<fo:conditional-page-master-reference master-
reference="PageMaster.content_left" page-position="first"
odd-or-even="odd"/>

<fo:conditional-page-master-reference master-
reference="PageMaster.content_left" page-position="rest"
odd-or-even="even"/>
```

```
<fo:conditional-page-master-reference master-
reference="PageMaster.content_right" page-position="rest"
odd-or-even="odd"/>

<fo:conditional-page-master-reference master-reference="PageMaster.left"
page-position="last"
odd-or-even="even" blank-or-not-blank="blank"/>

<fo:conditional-page-master-reference master-
reference="PageMaster.content_left" page-position="last"
odd-or-even="even"/>

<fo:conditional-page-master-reference master-
reference="PageMaster.content_right" page-position="last"
odd-or-even="odd"/>

</fo:repeatable-page-master-alternatives>

<fo:single-page-master-reference master-reference="PageMaster.refer"/>

</fo:page-sequence-master>
```

Durch diese Syntax wurde eine einfache Seitenfolge-Vorlage definiert. Als Name der Seitenfolge wurde in diesem Beispiel `Mein-Buch` gewählt. Daran schließen sich einige `fo:conditional-page-master-reference`-Bedingungen an, über die Alternativen bestimmt werden. Dabei wird über das `master-reference`-Attribut jeweils die Seitendefinition bzw. die Seitenfolge angegeben, die eingesetzt werden soll, wenn es sich um eine ganz bestimmte Seite (die erste, die letzte usw.) handelt. Über

```
odd-or-even="odd"
```

wird festgelegt, dass die erste Seite immer auf einer Seite mit ungerader Seitenzahl stehen soll. Wenn der Inhalt einer vorhergehenden Seite auf einer ungeraden Seite endet, wird automatisch eine leere Seite eingefügt. Dadurch beginnt die neue Seite immer auf einer Seite mit ungerader Seitenzahl.

In der nächsten `fo:conditional-page-master-reference`-Bedingung wird über `page-position="rest"` das Aussehen der übrigen Seiten festgelegt.

Und dann wird noch das Aussehen einer geraden, letzten und leeren Seite bestimmt.

```
page-position="last" odd-or-even="even" blank-or-not-blank="blank"
```

Auf die gezeigte Weise können Sie auf jede auftretende Bedingung reagieren.

8.3.4 Die Seitenfolge festlegen

Anhand des zuvor Beschriebenen ist deutlich geworfen, dass mit dem Element `fo:page-sequence-master` die Seiten und Seitenvorlagen definiert werden. Um nun aber auch noch festlegen zu können, wie diese Seitenfolgen dann letztendlich in der XML-Instanz aufgerufen werden, muss man das Element `fo:page-sequenz` einsetzen. Dieses Element kennt die folgenden Attribute:

- `id` – Der Seitenfolge wird eine ID zugewiesen.
- `master-name` – Damit gibt man den Namen der Seitenfolge an.
- `country` – Hierüber wird das Land angegeben, in dem das Dokument verfasst wurde.
- `language` – Gibt die Sprache des Dokumentinhalts an.

- `initial-page-number` – Hierüber bestimmt man den Startwert für die Seitennummerierung.
- `force-page-count` – Damit wird bestimmt, ob die Seitenfolge mit einer geraden oder ungeraden Seite enden oder anfangen soll.
- `format` – Damit legt man das Format für die Seitennummerierung fest.
- `grouping-separator` – Das Trennzeichen wird über dieses Attribut bestimmt.
- `grouping-size` – Damit wird die Länge festgelegt, nach der ein Trennzeichen eingefügt werden soll.
- `letter-value` – Bei alphanumerischer Zählung lässt sich darüber die Formatierung der Seitennummern bestimmen.

Das Element `fo:page-sequence` sorgt dafür, dass den innerhalb von `fo:simple-page-master` definierten Bereichen die entsprechenden Inhalte zugewiesen werden. Jedes `fo:page-sequence`-Element beschreibt eine konkrete und einzelne Seite, deren Eigenschaften teilweise durch den Page-Master vorgegeben sind.

Das folgende Beispiel zeigt, wie sich eine Zeichenfolge typischerweise festlegen lässt.

Listing 8.18 Eine Seitenfolge wird definiert.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">
 <fo:layout-master-set>
  <fo:simple-page-master margin-bottom="0.3cm" margin-left="1cm"
         margin-right="0.5cm" margin-top="0.3cm"
         master-name="Mein-Buch"
         page-height="12cm" page-width="12cm">
        <fo:region-body margin-top="3cm"/>
  </fo:simple-page-master>

  <fo:simple-page-master master-name="left"
        page-height="21cm"
        page-width="15cm"
        margin-left="3cm"
        margin-right="1.5cm"
        margin-top="0.5cm"
        margin-bottom="0.5cm">
      <fo:region-before extent="1cm"/>
      <fo:region-after extent="1cm"/>
      <fo:region-body margin-bottom="1.1cm" margin-top="1cm"/>
  </fo:simple-page-master>

  <fo:simple-page-master master-name="right"
        page-height="21cm"
        page-width="15cm"
        margin-left="1.5cm"
        margin-right="3cm"
        margin-top="0.5cm"
        margin-bottom="0.5cm">
      <fo:region-before region-name="header" extent="1cm"/>
      <fo:region-body margin-bottom="1.1cm" margin-top="1cm"/>
      <fo:region-after extent="1cm"/>
  </fo:simple-page-master>

  <fo:page-sequence-master master-name="content">
      <fo:repeatable-page-master-alternatives>
           <fo:conditional-page-master-reference
            master-reference="left"
            even="even"/>
           <fo:conditional-page-master-referenc
```

```
            master-reference="rightPage"
             odd-or-even="odd"/>
      </fo:repeatable-page-master-alternatives>
 </fo:page-sequence-master>
</fo:layout-master-set>

<fo:page-sequence master-reference="Mein Buch">
  <fo:flow flow-name="xsl-region-body">
      <fo:block font-family="Helvetica" font-size="18pt"
          text-align="end">Hallo, Welt!</fo:block>
  </fo:flow>
</fo:page-sequence>

<fo:page-sequence master-reference="contents"
initial-page-number="2">
 <fo:static-content flow-name="xsl-region-before">
      <fo:block font-family="Helvetica" font-size="10pt"
         text-align="center">
         Die zweite Seite
      </fo:block>
 </fo:static-content>

 <fo:static-content flow-name="xsl-region-after">
      <fo:block font-family="Helvetica" font-size="10pt"
         text-align="center">
         Seite <fo:page-number />
      </fo:block>
 </fo:static-content>

 <fo:flow flow-name="xsl-region-body">
     <fo:block font-size="10pt">
      <!-- Hier folgt das Buch -->
     </fo:block>
 </fo:flow>
</fo:page-sequence>
</fo:root>
```

Über das Element `fo:simple-page-master` werden allgemeine Angaben zu den Seiten gemacht. Dazu gehören Breite, Höhe, Randabstände usw. Die verwendeten Attribute erinnern auch hier wieder an CSS.

In dem gezeigten Beispiel wird über `fo:page-sequence` eine Seitenfolge erzeugt. Über das Attribut `master-reference` stellt man einen Bezug zu einer Musterseite bzw. einer Folge von Musterseiten her.

Durch die `fo:flow`-Elemente wird erreicht, dass die Inhalte fließen und ggf. auf neue Seiten umbrochen werden, wenn die Größe der Seite für den angegebenen Inhalt nicht ausreicht. Über das `fo:flow`-Elements werden sowohl Zeilen- als auch Seitenumbruch und somit letztendlich natürlich auch die Seitenanzahl bestimmt.

Für die Definition statischer Inhalte wird `fo:static-content` verwendet. Durch dieses Element werden die statischen Inhalte von Seitenbereichen außerhalb des Bereichs bestimmt, in den der fließende Inhalt geschrieben wird.

Listing 8.19 Hier wird statischer Inhalt verwendet.

```
<fo:page-sequence master-reference="FirstPage">

<fo:static-content flow-name="kopf">
   <fo:block>Kopfzeile</fo:block>
</fo:static-content>
```

```
<fo:static-content flow-name="fuss">
   <fo:block>Fußzeile</fo:block>
</fo:static-content>

<fo:static-content flow-name="xsl-region-start">
   fo:block>Region-Start</fo:block>
</fo:static-content>

<fo:static-content flow-name="xsl-region-end">
   <fo:block>Region-End</fo:block>
</fo:static-content>

<fo:flow flow-name="xsl-region-body">
   <fo:block>Body-Bereich</fo:block>
</fo:flow>

</fo:page-sequence>
```

Statischer Inhalt ist nicht für Seiten- und Zeilenumbrüche verantwortlich. Ist also beispielsweise der statische Inhalt größer als die Seite, wird nicht automatisch ein Seitenumbruch generiert. Stattdessen gibt der Formatierer einen Warnhinweis aus oder bricht mit einer Fehlermeldung ab.

Elemente von Typ `fo:static-content` sind optional.

Die Elemente `fo:flow` und `fo:static-content` können als Attribut `flow-name` besitzen. Der als Wert angegebene Name verweist auf das Attribut `region-name` innerhalb eines `fo:simple-page-master`-Elements.

Innerhalb eines `fo-page-sequence`-Elements darf es nur ein `fo:flow`-Element geben. Stehen muss dieses hinter allen vorhandenen `fo:static-content`-Elementen.

8.3.4.1 Typische Anwendungsszenarien

Auf den vorherigen Seiten wurde der Umgang mit Seitenvorlagen und Seitenfolgen beschrieben. Nun muss man üblicherweise sehr oft ähnliche dieser Abfolgen definieren. Hier einige typische Szenarien:

- Die erste Seite soll anders aussehen.
- Es ist ein Seitenwechsel vorgesehen.
- Es werden unterschiedliche Ränder für linke und rechte Seiten benötigt.

In diesem Abschnitt finden Sie zu den genannten Aspekten jeweils ein Beispiel, das Sie als Basis für Ihre eigenen Stylesheets verwenden können.

Zunächst der Fall, bei dem die erste Seite etwas anders aussehen soll.

Listing 8.20 Die erste Seite soll anders aussehen.

```
<fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">
  <fo:layout-master-set>
    <fo:simple-page-master master-name="start">
      <fo:region-body margin="3cm" border-width="1pt"
      padding="6pt" />
    </fo:simple-page-master>

    <fo:simple-page-master master-name="content">
      <fo:region-body margin="3cm" />
    </fo:simple-page-master>
```

```
    <fo:page-sequence-master master-name="buch">
      <fo:single-page-master-reference master-reference="start" />

      <fo:repeatable-page-master-reference
      master-reference="content" />

    </fo:page-sequence-master>

  </fo:layout-master-set>

  <fo:page-sequence master-reference="buch">

    ...

  </fo:page-sequence>

</fo:root>
```

Eine solche Anwendung ist immer dann sinnvoll, wenn in einem laufenden Text die erste Seite anders aussehen soll.

Weiter geht es mit einem Beispiel, in dem explizit unterschiedliche linke und rechte Seiten definiert werden.

Listing 8.21 Hier wurden unterschiedliche Seitenränder angelegt.

```
<fo:simple-page-master master-name="links">
  <fo:region-body margin="2cm 3cm 2cm 2cm" />
</fo:simple-page-master>

<fo:simple-page-master master-name="rechts">
  <fo:region-body margin="2cm 4cm 2cm 2.5cm" />
</fo:simple-page-master>

<fo:page-sequence-master master-name="buch">
  <fo:repeatable-page-master-alternatives>
    <fo:conditional-page-master-reference odd-or-even="even"
    master-reference="links" />

    <fo:conditional-page-master-reference odd-or-even="odd"
     master-reference="rechts" />
  </fo:repeatable-page-master-alternatives>
</fo:page-sequence-master>
```

Im folgenden Beispiel wird explizit eine Titelseite vorgesehen.

Listing 8.22 Eine Titelseite wurde angelegt.

```
<fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">
  <fo:layout-master-set>

    <fo:simple-page-master master-name="titel">
      <fo:region-body margin="3cm" />
    </fo:simple-page-master>

    <fo:simple-page-master master-name="text">
      <fo:region-body margin="2cm" />
    </fo:simple-page-master>

    <fo:page-sequence-master master-name="buch">
      <fo:repeatable-page-master-reference
      master-reference="text" />
    </fo:page-sequence-master>
  </fo:layout-master-set>

  <fo:page-sequence master-reference="titel">
    <!-- Titelseite -->
  </fo:page-sequence>
```

```
    <fo:page-sequence master-reference="buch">
      <!-- Inhalt -->
    </fo:page-sequence>

</fo:root>
```

Sie haben anhand einfacher Beispiele gesehen, wie sich grundlegende Layouts bzw. Anforderungen definieren lassen. Im weiteren Verlauf dieses Kapitels gibt es noch viel mehr solcher Anwendungen.

8.3.5 Druckbereiche festlegen

Nachdem das Grundlayout der Seite(n) definiert wurde, können die Druckbereiche angegeben werden.

- Kopfbereich = `fo:region-before`
- Fußbereich = `fo:region-after`
- Linker Bereich = `fo:region-start`
- Rechter Bereich = `fo:region-end`
- Hauptbereich = `fo:region-body`

Bei allen Bereichen handelt es sich jeweils um Kindelemente von `fo:simple-page-master`.

Listing 8.23 Die Druckbereiche werden definiert.

```
<fo:simple-page-master-name="welt">
   <fo:region-before extent="20mm"/>
   <fo:region-after extent="20mm"/>
   <fo:region-body margin-top="5mm" margin-bottom="5mm"/>
</fo:simple-page-master>
```

Über das `extent`-Attribut gibt man jeweils die Ausdehnung des betroffenen Bereichs an. Bei Kopf- und Fußzeilen ist das die Höhe. Bei linken und rechten Bereichen wird hingegen die Breite bestimmt.

Und jetzt noch ein etwas ausführlicheres Beispiel, das die Definition der Druckbereiche veranschaulicht.

Listing 8.24 Auch hier wird der Druckbereich festgelegt.

```
<fo:layout-master-set>
    <fo:simple-page-master master-name="seite">
      <fo:region-body margin="1.2cm 1cm"/>
      <fo:region-before extent="2cm"
                        padding="6pt 1in"
                        border-bottom="0.5pt silver solid"
                        display-align="after"/>
    </fo:simple-page-master>
  </fo:layout-master-set>

  <fo:page-sequence master-reference="seite">
    <fo:static-content flow-name="xsl-region-before"
                       font="italic 10pt Times">
       <fo:block>Hallo, Welt </fo:block>
    </fo:static-content>
```

```
    <fo:flow flow-name="xsl-region-body">
       <fo:block>Hello, world!</fo:block>
    </fo:flow>

  </fo:page-sequence>
```

8.3.6 Mit Blöcken arbeiten

In den vorherigen Beispielen wurde bereits mit Blöcken gearbeitet. Das Konzept der Blöcke ist mit dem Box-Modell aus HTML und CSS identisch. Blöcke werden für die Platzierung beliebiger Elemente verwendet. So kann man z.B. in einen Block Grafiken, Text und Tabellen einfügen. Diese Blöcke wiederum lassen sich dann ineinander verschachteln.

Bei der Anwendung von Block-Containern gilt es, einige Aspekte zu berücksichtigen.

- Die Platzierungskoordinaten innerhalb eines Seitenbereichs oder einer Seite.
- Breite und Höhe des Block-Containers.
- Die absolute Positionierung in der Seite bzw. den Seitenbereichen, in denen der Block-Container zur Formatierung aufgerufen wird.
- Wenn sich die Block-Container überlagern, muss ihre Stapelungsebene berücksichtigt werden.

Für die Definition von Blöcken wird das `fo:block`-Element verwendet. Innerhalb dieser Blockdefinitionen lassen sich dann die verschiedensten Eigenschaften definieren. Hier ein paar typische Beispiele:

- Schriftart
- Hintergrundfarbe
- Farbe
- Rahmen
- Ränder
- Seiten- und Zeilenumbruch

Abbildung 8.9 zeigt, wie vielfältig sich in XSL-FO Inhalte von Blockelementen durch die entsprechenden Attribute und deren Werte gestalten lassen.

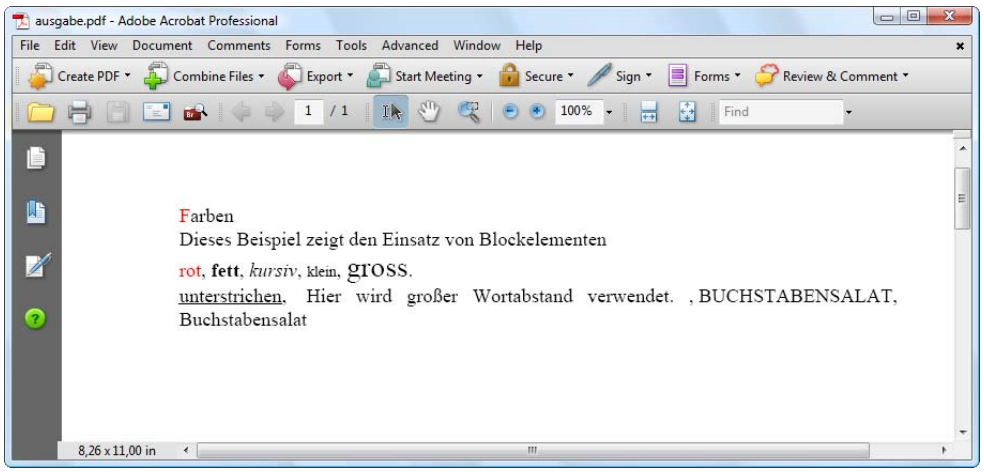

Abbildung 8.9 Verschiedene Blockelemente wurden eingesetzt.

Die in diesem Beispiel verwendeten Attribute werden im weiteren Verlauf dieses Kapitels natürlich noch ausführlich vorgestellt. Dennoch vermittelt die Syntax bereits jetzt einen ersten Eindruck davon, wie sich Blöcke definieren und ihnen Eigenschaften zuweisen lassen.

Listing 8.25 Hier wurden bereits einige Attribute verwendet.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">
  <fo:layout-master-set>
    <fo:simple-page-master master-name="seite">
      <fo:region-body margin="1in"/>
    </fo:simple-page-master>
  </fo:layout-master-set>

  <fo:page-sequence master-reference="seite">
     <fo:flow flow-name="xsl-region-body" font="12pt Times">
       <fo:block font="italic 24pt Helvetica">
         <fo:inline color="red">F</fo:inline>arben
       </fo:block>

       <fo:block>
         Dieses Beispiel zeigt den Einsatz von Blockelementen
       </fo:block>

       <fo:block>
         <fo:inline color="red">rot</fo:inline>,
         <fo:inline font-weight="bold">fett</fo:inline>
         <fo:inline font-style="italic">kursiv</fo:inline>,
         <fo:inline font-size="75%">klein</fo:inline>,
         <fo:inline font-size="150%">gross</fo:inline>.
       </fo:block>

      <fo:block>
         <fo:inline text-decoration="underline">
            unterstrichen
         </fo:inline>,

         <fo:inline word-spacing="6pt">
            Hier wird großer Wortabstand verwendet.
         </fo:inline>,
```

```
            <fo:inline text-transform="uppercase">
               buchstabensalat
            </fo:inline>,

            <fo:inline text-transform="capitalize">
               Buchstabensalat
            </fo:inline>

         </fo:block>

        </fo:flow>

      </fo:page-sequence>

   </fo:root>
```

Im weiteren Verlauf dieses Kapitels werden Sie noch die verschiedenen Möglichkeiten für die Definition von Blöcken kennenlernen. Wichtig ist jetzt erst einmal zu verinnerlichen, dass durch Blöcke Bereiche innerhalb der Seite definiert werden, in denen sich Elemente anordnen lassen.

8.3.7 Mit Inline-Elementen arbeiten

Wie sich Blöcke erzeugen lassen, wurde auf den vorherigen Seiten gezeigt. Neben dieser Blockbildung gibt es aber auch noch die Möglichkeit zur inzeiligen Definition. Besonders interessant sind dabei die folgenden Varianten:

- `fo:inline` – inzeilige Hervorhebung
- `fo:character` – Gestaltung eines einzelnen Zeichens
- `fo:inline-container` – Container für Blockelemente

Die vorgestellten Elemente verdienen noch eine genauere Betrachtung.

Durch das Element `fo:inline` können Sie inzeilige Formatierungen definieren. Da die meisten Eigenschaften vererbbar sind, kann man diese kombinieren.

Listing 8.26 Zwei Inline-Elemente werden definiert.

```
<fo:block font-family="Times" font-size="14pt" font-style="italic">
   <fo:inline color="red">H</fo:inline>allo,
   <fo:inline font-weight="bold">Welt!</fo:inline>
</fo:block>
```

Und hier das Ergebnisdokument:

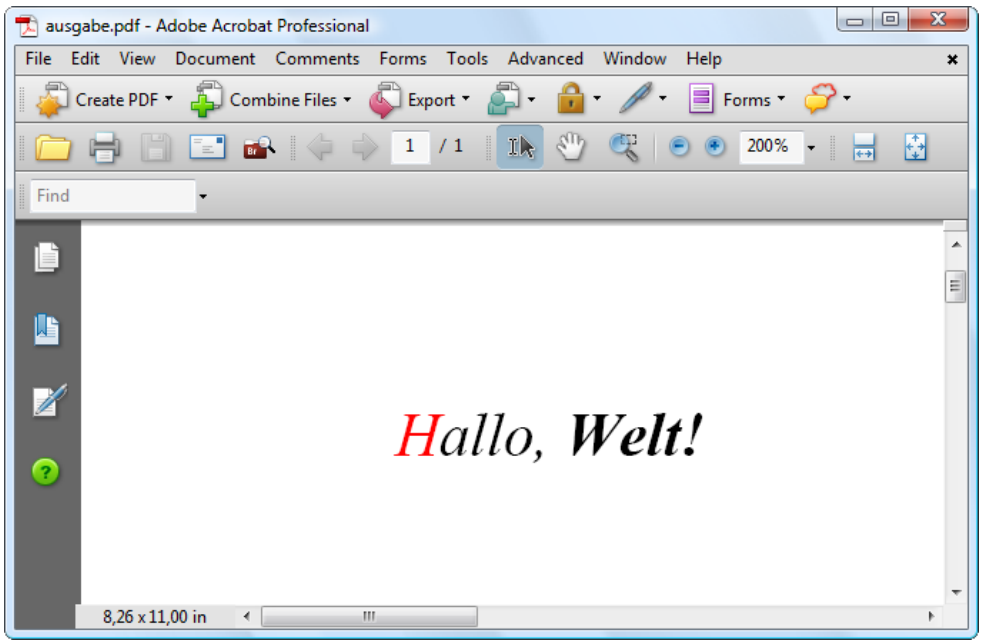

Abbildung 8.10 Einzelne Buchstaben lassen sich anders gestalten.

Mit `fo:inline` können Sie also innerhalb einer Zeile Formatierungen vornehmen, ohne dabei eigene Blöcke zu bilden.

Das Element `fo:character` unterscheidet sich von `fo:inline` lediglich dadurch, dass durch `character` ein einzelnes Zeichen für die Darstellung definiert werden kann. Somit eignet sich `fo:character` beispielsweise zur Gestaltung von Listen.

Listing 8.27 Auch für Listen ist das interessant.

```
<fo:list-item-label end-indent="label-end()">
   <fo:block font-weight="bold">
      <fo:character character="&#x2022;" />
   </fo:block>
</fo:list-item-label>
```

Inline-Container gibt es ebenfalls. Erzeugt werden diese über das Element `fo:inline-container`. Diese Inline-Container werden relativ zur Zeile, in der sie definiert wurden, platziert. Auch für diese Container lassen sich Höhe und Breite bestimmen. Eingesetzt wird diese Container-Art z.B. für Einträge in Wörterbüchern o.Ä.

Listing 8.28 Ein Inline-Container wird angelegt.

```
<fo:block>
  <fo:inline-container>
    <fo:block>
     Das ist ein Inline-Container
    </fo:block>
  </fo:inline-container>
</fo:block>
```

Es gibt also Block- und Inline-Container. Wie sich diese mit den passenden Inhalten füllen und gestalten lassen, dazu dann auf den folgenden Seiten mehr.

8.3.8 Schreibrichtung bestimmen

Die Schreibrichtung ist vor allem dann interessant, wenn Sie Texte verwenden, die eine andere Laufrichtung als von links nach rechts verwenden. Für die Definition der Schreibrichtung kann auf zwei unterschiedliche Elemente zurückgegriffen werden. Einmal wäre da `fo:simple-page-master`. Dadurch wird die Schreibrichtung für ganze Seiten bestimmt. Ebenso lässt sich die Schreibrichtung aber auch einzelnen Textblöcken und sogar im inzeiligen Text definieren. Das Attribut ist bei allen Varianten gleich, nämlich `writing-mode`. Dieses Attribut besitzt drei mögliche Werte.

- `lr-tb`(`leftright-topbottom`) – Von links nach rechts und von oben nach unten.
- `rl-tb`(`rightleft-topbottom`) – Von rechts nach links und von oben nach unten.
- `tb-rl`(`topbottom-rightleft`) – Von oben nach unten und von rechts nach links.

Die erste Buchstabenfolge gibt also immer die Schreibrichtung innerhalb von Zeilen vor, die zweite bestimmt die Richtung der Aneinanderreihung für die Textblöcke.

Listing 8.29 So wird die Schreibrichtung bestimmt.

```
<fo:layout-master-set>
   <fo:simple-page-master
      master-name="ErsteSeite"
      page-height="210mm"
      page-width="297mm"
      margin="15mm"
      margin-right="30mm"
      writing-mode="lr-tb"
      reference-orientation="90">
         <fo:region-body/>
   </fo:simple-page-master>
</fo:layout-master-set>
```

Die Schreibrichtung zusätzlich lässt sich über das Attribut `reference-orientation` beeinflussen. Mehr dazu im nächsten Abschnitt.

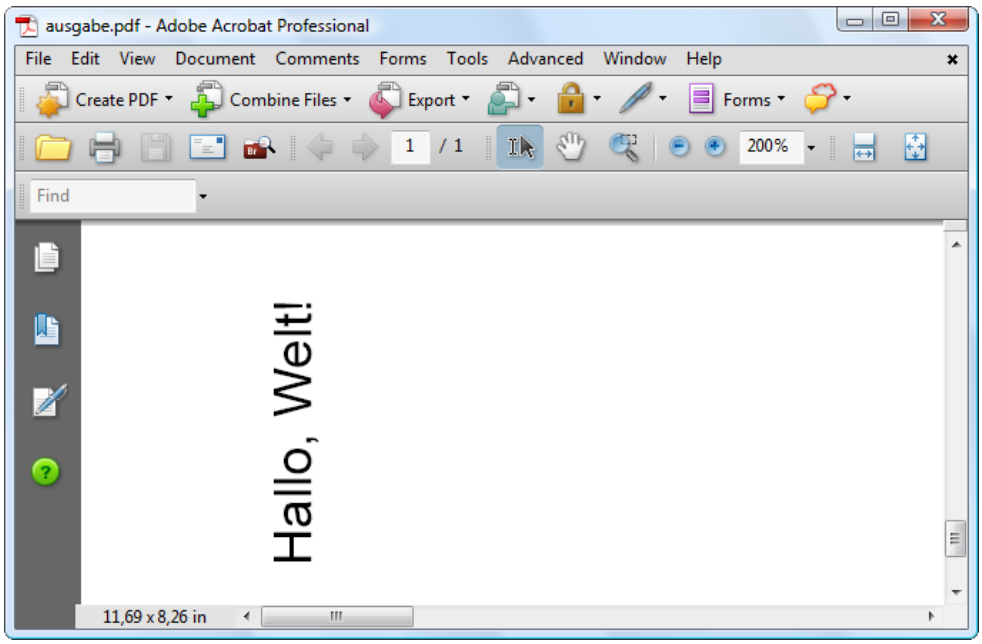

Abbildung 8.11 Der Schriftzug wurde gedreht.

8.3.9 Die Ausrichtung festlegen

Über das Attribut `reference-orientation` lässt sich die Ausrichtung von Elementen bestimmen. Damit ist nicht die normale vertikale oder horizontale Ausrichtung gemeint. Vielmehr geht es darum, Elemente zu drehen. `reference-orientation` kennt insgesamt sieben mögliche Werte.

- 0
- 90
- 180
- 270
- -90
- -180
- -270

Diese Werte sind als Gradangaben zu verstehen.

Listing 8.30 Ein Bereich wird gedreht.

```
<fo:region-start extent="1.5cm"
   region-name="linkeseite"
   reference-orientation="90"
   padding="6pt 1in"
   border-right="0.5pt silver solid"
   display-align="after"/>
```

Das Drehen von Bereichen ist ausschließlich innerhalb des Container-Elements `fo:block-container` erlaubt.

8.3.10 Das Farbmanagement in XSL-FO

Farben können in XSL-FO auf ganz verschiedene Art und Weise eingesetzt werden. Dabei geht es in diesem Abschnitt allerdings nicht um Eigenschaften, mit denen man Elementen Farben zuweisen kann. Vielmehr wird gezeigt, welche Farbmodelle genutzt werden können. Denn während die einen vielleicht eher RGB-Farben nutzen, wollen die anderen lieber auf Farbnamen setzen.

8.3.10.1 Farbnamen

Die einfachste Variante für die Zuweisung von Farben sind sicherlich Farbnamen.[2] Genauso wie in CSS stehen hierfür standardmäßig 16 Farbwörter zur Verfügung.

- `aqua`
- `black`
- `blue`
- `fuchsia`
- `gray`
- `green`
- `lime`
- `maroon`
- `navy`
- `olive`
- `purple`
- `red`
- `silver`
- `teal`
- `white`
- `yellow`

Einsetzen lassen sich Farbnamen folgendermaßen:

```
<fo:block color="red">Roter Text</fo:block>
```

[2] Das ist sicherlich eine subjektive Einschätzung. Dennoch zeigt die Erfahrung, dass sich Farbnamen wie `gray` besser merken lassen.

8.3.10.2 RGB-Farben

Eine weitere Variante zur Farbdefinition sind RGB-Farben. Hier können Sie eine hexadezimale Schreibweise nach folgendem Schema verwenden:

```
#RRGGBB
```

Dabei steht `R` für Rot, `G` für Grün und `B` für Blau.

```
<fo:block color="#FF0000">Roter Text</fo:block>
<fo:block color="#F00">Roter Text</fo:block>
```

Eine andere Variante bietet das Schema `rgb(rrr,ggg,bbb)`.

```
<fo:block color="rgb(255,0,0)">Roter Text</fo:block>
<fo:block color="rgb(100%,100%,100%)">Roter Text</fo:block>
```

Für alle drei Werte innerhalb der Klammern sind entweder absolute Zahlen zwischen `0` (kein Farbanteil) und `255` (maximaler Farbanteil) oder Prozentzeichen erlaubt.

8.3.10.3 ICC-Profile

Der Einsatz von ICC-Profilen ist ebenfalls möglich. Bei einem solchen ICC-Profil handelt es sich um einen genormten Datensatz, durch den der Farbraum eines Eingabe- oder Ausgabegeräts beschrieben wird.

Um in XSL-FO einen Farbraum zu spezifizieren, wird das Element `fo:color-profile` verwendet. Die oder das zu verwendende Farbprofil gibt man innerhalb des `fo:declaration`-Elements an.

Listing 8.31 Die Profile werden eingebunden.

```
<fo:declarations>
   <fo:color-profile
   color-profile-name="RGBColorProfile"   src="AppleRGB.icc"/>
   <fo:color-profile
   color-profile-name="CMYKColorProfile" src="AppleCMYK.icc"/>
</fo:declarations>
```

Dem `src`-Attribut werden dabei Pfad und Name des Profils zugewiesen. Nachdem diese Definition abgeschlossen ist, kann das angegebene Farbprofil eingesetzt werden.

Listing 8.32 So werden Farbprofile verwendet.

```
<fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">
    <fo:declarations>
        <fo:color-profile
        color-profile-name="RGBColorProfile" src="AppleRGB.icc"/> ¢
        <fo:color-profile
        color-profile-name="CMYKColorProfile" src="AppleCMYK.icc"/>
    </fo:declarations>
    ...

    <fo:page-sequence master-reference="default-sequence">
        <fo:flow flow-name="xsl-region-body"
        font-size="12pt" font-family="Times New Roman">

            <fo:block background="rgb(255,0,0)">RGB Color</fo:block>
            <fo:block background="rgb-icc(255,0,0,
            #RGBColorProfile, 1, 0, 0) □">
                ICC RGB Color
```

```
            </fo:block>

            <fo:block background="rgb-icc(255,0,0,
            #CMYKColorProfile, 0, 1, 1, 0) □">
                ICC CMYK Color
            </fo:block>

        </fo:flow>

    </fo:page-sequence>
</fo:root>
```

8.4 Typografische Gestaltung

Auf den folgenden Seiten geht es um die Grundlagen der typografischen Gestaltung. Sie werden sehen, wie sich Seiteninhalte gestalten lassen. Zunächst werden die elementaren Dinge gezeigt. Dazu gehören Blöcke, Container und Container für Blöcke. Im weiteren Verlauf werden dann Angaben zur Schriftgestaltung, zu den Rahmen und den Abständen vorgestellt. Den Anfang machen allerdings Aspekte, die oftmals vergessen bzw. vernachlässigt werden. Denn im FO-Konzept gibt es Prinzipien, durch die sich die Gestaltung von Inhalten abkürzen und somit vereinfachen lässt.

8.4.1 Defaults, Vererbung, Verschachtelung

Im FO-Konzept existieren drei Aspekte für eine einfachere Gestaltung.

- **Defaults** – Für die meisten FO-Eigenschaften existieren Standardwerte. Dank denen muss man in sehr vielen Gestaltungssituationen die betreffende Eigenschaft nicht explizit angeben.
- **Vererbung** – Dort, wo es sinnvoll ist, vererben sich die FO-Eigenschaften, die auf einer höheren Ebene der FO-Struktur spezifiziert sind, in der Dokumenthierarchie automatisch nach unten. Diese Vererbung wird so lange fortgesetzt, bis die Eigenschaft ggf. durch eine Spezifikation auf einer untergeordneten Ebene neu oder anders gesetzt wird.
- **Verschachtelung** – FO-Strukturen[3] können Sie – wo das sinnvoll erscheint – verschachteln, also rekursiv nutzen.

Die genannten Konzepte funktionieren genauso wie in CSS und werden daher an dieser Stelle nicht weiter unter die Lupe genommen.

8.4.2 Seitenzahlen einfügen

Eine der am meisten nachgefragten Anwendungen ist zweifellos die nach dem Einfügen der Seitenzahlen. Das ist auch der Grund, warum dieser Aspekt hier so zeitig in diesem Kapitel behandelt wird. So können Sie bei Bedarf Ihr Dokument gleich von Anfang an mit entsprechenden Seitenzahlen ausstatten.

[3] Das sind die Elemente mit dem Namensraum-Präfix `fo`.

Eigentlich ist es ganz einfach. Zunächst einmal muss man die Fußzeile definieren, in der die Seitenzahlen letztendlich angezeigt werden sollen.

```
<fo:region-body margin-bottom="1cm"/>

<fo:region-after extent="1cm"/>
```

Die aktuelle Seitenzahl wird über `fo:page-number` ermittelt. Dieses Element muss nun nur noch in die Fußzeile eingefügt werden.

Listing 8.33 Die Seitenzahl wird eingefügt.

```
<fo:static-content flow-name="xsl-region-after">
   <fo:block text-align-last="right">
      <fo:page-number/>
   </fo:block>
</fo:static-content>
```

Soweit ein kurzes Beispiel. Wie sich das Ganze innerhalb einer etwas komplexeren Anwendung darstellt, zeigt die folgende Syntax.

Listing 8.34 Die aktuelle Seitenzahl und die Gesamtseitenzahl werden ausgegeben.

```
  <fo:page-sequence master-reference="buch">
       <fo:static-content flow-name="fusszeile">

            <fo:block text-align="right">
                Seite
                <fo:page-number />
                von
                <fo:page-number-citation  ref-id="ende" />
            </fo:block>

       </fo:static-content>

       <fo:flow flow-name="xsl-region-body">

            <fo:block>
               Das ist die erste Seite
            </fo:block>

            <fo:block break-before="page">
                Das ist die zweite Seite.
            </fo:block>

            <fo:block id="ende" />

       </fo:flow>

    </fo:page-sequence>
```

Auch hier wurde eine Fußzeile definiert, in der jeweils die Seitenzahl nach dem Schema `Seite x von x` ausgegeben wird. Das Element `fo:page-number-citation` gibt die Seitenzahl des letzten Blockelements im `fo:flow`-Element zurück. Auf diese Weise kann man die Gesamtseitenzahl ermitteln.

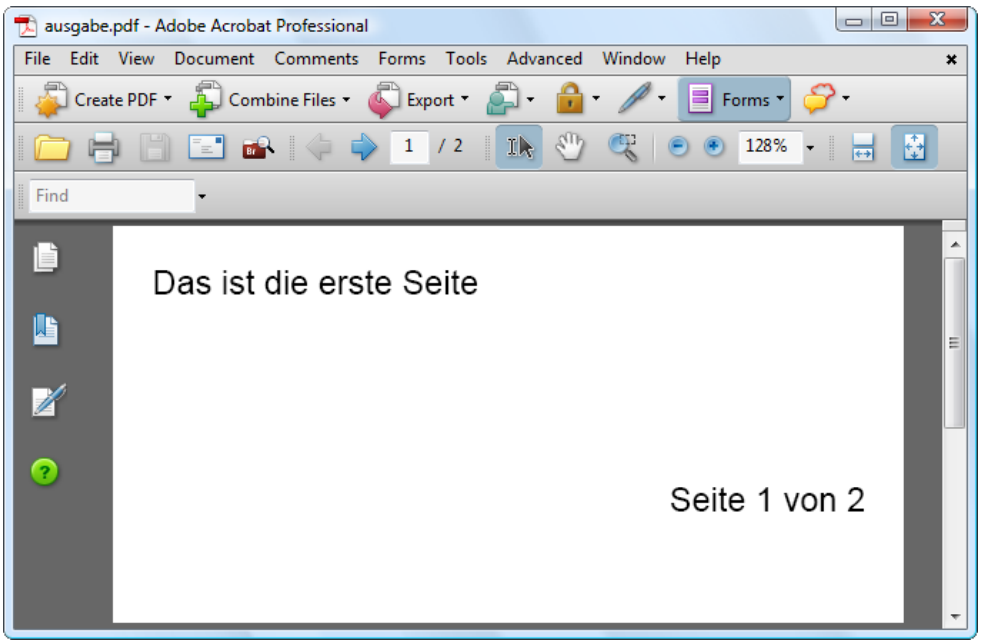

Abbildung 8.12 Die Seitenzahlen werden ermittelt und eingetragen.

8.5 Rahmen und Ränder

In XSL-FO gibt es ein Konzept für die Positionierung von Inhalten in Blöcken sowie für die Generierung von Rahmen, Einrückungen, Rändern und Abständen. Auf den folgenden Seiten werden die verschiedenen Möglichkeiten vorgestellt. Die gezeigten Varianten orientieren sich sehr stark an CSS. Wer sich also mit den Cascading Stylesheets auskennt, dem werden die entsprechenden Attribute und Werte bekannt vorkommen.

8.5.1 Außenabstände bestimmen

Über die verschiedenen `margin`-Attribute lassen sich Außenränder bzw. Abstände definieren. Durch beides wird ein erzwungener Leerraum zwischen dem aktuellen Element und seinem Eltern- oder Nachbarelement eingefügt.

Die Angaben beziehen sich auf die Flächen zwischen den Bereichsgrenzen und den darin liegenden Blöcken sowie den Flächen zwischen den Blöcken.

Insgesamt sind fünf verschiedene Attribute verfügbar.

- `margin-top` – Außenabstand nach oben
- `margin-bottom` – Außenabstand nach unten
- `margin-left` – Außenabstand nach links
- `margin-right` – Außenabstand nach rechts

- `margin` – Dabei handelt es sich um eine Zusammenfassung der vier Attribute `margin-top`, `margin-bottom`, `margin-left` und `margin-right`.

Normalerweise sollten Randabstände immer für alle Seiten angegeben werden. Warum das so ist, zeigen die folgenden Beispiele.

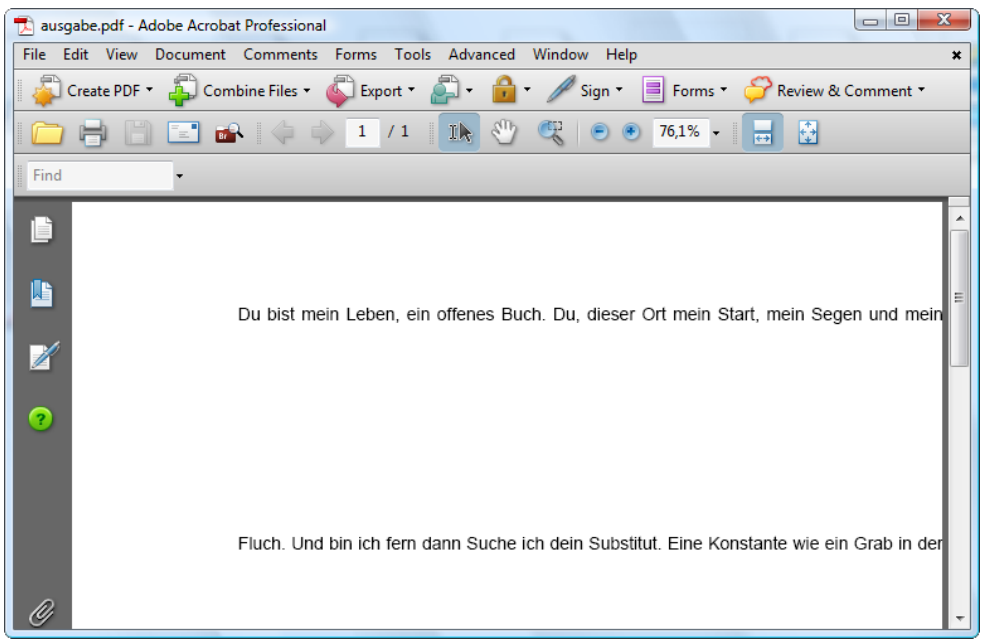

Abbildung 8.13 Der Text passt nicht mehr in die Seite.

In diesem Beispiel wurde mit `margin-left` ein Außenabstand nach links definiert. Auf der linken Seite ist das so auch völlig in Ordnung. Problematisch ist jedoch die rechte Seite. Denn dort läuft der Text über den Rand hinaus. Verhindern lässt sich das durch die Definition von Außenabständen auf den jeweils gegenüberliegenden Seiten. Im aktuellen Fall wurde das folgendermaßen gelöst:

Listing 8.35 Zwei verschiedene Randabstände

```
<fo:block line-height="5.5cm" text-align="justify" margin-left="4cm"
margin-right="3cm">
   Du bist mein Leben,
   ein offenes Buch.
   Du, dieser Ort mein Start,
   mein Segen und mein Fluch.
   Und bin ich fern dann Suche
   ich dein Substitut.
   Eine Konstante wie ein Grab
   in der mein Leben ruht.
</fo:block>
```

Hier wurden zwei unterschiedliche Außenabstände für die linke und für die rechte Seite definiert.

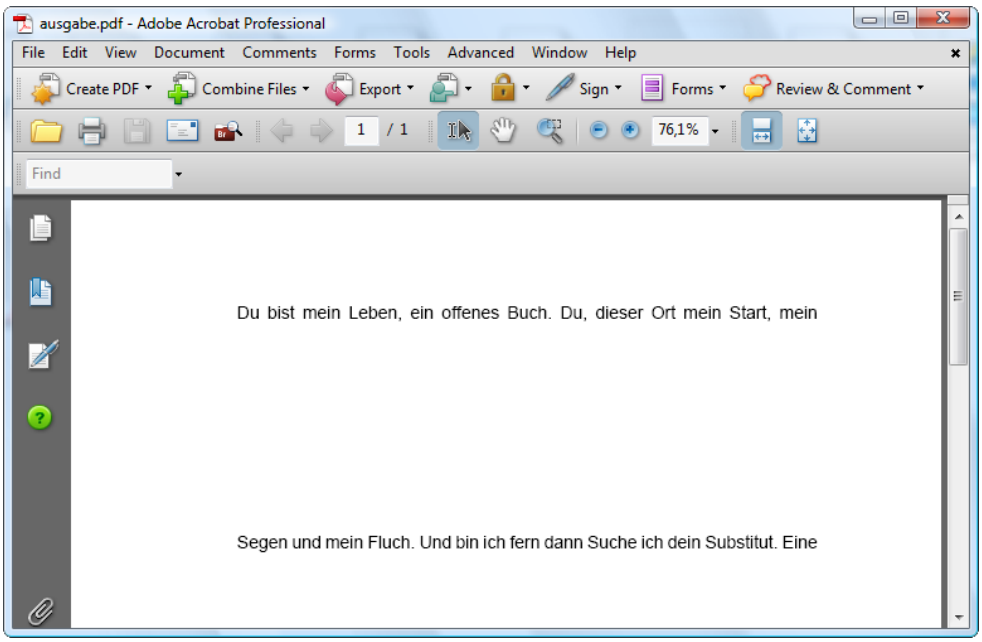

Abbildung 8.14 Der gleiche Abstand auf beiden Seiten.

Ebenso könnte man aber auch auf beiden Seiten den gleichen Außenabstand festlegen. Das gelingt über die allgemeine `margin`-Eigenschaft.

Listing 8.36 So klappt das mit den gleichen Abständen.

```
<fo:block line-height="5.5cm" text-align="justify" margin="4cm">
   Du bist mein Leben,
   ein offenes Buch.
   Du, dieser Ort mein Start,
   mein Segen und mein Fluch.
   Und bin ich fern dann Suche
   ich dein Substitut.
   Eine Konstante wie ein Grab
   in der mein Leben ruht.
</fo:block>
```

Und zum Abschluss noch ein Beispiel, in dem verschiedene `margin`-Angaben verwendet werden.

Listing 8.37 Es werden verschiedene `margin`-Werte eingesetzt.

```
<fo:block margin-left="40mm">
   Abstand nach links (40mm)

      <fo:block margin-left="100mm">
         Abstand nach links (100mm)
      </fo:block>

   </fo:block>

<fo:block margin-top="50mm">
   Abstand nach oben 50mm
</fo:block>
```

Das Ergebnisdokument sieht folgendermaßen aus:

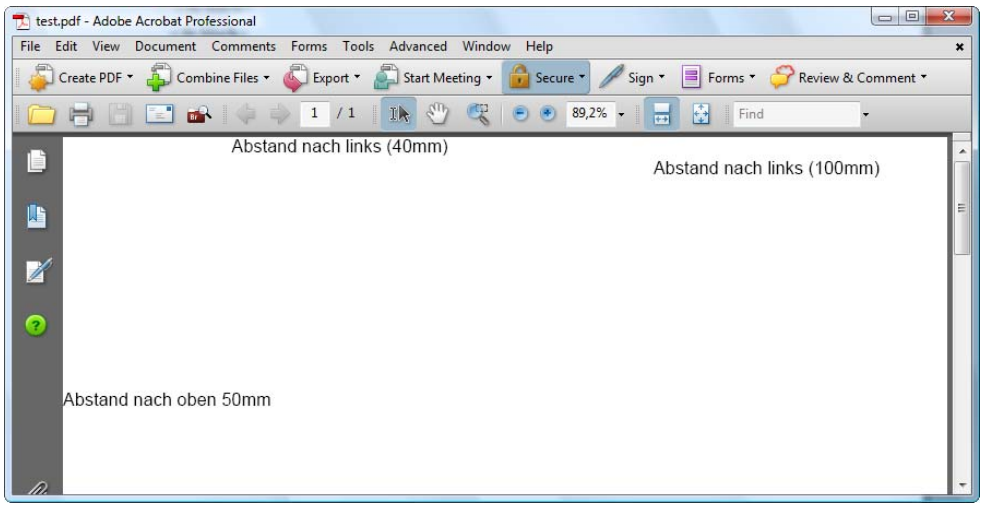

Abbildung 8.15 Hier wurden verschiedene `margin`-Anweisungen verwendet.

Noch einige allgemeine Hinweise dazu, nach welchen Regeln die Formatierer die Ränder und Blöcke setzen.

- Werden keine Ränder angegeben, erstreckt sich der Block über die gesamte Breite des Satzspiegels. Solche Blöcke, die aufeinanderfolgen, werden ohne weitere Abstände untereinander angeordnet. Das rührt daher, dass in XSL-FO die Standardwerte für Ränder standardmäßig auf `0` gesetzt sind.
- Negative Werte für linke und rechte Ränder können zum Überschreiten der Grenzen des Satzspiegels führen.
- Gibt man negative Werte für unten und oben an, werden diese ignoriert. Man sollte und braucht diese demnach nicht zu definieren.
- Gibt man für den linken oder rechten Rand einen positiven Wert an, wird der Block links oder rechts entsprechend eingezogen.
- Weist man dem oberen oder unteren Rand einen positiven Wert zu, entsteht unter oder über dem Block eine entsprechende Fläche. Ein darüber oder darunter stehender Block wird verschoben.
- Die Werte für Ränder werden nicht vererbt.

Vorsicht ist bei verschachtelten Blöcken geboten. Denn dort übernimmt der innere Block die Werte des übergeordneten Blocks bzw. addiert diese hinzu. Das ist übrigens kein Widerspruch zu dem letzten Punkt in der Aufzählung *Die Werte für Ränder werden nicht vererbt.* Denn wenn Blöcke ineinander verschachtelt sind, werden die Werte des äußeren auf die des inneren Blocks übernommen. Das folgende Beispiel soll diesen Aspekt verdeutlichen:

Listing 8.38 Mehrere Blöcke wurden definiert.

```
<fo:block margin-left="20mm">
   Du bist mein Leben,
   ein offenes Buch.

<fo:block margin-left="20mm">
   Du, dieser Ort mein Start,
   mein Segen und mein Fluch.
</fo:block>
</fo:block>

<fo:block margin-top="20mm" margin-left="-10mm">
   Und bin ich fern dann Suche
   ich dein Substitut.
</fo:block>

<fo:block margin-top="20mm" margin-left="-30mm">
   Eine Konstante wie ein Grab
   in der mein Leben ruht.
</fo:block>
```

Hier wird im ersten Block durch `margin-left` ein Abstand von 20 Millimetern zur linken Satzspiegelgrenze erzeugt. Bei dem zweiten Block handelt es sich um ein Kindelement des ersten Blocks. Dieser Block wird ebenfalls um 20 Millimeter eingezogen. Allerdings kommen hier noch die 20 Millimeter des Elternblocks hinzu. Insgesamt wird der zweite Block somit also um 40 Millimeter eingezogen. Die anderen beiden Blöcke zeigen noch einmal, was für Effekte mit negativen Werten möglich sind.

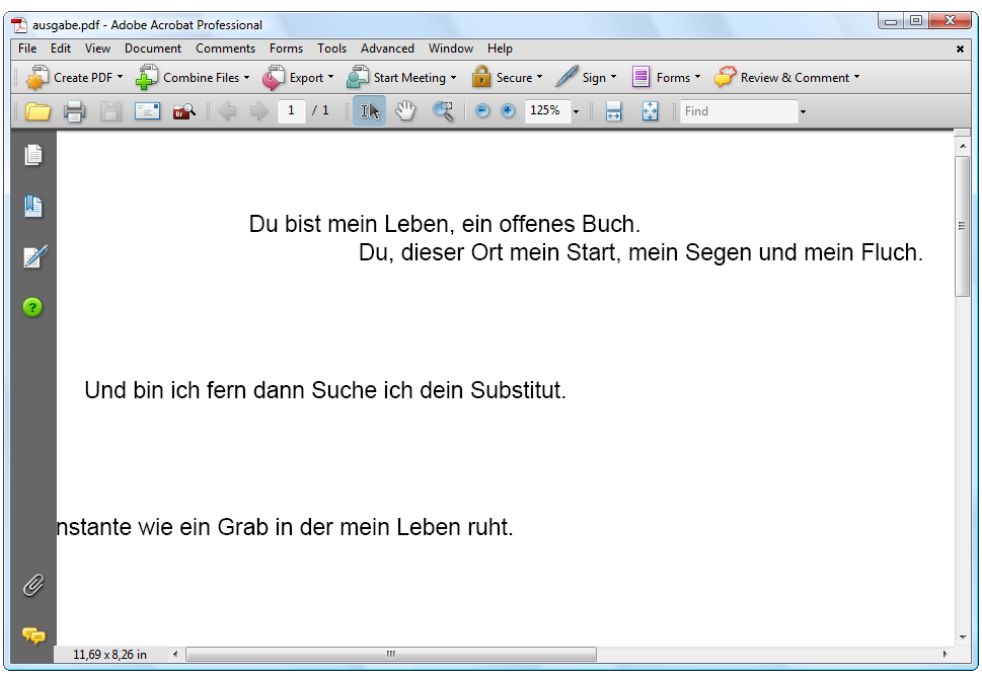

Abbildung 8.16 Alles wird unterschiedlich angezeigt.

8.5.2 Innenabstände

Bei dem Innenabstand handelt es sich um den erzwungenen Leerraum zwischen dem Elementinhalt und seinem eigenen Elementrand. Sinnvoll sind die folgenden Eigenschaften in Elementen, die einen eigenen Absatz erzeugen beziehungsweise einen Block bilden.

Hier eine Übersicht der möglichen `padding`-Attribute:

- `padding-start` – Bestimmt den Abstand am Anfang des Blocks.
- `padding-end` – Hierüber definiert man den Abstand am Ende des Blocks.
- `padding-left` – Legt den Abstand zwischen Blockinhalt und linker Blockgrenze fest.
- `padding-right` – Legt den Abstand zwischen Blockinhalt und rechter Blockgrenze fest.
- `padding-before` – Bestimmt den Abstand vor dem Block.
- `padding-after` – Definiert den Abstand nach dem Block.
- `padding-top` – Legt den Abstand zwischen Blockinhalt und oberer Blockgrenze fest.
- `padding-bottom` – Legt den Abstand zwischen Blockinhalt und unterer Blockgrenze fest.
- `padding` – Dabei handelt es sich um die Kurzform der vier Einzelangaben `padding-left`, `padding-right`, `padding-top` und `padding-bottom`.

Beim Umsetzen der `padding`-Attributwerte wenden die Formatierer folgende Regeln an:

- Wurden auf der linken und/oder rechten Seite mittels `margin` Ränder gesetzt, gelten die `padding`-Werte von dort aus.
- Gibt es keine `margin`-Angaben, wird der betreffende Block über die Satzspiegelgrenzen nach links oder rechts vergrößert.
- Angaben zu `padding-top` oder `padding-bottom` führen zum Einfügen von Abständen zum vorhergehenden oder nachfolgenden Block. Sind solche Abstände gewünscht, sollte man normalerweise auf `space` oder `margin` zurückgreifen.
- Die `padding`-Werte werden nicht vererbt.
- Negative Werte werden ignoriert.

Das folgende Beispiel zeigt, wie sich durch den Einsatz von `padding` Texte übersichtlicher gestalten lassen.

Listing 8.39 Hier wird mit verschiedenen Innenabständen gearbeitet.

```
<fo:block padding-before="12pt" padding-after="6pt"
    font-size="150%" font-weight="bold"
    background-color="#afafaf">Dein Leben</fo:block>

   <fo:block padding-before="9pt" padding-after="3pt"
    text-indent="0.5in" text-align="justify">
    Ich bin der Wind, der durch Deine Strasse zieht,
    und ich bin in der Schlange bei dem Gig von der
    Band, die Du liebst.
    Ich war auf dem Spielplatz, bei Deinem ersten Kuss wurdest
    Du zum Mann ein erstes Mal Romantik und die Einsicht,
```

```
    dass man Herzen brechen kann.
</fo:block>

 <fo:block padding-before="12pt" padding-after="3pt"
    font-weight="bold">
    All das kann Dir keiner nehmen, das ist echt,
    das ist Dein Leben.
</fo:block>

<fo:block padding-before="9pt" padding-after="3pt"
    text-indent="0.5in" text-align="justify">
    Ich bin die Nacht, gepaffte Kippen und der
    Geist, der bei Dir wohnt.
    Deine erste Gitarre, Lachen und Trinken und
    Küssen unter dem dunklen Mond.
    Wo ich bin, ist oben, und die Gewissheit, dass
    ich nicht verlier. Etwas verband uns, was keiner wusste,
    außer Dir und mir
   Jetzt führst Du mich und ich hoffe, Du wählst den richtigen Weg
   ich vertraue Dir, wenn ich mein Schicksal in Deine Hände leg
</fo:block>
```

Und auch hier natürlich wieder das entsprechende Ergebnisdokument.

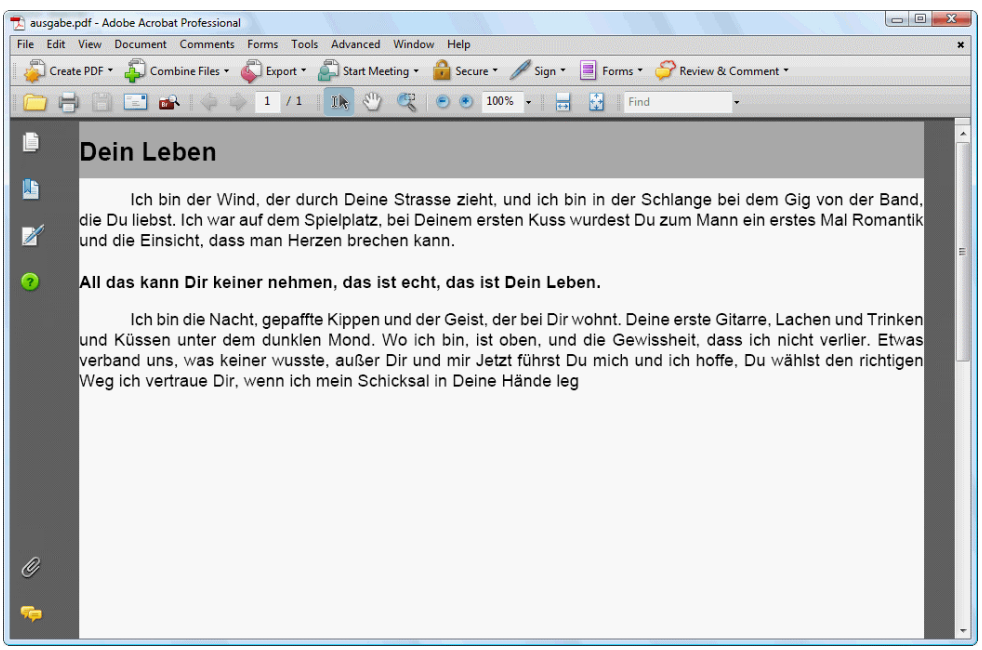

Abbildung 8.17 So sieht es schon besser aus.

Es lassen sich durch den Einsatz der `padding`-Attribute also durchaus ansprechende Ergebnisse erzielen.

Die Definition für die Innenabstände lässt sich auch verkürzen. Möchte man beispielsweise auf allen Seiten den gleichen Abstand, kann man folgende Syntax verwenden:

```
<fo:block padding="2pt">
```

Darüber hinaus sind auch noch andere Varianten möglich.

```
padding="2pt 4pt"
```

Der erste Wert bestimmt die Abstände oben und unten. Der zweite Wert legt die Abstände für links und rechts fest.

```
padding="2pt 4pt 6pt"
```

Bei drei Werten bedeutet die erste Angabe den Abstand für oben, die zweite den Abstand für rechts und links und die dritte den Abstand nach unten.

```
padding="2pt 4pt 6pt 8pt"
```

Die erste Angabe definiert den Abstand für oben, die zweite den Abstand für rechts, die dritte den Abstand für unten und die vierte den Abstand für links.

8.5.3 Vertikale Abstände

Über die verschiedenen `space`-Attribute lassen sich vertikale Abstände, die sogenannten Vorschübe, zwischen Blöcken definieren. Jetzt könnte man natürlich einwerfen, dass doch genau dafür bereits die `margin`-Attribute zur Verfügung stehen. Denn schließlich lässt sich doch auch mit denen ein entsprechender Abstand zu anderen Blöcken definieren. Dabei werden die Rändern allerdings lediglich hinzuaddiert, wenn für den vorherigen Block ein `margin-bottom`-Wert und für den nachfolgenden Block ein Wert für `margin-top` angegeben wurde.

Durch die `space`-Attribute kann man die Abstände sehr viel feiner steuern. Für die beiden Attribute `space-after` und `space-before` stehen die folgenden Varianten zur Verfügung:

- `space-after.precedence` bzw. `space-before.precedence` – Definiert einen Vorschub davor bzw. einen Vorschub danach.
- `space-after.conditionality` bzw. `space-before.conditionality` – Legt einen bedingten Vorschub bei Seitenumbrüchen fest.
- `space-after.optimum` bzw. `space-before.optimum` – Hierüber wird ein optimaler Vorschub bestimmt.
- `space-after.minimum` bzw. `space-before.minimum` – Legt den minimalen Vorschub fest.
- `space-after.maximum` bzw. `space-before.maximum` – Legt den maximalen Vorschub fest.

Durch `.precedence` wird für den Fall, dass `after`- und `before`-Werte aufeinandertreffen, bestimmt, welcher Wert bevorzugt behandelt werden soll. Die Größe des entsprechenden Vorschubs wird dann über `space-after` bzw. `space-before` oder mit `.optimum`, `.minimum`, `.maximum` bestimmt.

Mit `.conditionality` legt man das Verhalten von zwei aufeinanderfolgenden Vorschub-Spezifikationen bei Seitenumbrüchen (`space-after` im vorherigen, `space-before` im folgenden Block) über die beiden Attribute `discard` und `retain` fest. Dabei gilt `discard` dann als Vorgabewert, wenn `.conditionality` nicht verwendet wird. Dadurch wird er-

reicht, dass bei Seitenumbrüchen der Block auf der neuen Seite immer am oberen Satzspiegel beginnt. Verwendet man `retail`, sieht das anders aus. Dann nämlich wird der Abstand zum oberen Satzspiegel auf der neuen Seite beibehalten.

Zunächst ein einfaches Beispiel:

Listing 8.40 Hier wird mit vertikalen Abständen gearbeitet.

```
<fo:block space-after="20mm" border-style="solid" border-color="black"
border-width="5mm" font-size="30pt">
   An manchen Tagen riechen Blumen schlecht, selbst Rosen stinken.
   In diesen Stunden steigt die Kälte auf und Schiffe sinken.
</fo:block>

<fo:block space-before="20mm" padding="5mm 5mm 5mm 5mm" border-
style="solid" border-color="black" border-width="5mm" font-size="20pt">
   Die Wärme im Leben ist unbeständig hier.
   Verlorene Seelen, traurige Augen, Leere in mir.
</fo:block>
```

Und auch hier wieder die Ansicht im Ergebnisdokument:

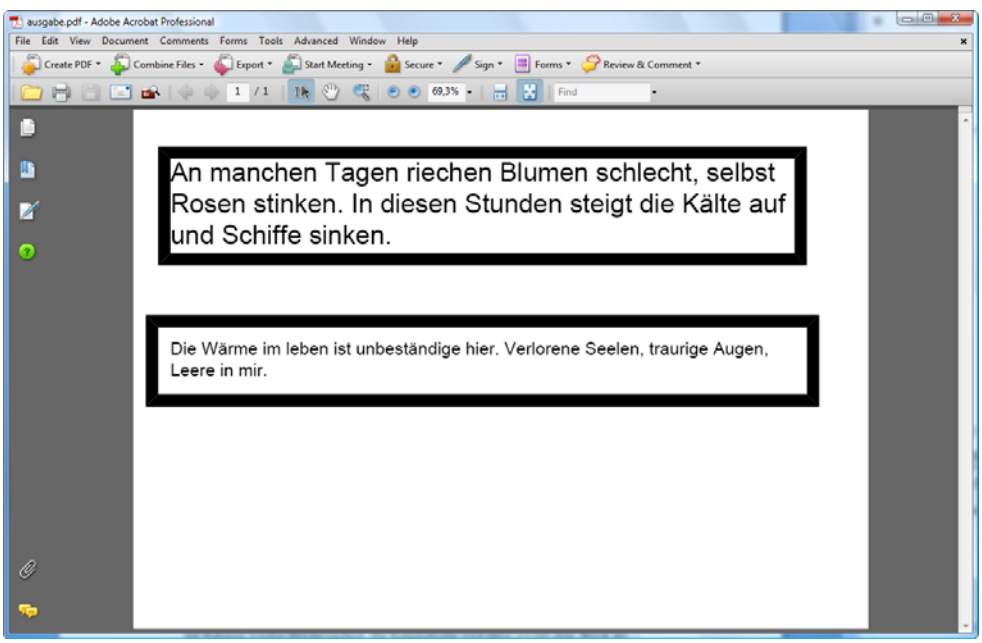

Abbildung 8.18 Hier wurden Abstände definiert.

Durch die beiden Attribute `space-after` und `space-before` wird zwischen den beiden Blöcken ein Abstand eingefügt. Dieser Abstand beträgt 20 Millimeter.

8.5.4 Rahmen definieren

Mit Rahmen werden Blöcke eingefasst. Die Rahmenbreite wird dabei jeweils dem Block als Fläche außerhalb hinzugefügt. Für die Definition vorn Rahmen wird das `border`-Attribut verwendet. Das `border`-Attribut kennt die folgenden vier Werte:

- `border-before` – Rand oben
- `border-after` – Rand unten
- `border-start` – Rand links
- `border-end` – Rand rechts

Das sind die Grundvarianten. Durch zusätzliche Spezifikationen lassen sich die Rahmen noch weiter anpassen.

- `color` – Rahmenfarbe
- `style` – Legt den Stil des Rahmens fest. Die möglichen Werte werden im Anschluss vorgestellt.
- `width` – Rahmenbreite bzw. Linienstärke

Über `style` lässt sich die Rahmenart bestimmen. Dabei sind verschiedene Angaben möglich.

- `none` – kein Rahmen
- `hidden` – versteckter Rahmen
- `dotted` – gepunktet
- `dashed` – gestrichelt
- `solid` – durchgezogen
- `double` – doppelt durchgezogen
- `groove` – 3D-Linie
- `ridge` – 3D-Linie
- `inset` – 3D-Linie
- `outset` – 3D-Linie

Auch diese Angaben sind Ihnen möglicherweise von CSS her bekannt. Wie eine solche Rahmendefinition aussehen kann, zeigt das folgende Beispiel.

Listing 8.41 Eine einfache Rahmendefinition.

```
<fo:block border-style="solid" border-width="1mm" margin-bottom="10mm"
padding="12mm 12mm 12mm 12mm">
   Ich bin der Text

<fo:block border-style="dotted" border-width="1mm" padding="10mm 10mm
10mm 10mm">
   Ich bin der Text
</fo:block>
</fo:block>
```

```
<fo:block border-style="ridge" border-width="1mm">
   Ich bin der Text
</fo:block>
```

Damit Sie sich etwas unter den zuvor beschriebenen Rahmenstilen etwas vorstellen können, zeigt **Abbildung 8.19** die möglichen Varianten.

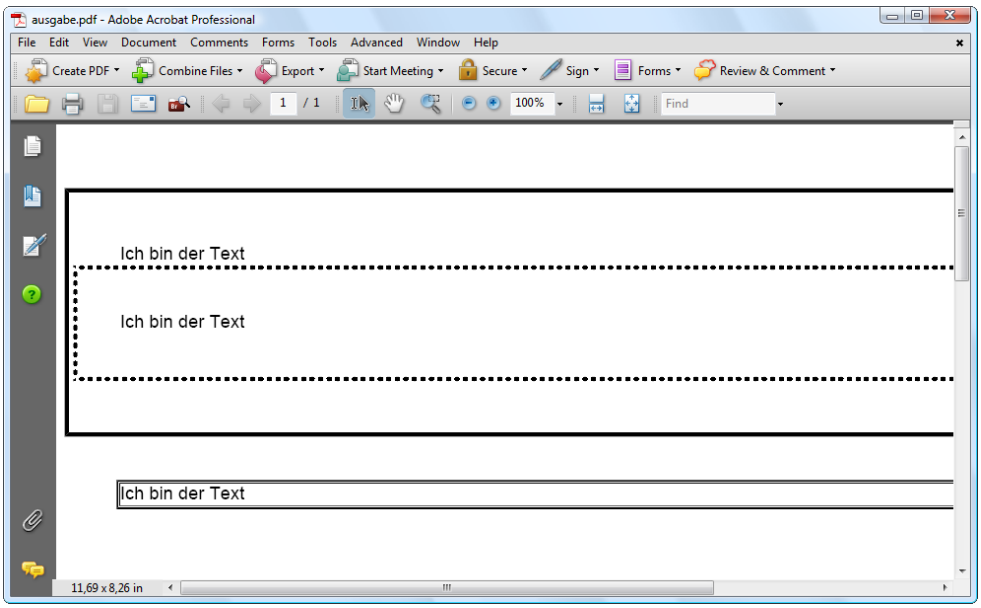

Abbildung 8.19 Das sind die verschiedenen Rahmenarten.

Anstelle der Einzeldefinitionen kann man auch gleiche Definitionen für alle Rahmenseiten vornehmen.

Listing 8.42 Eine verkürzte Rahmendefinition

```
<fo:block border-width="1pt" border-style="ridge"
border-color="red">
...
</block>

<fo:block border-width="2pt 3pt" border-style="grove"
border-color="red yellow blue green">
...
</block>
```

Die Syntax lässt sich sogar noch weiter verkürzen.

```
<fo:block border="2pt dotted green">
```

Eine ähnliche Abkürzung ist auch möglich, wenn die Angaben nur für eine Rahmenseite gelten sollen.

```
<fo:block border-top="1pt dashed yellow">
```

8.6 Schriftgestaltung

XSL-FO stellt zahlreiche Attribute für die Gestaltung von Texten zur Verfügung. Diese reichen von Möglichkeiten der unterschiedlichen Ausrichtung über Einrückungen bis hin zur Definition verschiedener Schriftarten. Genau darum geht es auf den folgenden Seiten. Viele der hier vorgestellten Attribute und deren Werte sind verwandt mit CSS. So wird z.B. auch in XSL-FO zum Ausrichten von Text `text-align` verwendet.

8.6.1 Ausrichtung

Für die Ausrichtung von Texten gibt es die beiden Attribute `text-align` und `text-align-last`. Über beide Attribute wird bestimmt, wie der Textinhalt von Blöcken ausgerichtet werden soll. Mit `text-align-last` lässt sich die letzte Zeile eines Blocks gesondert behandeln.

Beide Attribute kennen die folgenden Werte:

- `center` – zentrierte Ausrichtung
- `left` – linksbündig, rechts flatternd
- `right` – rechtsbündig, links flatternd
- `justify` – Blocksatz
- `inside` – linke Seite rechtsbündig, rechte Seite linksbündig
- `outside` – linke Seite linksbündig, rechte Seite rechtsbündig
- `start` – Diese Angabe ist in Verbindung mit einer anderen Schreibrichtung als `lr-tb` wichtig. Hierüber lässt sich nämlich die Bündigkeit mit dem `start`-Bereich definieren.
- `end` – Auch diese Angabe ist in Verbindung mit einer anderen Schreibrichtung als `lr-tb` wichtig. Hierüber lässt sich nämlich die Bündigkeit mit dem `end`-Bereich definieren.

Das folgende Beispiel zeigt einige typische Ausrichtungsvarianten:

Listing 8.43 Verschiedene Ausrichtungsvarianten werden eingesetzt.

```
<fo:block space-before="20mm" padding="5mm 5mm 5mm 5mm"  text-
align="left" space-after="1em">
   Willkommen
</fo:block>

<fo:block space-before="20mm" padding="5mm 5mm 5mm 5mm"  text-
align="center" space-after="1em">
   auf meinem
</fo:block>

<fo:block space-before="20mm" padding="5mm 5mm 5mm 5mm"  text-
align="right" space-after="1em">
   PDF-Dokument!
</fo:block>
```

Das Ergebnis sehen Sie auf **Abbildung 8.20**.

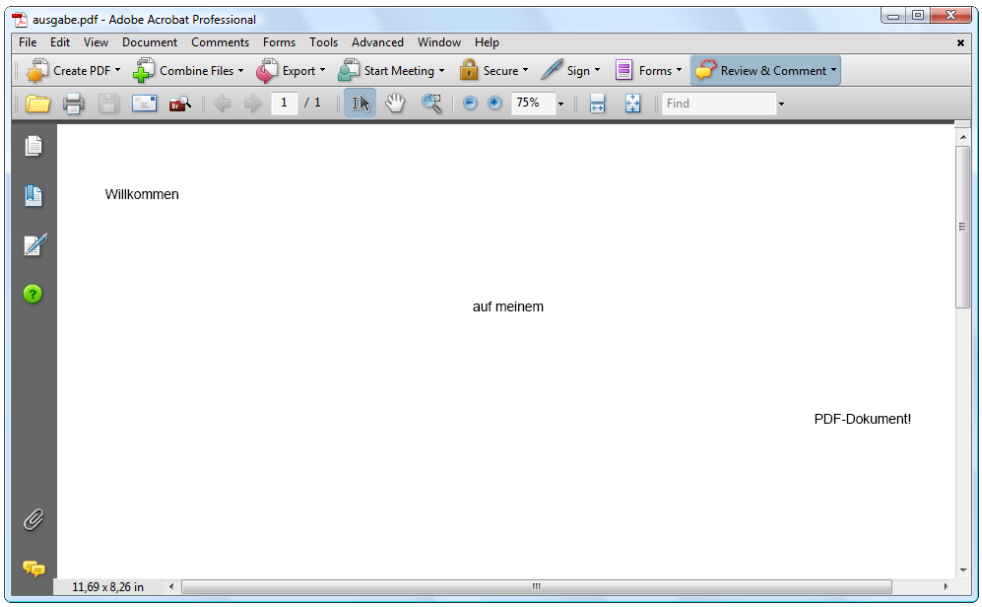

Abbildung 8.20 Alle drei Varianten in Aktion

8.6.2 Mit Einrückungen arbeiten

Für die Realisierung von Einrückungen stellt XSL-FO gleich mehrere Elemente zur Verfügung. Da wären einmal `start-indent` und `end-indent`. Mit beiden Attributen lassen sich ganze Blöcke einrichten. Das bedeutet also, dass alle Zeilen eines Blocks gleichmäßig eingerückt werden.

Listing 8.44 So werden Einrückungen definiert.

```
<fo:block start-indent="20mm">
   Und ich weiß nicht, wohin der Weg geht,
   was am Ende in deinem Buch steht, doch
   ich weiß, wir gehen schon mal vor.
</fo:block>

<fo:block space-before="2cm">
   So was verglüht nicht, so was verbrennt.
   Neid ist hässlich und Dummheit hemmt,
   und ich weiß, wir gehen schon mal vor.
</fo:block>
```

Das Ergebnis dieser Syntax zeigt **Abbildung 8.21**.

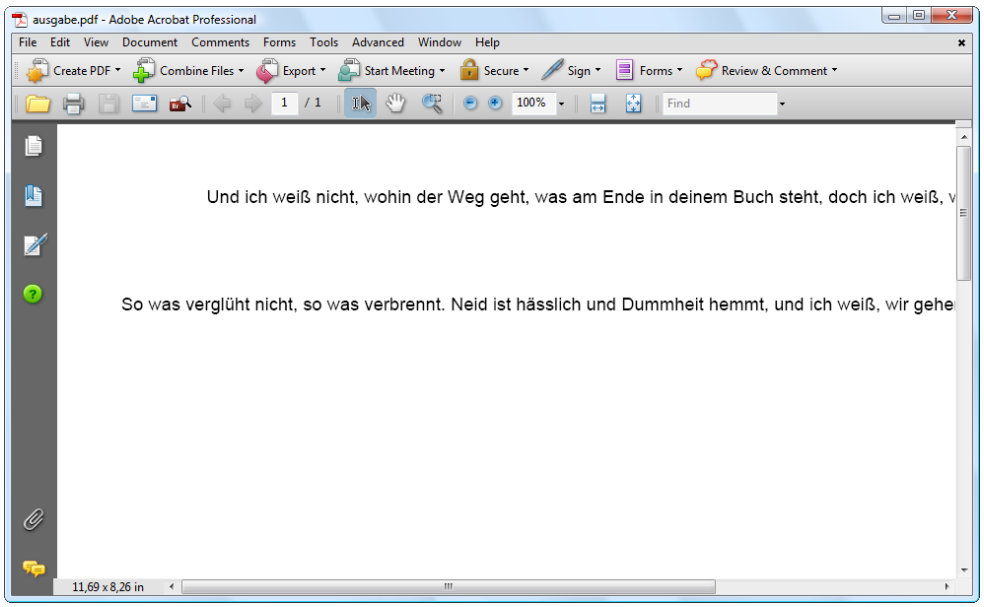

Abbildung 8.21 Hier wurde mit Einrückungen gearbeitet.

Neben `start-indent` und `end-indent` gibt es auch noch `text-indent` und `last-line-end-indent`. Das Attribut `text-indent` greift lediglich auf die erste Zeile im Startbereich zu. `last-line-end-indent` sorgt bei einem positiven Wert dafür, dass die Einrückung des Absatzes im Startbereich stattfindet. Setzt man hingegen einen negativen Wert, bewirkt das die Einrückung des Absatzes im Endbereich.

8.6.3 Schriftart festlegen

Die Schriftfamilie wird über `font-family` gesteuert. Hierbei kommt es immer wieder zu Problemen. Gerade das mag man gar nicht glauben, wenn man sich die sehr einfache Syntax ansieht.

Listing 8.45 `Times` wurde als Schriftart bestimmt.

```
<fo:block font-family="Times">
   Hallo, Welt!
</fo:block>
```

Dem Attribut `font-family` wird die Schriftfamilie oder eine Liste von Schriftfamilien zugewiesen. Bei einer Liste von Schriftfamilien werden die Schriften der Reihe nach im System gesucht. Ist die erste Schrift nicht vorhanden, wird nach der zweiten gesucht usw.

Die Namen der Schriftfamilien müssen durch Leerzeichen getrennt angegeben werden. Namen, die selbst Leerzeichen enthalten, muss man in Hochkommata einschließen.

```
'Times New Roman'
```

Darüber hinaus können auch generische Schriftfamilien angegeben werden. Die folgenden Schriftfamilien sind fest vordefiniert. Diese Angaben können Sie also neben Schriftartennamen verwenden.

- `serif` = Eine Schriftart mit Serifen
- `sans-serif` = Eine Schriftart ohne Serifen
- `fantasy` = Schreibschrift/Zierschrift
- `cursive` = Eine Schriftart für Schreibschrift
- `monospace` = Eine Schriftart mit diktengleichen Zeichen

Es ist empfehlenswert, solche generischen Schriftfamilien als letzte Angabe einer `font-family`-Wertzuweisung anzugeben. Somit hat der Formatierer die Chance, eine Schriftart auszuwählen, die wenigstens vom Typ her der gewünschten entspricht, wenn diese nicht auf dem System vorhanden sein sollte.

```
font-family="Arial, sans-serif"
```

8.6.4 Schriftgröße bestimmen

Für die Definition der Schriftgröße wird das Attribut `font-size` verwendet. Es können alle in XSL-FO verfügbaren Größenangaben verwendet werden.

- `cm`
- `mm`
- `in`
- `pt`
- `pc`
- `px`
- `em`
- `%`

Normalerweise wird die Schriftgröße allerdings in Punkt (`pt`) angegeben. Ebenso kann aber auch eine prozentuale Angabe gemacht werden. Dadurch wird die Schrift relativ zur vererbten Größe, also der in der Umgebung vorherrschenden Schriftgröße, angezeigt.

Millimeter-Werte sind ebenso möglich. Dabei wird jedoch lediglich eine Stelle hinter dem Komma verwertet. Gibt man also

```
14,28mm
```

an, so wird lediglich

```
14,2mm
```

interpretiert. Das mag in vielen Fällen nicht dramatisch sein. Wenn es aber exakt werden soll, verwenden Sie besser einen Punktwert.

Listing 8.46 So wird die Schriftgröße definiert.

```
<fo:block font-family="Times" font-size="14pt">
   Hallo, Welt!
</fo:block>
```

In diesem Fall wird der Text in einer Schriftgröße von 14 Punkt angezeigt.

8.6.5 Zeilenhöhe

Über das Attribut `line-height` lassen sich die Zeilenhöhe und somit auch der Zeilenabstand bestimmen. Bei der Definition der Zeilenhöhe können sämtliche XSL-FO-Maßeinheiten angegeben werden.

- `cm`
- `mm`
- `in`
- `pt`
- `pc`
- `px`
- `em`
- `%`

Es empfiehlt sich, immer die Zeilenhöhe festzulegen. Denn fehlt sie im Stylesheet, kann es passieren, dass die Voreinstellung des jeweiligen Formatierers greift. Eine echte Kontrolle über das Schriftbild ist dann nicht möglich.

Ein Beispiel, in dem verschiedene Zeilenhöhen verwendet werden:

Listing 8.47 Hier wird mit unterschiedlichen Zeilenhöhen gearbeitet.

```
<fo:block line-height="5.5cm" text-align="justify">
   Du bist mein Leben,
   ein offenes Buch.
   Du, dieser Ort mein Start,
   mein Segen und mein Fluch.
   Und bin ich fern dann Suche
   ich dein Substitut.
   Eine Konstante wie ein Grab
   in der mein Leben ruht.
</fo:block>

<fo:block line-height="3.5cm" text-align="justify">
   Trag mich zurück zu dir
   bringt mich zum Beton.
   Legt mich auf dem Asphalt
   dem Boden von dem ich komm.
   bettet mich in grauen Staub
   lasst mich hier allein.
   Die Lichter aus meiner Stadt
   mein Heiligenschein.
</fo:block>
```

Und das Ergebnisdokument sieht folgendermaßen aus:

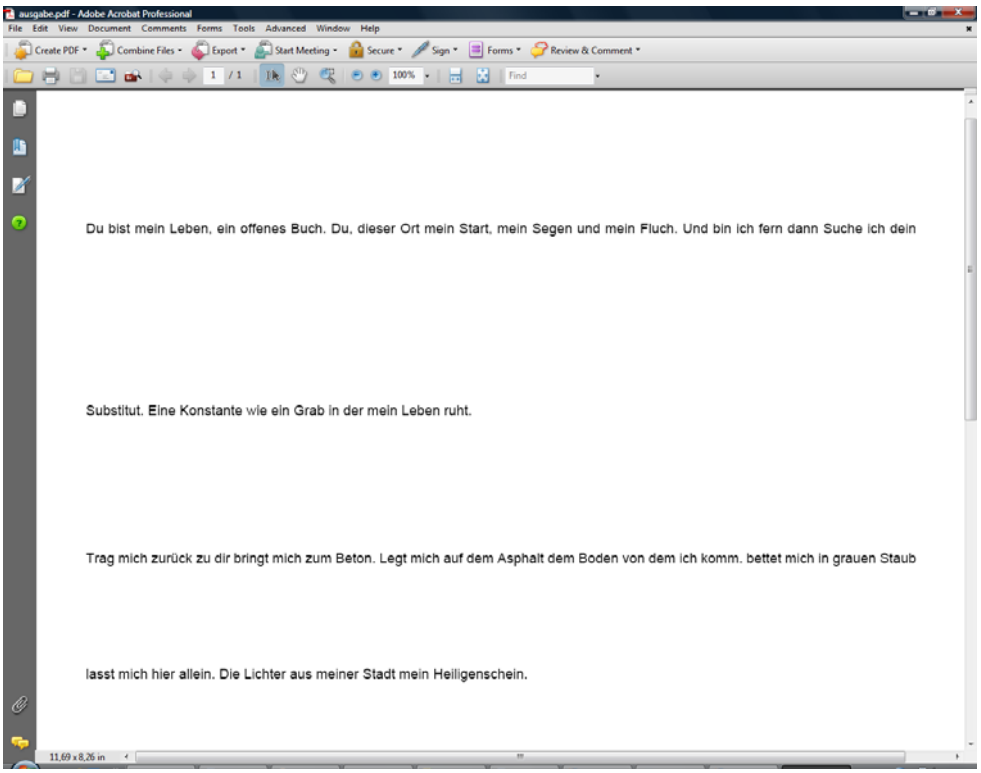

Abbildung 8.22 Hier wurden verschiedene Zeilenhöhen verwendet.

8.6.6 Unterstreichungen & Co.

Auch Unterstriche, Überstriche und durchgestrichene Texte sind möglich. Verwendet wird dafür das Attribut `text-decoration`. Wenn das Attribut auf einen Block angewendet wird, dann gilt der Attributwert für alle untergeordneten Elemente. Definiert man es hingegen für ein einzeiliges Element, dann gilt der Attributwert ausschließlich für den Inhalt des betreffenden Elements.

`text-decoration` kennt die folgenden Werte:

- `underline` – Der Text wird einfach unterstrichen.
- `overline` – Der Text bekommt einen einfachen Überstrich.
- `line-through` – Der Text wird durchgestrichen.
- `none` – Der Text wird normal angezeigt. Dabei handelt es sich um die Standardeinstellung.

Beachten Sie, dass es sich bei `text-decoration` nicht um ein Font-Merkmal, sondern um ein Leistungsmerkmal des Formatierers handelt.

Im folgenden Beispiel werden die verfügbaren Werte für `text-decoration` verwendet. (Lediglich `none` wurde nicht eingesetzt.)

Listing 8.48 Drei von vier Varianten

```
<fo:block text-decoration="underline">
   Unterstrichen
</fo:block>

<fo:block text-decoration="line-through">
   Durchgestrichen
</fo:block>

<fo:block text-decoration="overline">
   Überstrichen
</fo:block>
```

Und auch hier natürlich wieder das Ergebnis:

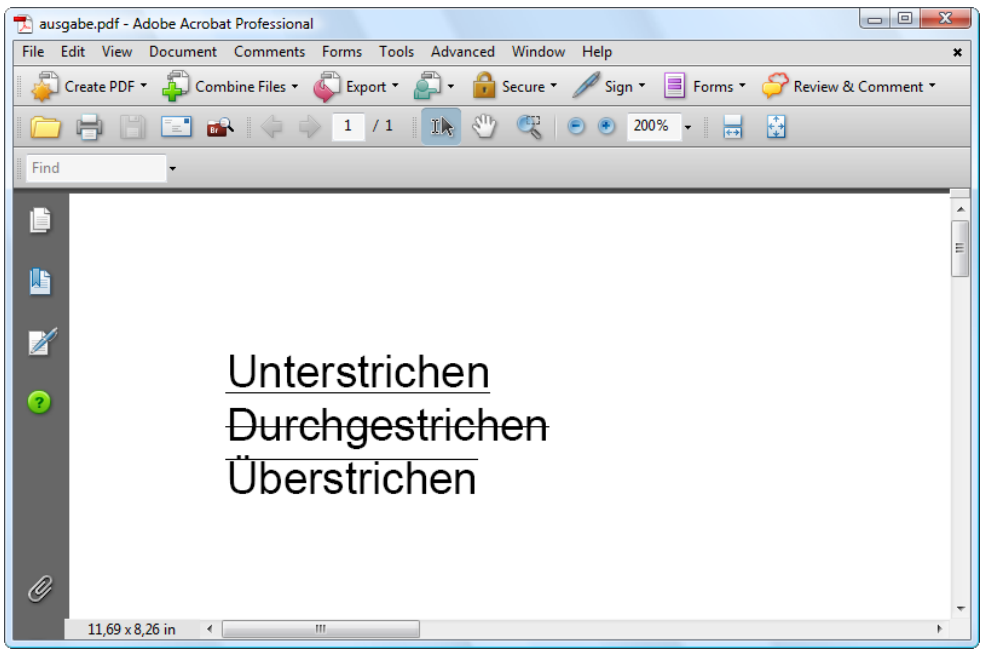

Abbildung 8.23 Die verschiedenen Textdekorationen wurden angewendet.

Sie sollten mit `text-decoration` übrigens sehr vorsichtig umgehen und dieses Attribut nur im begrenzten Maß einsetzen. Texte, bei denen permanent unter- und überstrichen wird, wirken schnell unübersichtlich.

8.6.7 Horizontale Linien innerhalb von Blöcken

Mit dem Element `fo:leader` lassen sich horizontale Linien innerhalb von Zeilen und Blöcken generieren. Bevor ins Detail gegangen wird, hier ein typisches Beispiel, wofür sich solche Linien einsetzen lassen.

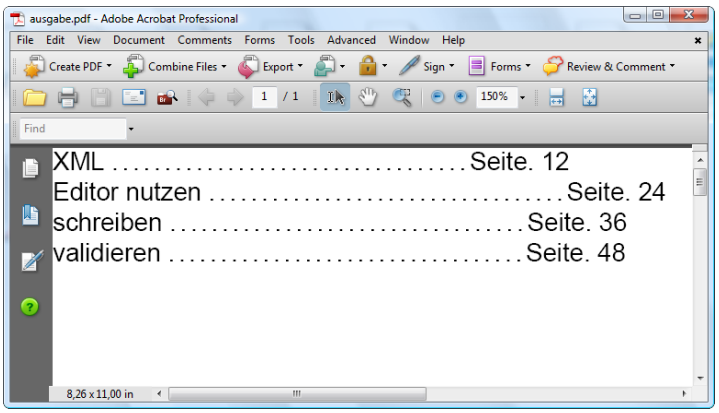

Abbildung 8.24 So könnte ein Index aussehen.

Mit wenigen Zeilen Code kann man so beispielsweise einen Index oder ein Inhaltsverzeichnis generieren. Verwendet wird für das Anlegen von Linien das Element `fo:leader`. Dieses kennt eine Vielzahl an Attributen. Laut der Spezifikation kann aus den folgenden Eigenschaften gewählt werden:

- Spracheigenschaften
- Rahmeneigenschaften
- Abstände
- Hintergründe
- Schrifteigenschaften
- Randabstände
- `alignment-adjust`
- `alignment-baseline`
- `baseline-shift`
- `color`
- `dominant-baseline`
- `text-depth`
- `text-altitude`
- `id`
- `keep-with-next`
- `keep-with-previous`
- `leader-alignment`
- `leader-length`
- `leader-pattern`
- `leader-pattern-width`
- `rule-style`
- `rule-thickness`

- `letter-spacing`
- `line-height`
- `text-shadow`
- `visibility`
- `word-spacing`

Die meisten dieser Eigenschaften werden derzeit von den Formatierern allerdings noch nicht unterstützt. Die folgenden Varianten können Sie aber bedenkenlos einsetzen.

- `leader-length` – Hierüber wird die Linienlänge bestimmt.
- `leader-pattern` – Damit legt man das Füllzeichenmuster fest. Mögliche Werte sind `space` (Leerzeichen), `rule` (Linie), `dots` (gepunktet) und `use-content` (ein beliebiges Zeichenmuster, das als Inhalt im `fo:leader`-Element enthalten ist).
- `rule-style` – Hierüber wird das Aussehen der Linie bestimmt. Mögliche Werte sind `dotted` (gepunktet), `solid` (durchgehend), `double` (doppelt), `groove` (3-D) und `ridge` (3-D).
- `rule-thickness` – Hierüber wird die Dicke der Linie angegeben. Standardmäßig ist die Linie ein Pixel dick.
- `color` – Damit kann die Farbe der Linie bestimmt werden. Standardmäßig wird bei den Formatierern Schwarz verwendet.

Das folgende Beispiel zeigt das eingangs dieses Abschnitts auf **Abbildung 8.24** zu sehende Inhaltsverzeichnis.

Listing 8.49 Linien werden definiert.

```
<fo:block text-align="start">XML
   <fo:leader leader-pattern="dots" leader-pattern-width="5pt"
   leader-alignment="reference-area" leader-length="6cm"/>Seite. 12
 </fo:block>

<fo:block text-align="start">Editor nutzen
   <fo:leader leader-pattern="dots" leader-pattern-width="5pt"
   leader-alignment="reference-area" leader-length="6cm"/>Seite. 24
</fo:block>

<fo:block text-align="start">schreiben
   <fo:leader leader-pattern="dots"
   leader-pattern-width="5pt"
   leader-alignment="reference-area" leader-length="6cm"/>Seite. 36
</fo:block>

<fo:block text-align="start">validieren
   <fo:leader leader-pattern="dots"
   leader-alignment="reference-area"
   leader-pattern-width="5pt" leader-length="6cm"/>Seite. 48
</fo:block>
```

Diese Syntax zeigt, wie die einzelnen Attribute kombiniert werden. Durch die entsprechenden Längenangaben lässt sich so also z.B. ein Inhaltsverzeichnis oder Index erstellen.

Es ist bereits angeklungen, dass sich durch `rule-style` die Linienart bestimmten lässt. Auch hierzu wieder ein Beispiel, in dem die verschiedenen Linienvarianten verwendet werden.

Listing 8.50 Das sind die Linienarten in Aktion.

```
<fo:block text-align="center" space-before.optimum="12pt" space-
after.optimum="12pt">
<fo:leader leader-pattern="rule" leader-length="18cm" rule-style="dashed"
rule-thickness="1pt" color="black"/>
</fo:block>

<fo:block text-align="center" space-before.optimum="12pt" space-
after.optimum="12pt">
<fo:leader leader-pattern="rule" leader-length="18cm" rule-style="dotted"
rule-thickness="1pt" color="black"/>
</fo:block>

<fo:block text-align="center" space-before.optimum="12pt" space-
after.optimum="12pt">
<fo:leader leader-pattern="rule" leader-length="18cm" rule-style="double"
rule-thickness="1pt" color="black"/>
</fo:block>

<fo:block text-align="center" space-before.optimum="12pt" space-
after.optimum="12pt">
<fo:leader leader-pattern="rule" leader-length="18cm" rule-style="groove"
rule-thickness="1pt" color="black"/>
</fo:block>

<fo:block text-align="center" space-before.optimum="12pt" space-
after.optimum="12pt">
<fo:leader leader-pattern="rule" leader-length="18cm" rule-style="ridge"
rule-thickness="1pt" color="black"/>
</fo:block>
```

Abbildung 8.25 zeigt die Linienarten im Ergebnisdokument.

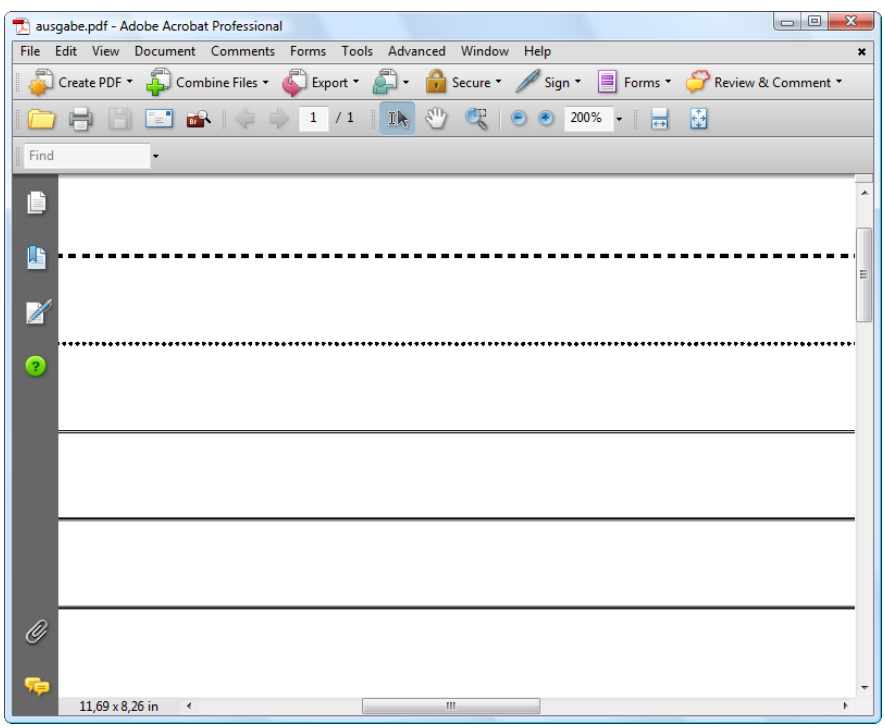

Abbildung 8.25 Das sind die verfügbaren Linienarten.

8.6.8 Silbentrennung

Standardmäßig werden Texte in XSL-FO-Dokumenten nicht getrennt. Allerdings steht eine Vielzahl an Elementen bereit, mit der sich eine Silbentrennung realisieren lässt. Bei der Silbentrennung handelt es sich zunächst einmal nicht direkt um ein typografisches Merkmal. Allerdings lässt sich damit durchaus das Schriftbild beeinflussen. (Und das sowohl im positiven wie auch im negativen Sinn.)

- `hyphenate` – Bestimmt, ob eine Silbentrennung stattfinden soll oder nicht. Mögliche Werte sind `true` (findet statt) und `false` (findet nicht statt).
- `hyphenation-character` – Hierüber wird das Trennzeichen festgelegt. Als Standardzeichen wird der herkömmliche Trennstrich verwendet.
- `hyphenation-keep` – Die Silbentrennung wird am Ende einer Spalte oder einer Seite durchgeführt. Durch die Werte `column` und `page` kann die Silbentrennung am Ende einer Spalte oder einer Seite verhindert werden.
- `hyphenation-ladder-count` – Damit wird die Anzahl der maximal aufeinanderfolgenden Zeilen angegeben, für die die Silbentrennung erlaubt sein soll.
- `hyphenation-push-character-count` – Gibt für ein Wort die Mindest-Buchstabenzahl an, die nach dem Trennzeichen in der nächsten Zeile stehen muss. Der Standardwert ist `2`.
- `hyphenation-remain-character-count` – Gibt für ein Wort die Mindest-Buchstabenzahl an, die vor dem Trennzeichen stehen muss. Der Standardwert ist `2`.
- `country` – Hier wird das Land angegeben, dessen Regeln bei der Silbentrennung berücksichtigt werden sollen. Für Deutschland geben Sie `de` an.
- `language` – Darüber gibt man die verwendete Sprache an. Für Deutschland notieren Sie `de`.

Ein Beispiel, wie sich die verschiedenen Attribute kombinieren lassen.

Listing 8.51 Die Silbentrennung wird festgelegt.

```
<fo:block hyphenate="true"
   hyphenation-character="!"
   hyphenation-push-character-count="2"
   hyphenation-remain-character-count="3"
   language="de">
      Bitte nur seeeeeeeeeeeeeeeeehr lange Wörter trennen
<fo:block>
```

Die Trennung funktioniert nur, wenn die Sprache des Textes angegeben wurde. Das geschieht entweder im `fo:block`-Element selbst, in einem ihm hierarchisch übergeordneten `fo:block`-Element oder innerhalb von `fo:page-sequence`.

8.6.9 Groß- und Kleinschreibung

Über das Attribut `text-transform` kann die Umwandlung von Groß- und Kleinbuchstaben erreicht werden.

- `capitalize` – Wandelt den ersten Buchstaben jedes Wortes in einen Großbuchstaben um.
- `uppercase` – Wandelt alle Kleinbuchstaben in Großbuchstaben um.
- `lowercase` – Wandelt alle Großbuchstaben in Kleinbuchstaben um.
- `none` – Dabei handelt es sich um die Standardeinstellung. Die vorhandenen Groß- und Kleinbuchstaben werden beibehalten.

Ein Beispiel dazu:

Listing 8.52 Unterschiedliche Varianten der Buchstabenanzeige

```
<fo:block>
   <fo:inline text-transform="uppercase">
      kleinbuchstaben
   </fo:inline>
   <fo:inline text-transform="capitalize">
      kleinbuchstaben
   </fo:inline>
</fo:block>
```

Hier wird das Wort `kleinbuchstaben` einmal mit `uppercase` und einmal mit `capitalize` ausgestattet. Das Ergebnisdokument sieht folgendermaßen aus:

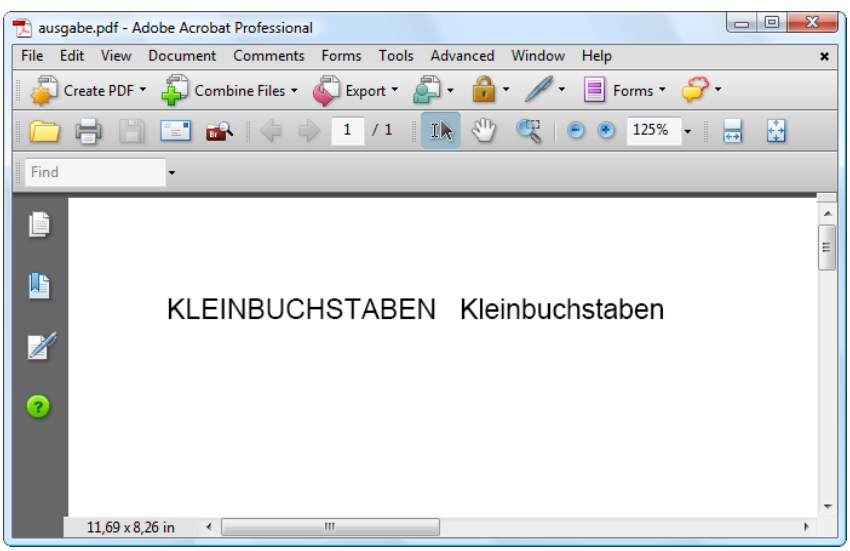

Abbildung 8.26 Klein- und Großbuchstaben werden angezeigt.

8.7 Hyperlinks und Querverweise setzen

Selbstverständlich können Sie auch Hyperlinks definieren. Dabei sind sowohl interne wie auch externe Links möglich. Wie die Links letztendlich definiert werden, hängt von dem Ausgabemedium ab. So erfüllen Hyperlinks beispielsweise in einem PDF-Dokument normalerweise einen anderen Zweck als in einem Print-Produkt.

Das zentrale Element für die Definition von Hyperlinks ist `fo:basic-link`.

Um innerhalb eines Dokuments verweisen zu können, wird eine eindeutige Zieldefinition benötigt. Daher muss jeder Zielpunkt mit einer ID versehen werden. Das dafür zuständige `id`-Attribut lässt sich innerhalb jedes Blockelements verwenden.

Das `fo:basic-link`-Element kennt insgesamt drei Attribute, darunter die zwei:

- `internal-destination` – dokumentinternes Verweisziel
- `external-destination` – externes Verweisziel

Zusätzlich gibt es noch das Attribut `show-destination`. Damit gibt man an, wo das Verweisziel geöffnet werden soll.

- `replace` – Das Verweisziel wird im gleichen Fenster geöffnet.
- `new` – Es wird ein neues Fenster geöffnet, in dem das Verweisziel angezeigt wird.

Zunächst ein Beispiel für einen dokumentinternen Verweis. Dabei muss als Erstes das Verweisziel über das `id`-Attribut angegeben werden.

```
<head id="anfang">Herzlich Willkommen!</head>
```

Damit sind die Vorarbeiten abgeschlossen, und der Verweis kann dementsprechend gesetzt werden.

Listing 8.53 So werden Hyperlinks definiert.

```
<fo:basic-link internal-destination="anfang"
text-decoration="underline">
   Zum Anfang
</fo:basic-link>
```

Neben der gezeigten Variante lassen sich auch noch Hyperlinks auf externe Ressourcen definieren. Verwendet wird dafür ebenfalls das Element `fo:basic-link`. Als Attribut verwendet man hier `external-destination`. Diesem Attribut weist man die gewünschte Zielressource zu.

Listing 8.54 Ein Link wird definiert.

```
<fo:basic-link external-destination="url('http://www.hanser.de/')"
               text-decoration="underline"
               color="blue">Hanser</fo:basic-link>
```

Die gezeigte Syntax funktioniert in dieser Form. Ebenso kann aber auch `url()` weggelassen werden. Das Ganze sähe dann folgendermaßen aus:

Listing 8.55 Diese Syntax funktioniert auch.

```
<fo:basic-link external-destination="http://www.hanser.de/"
               text-decoration="underline"
               color="blue">Hanser</fo:basic-link>
```

Und jetzt noch ein Blick auf das Ergebnisdokument.

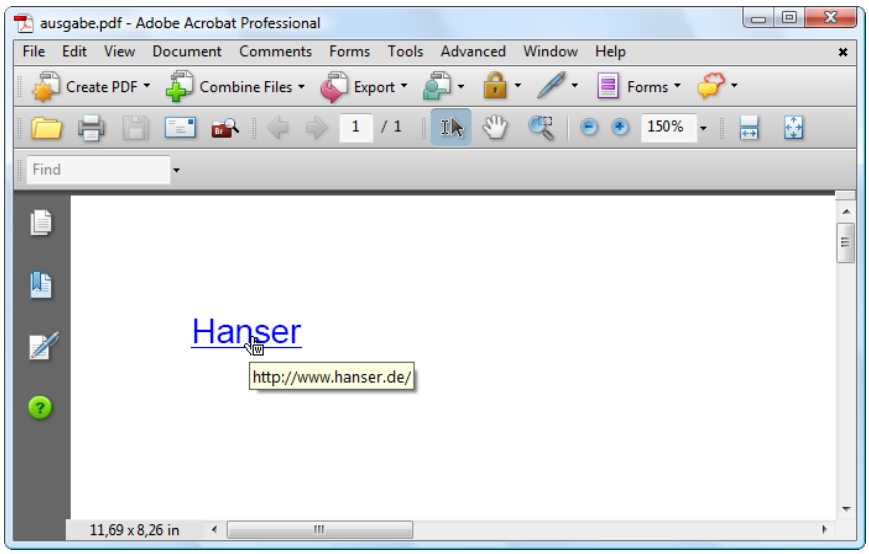

Abbildung 8.27 Der Link funktioniert.

Standardmäßig werden Hyperlinks in blauer Schriftfarbe angezeigt. Um ihnen zusätzlich den typischen Unterstrich zuzuweisen, muss man `text-decoration="underline"` verwenden.

Klickt ein Anwender beispielsweise im Adobe Acrobat auf einen solchen Link, wird ein entsprechender Warnhinweis angezeigt.

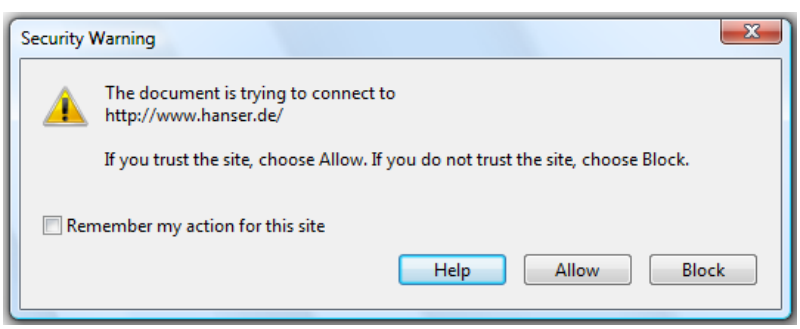

Abbildung 8.28 Das Ausführen des Links muss bestätigt werden.

Erst wenn man diesen bestätigt, wird das Verweisziel geöffnet.

8.8 Listen und Aufzählungen

Für die übersichtliche Anordnung von Elementen bieten sich oftmals Listen und Aufzählungen an. In XSL-FO lässt sich so etwas recht einfach umsetzen. Dabei sind ganz unterschiedliche Listenarten möglich.

- Geordnete Listen (Jedes Element bekommt eine Ordnungsbezeichnung. Üblicherweise ist das eine Ziffer.)
- Ungeordnete Listen (Jedem Element wird ein Vorzeichen zugewiesen. Das ist oftmals ein Bindestrich oder ein Punkt.)
- Beschreibungslisten (Jedem Element wird eine den Inhalt des jeweiligen Elements beschreibende Bezeichnung zugewiesen.)

XSL-FO hat also auch hinsichtlich der Listengestaltung einiges zu bieten. Dabei sieht das Konzept vor, dass sich Listen aus vier Elementen zusammensetzen.

- `fo:list-block` – Definiert die Liste.
- `fo:list-item` – Definiert das Listenelement.
- `fo:list-item-label` – Weist dem Listenelement ein Label bzw. Etikett zu.
- `fo:list-item-body` – Gibt den Inhalt des Listenelements an.

Innerhalb von `fo:list-item-label` und `fo:list-item-body` werden die normalen Blockelemente definiert. Wie eine typische Definition einer Liste in XSL-FO aussehen kann, zeigt das folgende Beispiel:

Listing 8.56 Eine Liste wird angelegt.

```
<fo:block>
   <fo:list-block>
      <fo:list-item>
   <fo:list-item-label>
      <fo:block>
         1.
      </fo:block>
   </fo:list-item-label>

   <fo:list-item-body>
      <fo:block>
         Element
      </fo:block>
   </fo:list-item-body>
</fo:list-item>

<fo:list-item>
   <fo:list-item-label>
      <fo:block>
         2.
      </fo:block>
   </fo:list-item-label>

   <fo:list-item-body>
      <fo:block>
         Element
      </fo:block>
   </fo:list-item-body>
</fo:list-item>
   </fo:list-block>
</fo:block>
```

Den äußeren Rahmen von Listen bildet das Blockelement `fo:list-block`. Für jedes einzelne Listenelement muss innerhalb der nächsten Schachtelungsebene ein `fo:list-item` definiert werden. Dieses ist wiederum in `fo:list-item-label` für das Aufzählungszeichen und in `fo:list-item-body` für den Inhalt unterteilt.

Die letzte Ebene, noch bevor der eigentliche Inhalt eingefügt wird, sind ein oder mehrere Blockelemente wie z.B. `fo:block` oder `fo:flock-container`.

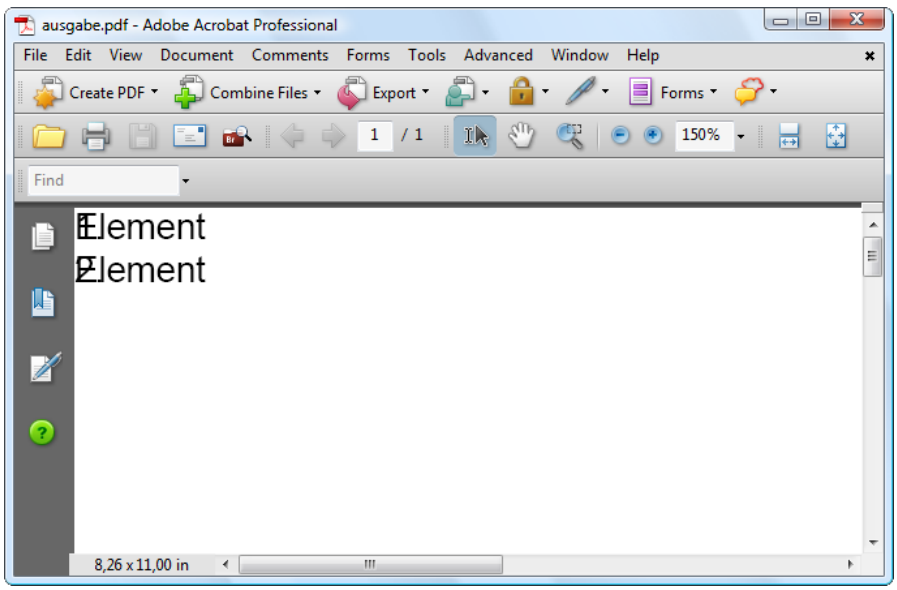

Abbildung 8.29 Die Liste wird nicht wie gewünscht angezeigt.

In der gezeigten Form ist die Liste noch nicht wirklich praxistauglich. Es fehlen entsprechende Abstandsdefinitionen. Wie man diese anlegt, wird im nächsten Abschnitt beschrieben.

8.8.1 Abstände innerhalb von Listen

Für die optische Anpassung von Listen stellt XSL:FO verschiedene Attribute zur Verfügung, über die sich die Abstände innerhalb der Listen festlegen lassen. Zunächst ein Blick auf die entsprechenden Attribute.

- `provisional-distance-between-starts` – Hierüber wird der Abstand zwischen dem Beginn des Listenelement-Etiketts und dem Beginn des Inhalts des Listenelements bestimmt. Angegeben wird dieser Abstand in einem absoluten Wert in einer der möglichen Maßeinheiten.
- `provisional-label-separation` – Um den Abstand zwischen dem Ende des Listenelement-Etiketts und dem Anfang des Listenelement-Inhalts anzugeben, wird dieser Abstand in einem absoluten Wert in einer der möglichen Maßeinheiten notiert.
- `end-indent="label-end()"` – Dieses Attribut bestimmt den Abstand zwischen dem Ende des Listenelement-Etiketts und dem rechten Rand. Über die Funktion `label-end()` wird dieser Abstand so ermittelt, dass auch die Werte von `provisional-`

`distance-between-starts` und `provisional-label-separation` mit einbezogen werden. Anstelle von `label-end()` kann man auch einen Wert angeben.

- `start-indent="body-start()"` – Hierüber wird der Abstand zwischen dem Ende des Listenelement-Etiketts und dem Anfang des Listenelement-Inhalts bestimmt. Über die Funktion `body-start()` wird dabei der betreffende Abstand ermittelt, wobei auch hier die Werte von `provisional-distance-between-starts` und `provisional-label-separation` mit einbezogen werden. Anstelle von `body-start()` kann man auch einen Wert angeben.

Hierzu ebenfalls ein Beispiel:

Listing 8.57 Jetzt werden Abstände eingefügt.

```
<fo:list-block>

   <fo:list-item>
      <fo:list-item-label>
         <fo:block>a)</fo:block>
   </fo:list-item-label>

   <fo:list-item-body start-indent="body-start()">
      <fo:block>Erster Eintrag</fo:block>
   </fo:list-item-body>
   </fo:list-item>

   <fo:list-item>
      <fo:list-item-label>
         <fo:block>b)</fo:block>
      </fo:list-item-label>

      <fo:list-item-body start-indent="body-start()">
         <fo:block>Zweiter Eintrag</fo:block>
      </fo:list-item-body>
   </fo:list-item>

   <fo:list-item>
      <fo:list-item-label>
         <fo:block>c)</fo:block>
      </fo:list-item-label>
      <fo:list-item-body start-indent="body-start()">
         <fo:block>Dritter Eintrag</fo:block>
   </fo:list-item-body>
</fo:list-item>

</fo:list-block>
```

Wie auf **Abbildung 8.30** zu sehen ist, sieht die Liste nun wie eine richtige Liste aus.

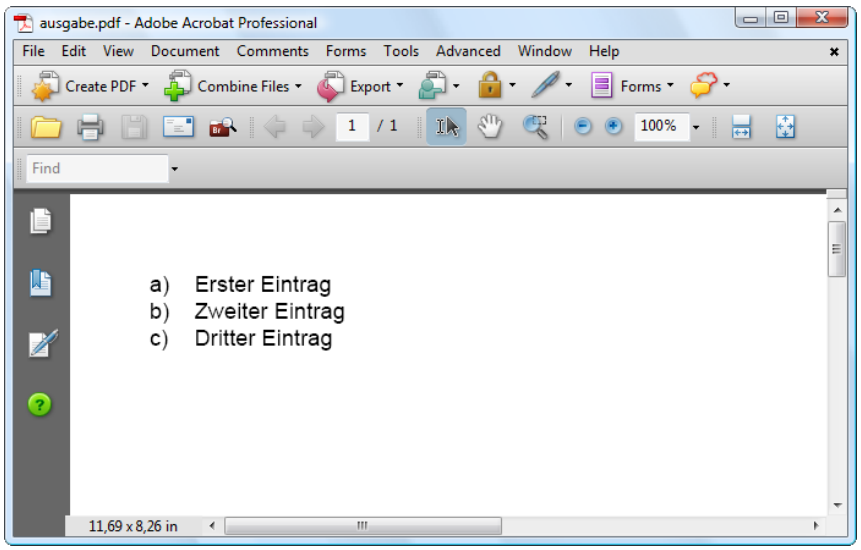

Abbildung 8.30 Jetzt wird die Liste korrekt angezeigt.

8.9 Fußnoten

Auch Fußnoten lassen sich einfügen. Verwendet wird dafür das Element `fo:footnote`. Bevor gezeigt wird, wie man solche Fußnoten anlegt, zunächst ein Blick darauf, wie eine solche Fußnoten aussieht.

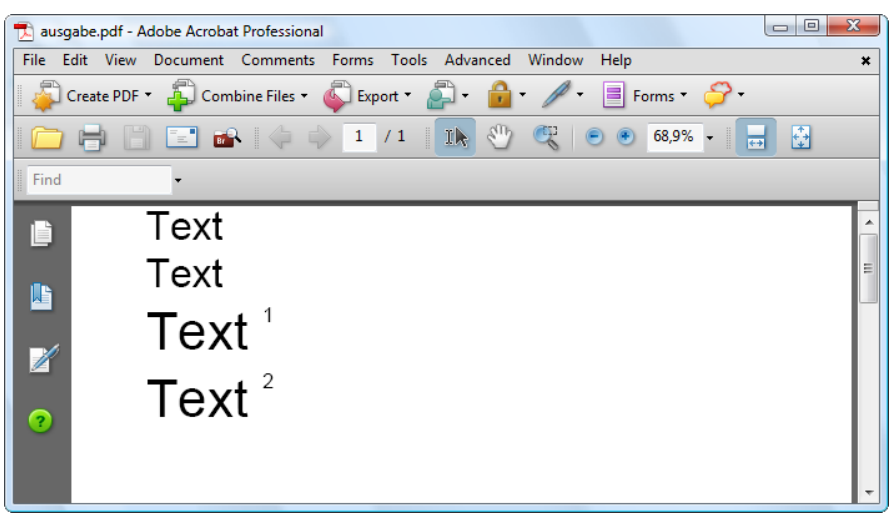

Abbildung 8.31 Die kleinen Ziffern kennzeichnen Fußnoten.

Im Fließtext werden kleine Ziffern angezeigt. Die Fußnoten selbst sind dann im unteren Seitenbereich des Dokuments zu sehen.

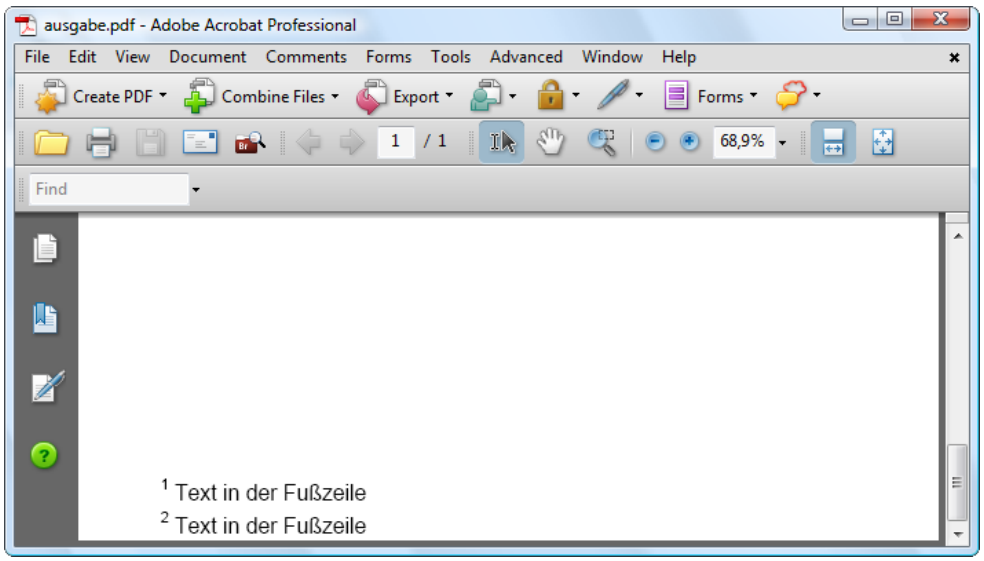

Abbildung 8.32 Und so sehen die Fußnoten letztendlich aus.

Wie sich das gezeigte Beispiel anlagen lässt, zeigt die folgende Syntax:

Listing 8.58 So werden Fußnoten definiert.

```
<fo:block font-size="30pt"> Text </fo:block>
<fo:block font-size="30pt"> Text </fo:block>
<fo:block font-size="40pt">
 Text
 <fo:footnote
  <fo:inline font-size="15pt" baseline-shift="super">1</fo:inline>
  <fo:footnote-body>

   <fo:block font-size="15pt">
    <fo:inline font-size="10pt" baseline-shift="super">1</fo:inline>
    Text in der Fußzeile
   </fo:block>
  </fo:footnote-body>
 </fo:footnote>
</fo:block>

<fo:block font-size="40pt">
 Text
 <fo:footnote>
  <fo:inline font-size="15pt" baseline-shift="super">2</fo:inline>
  <fo:footnote-body>

   <fo:block font-size="15pt">
    <fo:inline font-size="10pt" baseline-shift="super">2</fo:inline>
    Text in der Fußzeile
   </fo:block>

  </fo:footnote-body>
 </fo:footnote>
</fo:block>
```

Das Element `fo:footnote` leitet die Definition einer Fußnote ein. Um die Fußnotenziffern so aussehen zu lassen, wie Fußnoten eben aussehen sollten, verwendet man üblicherweise eine Kombination aus `fo:inline` und `baseline-shift`. Dadurch wird eine inzeilige und

hochgestellte Anzeige erreicht. Der eigentliche Inhalt der Fußzeile steht innerhalb des Elements `fo:footnote-body`.

Bei einem Blick auf das vorherige Ergebnis fällt auf, dass etwas Entscheidendes fehlt. Denn bislang wird die Fußnote nicht genügend vom übrigen Seiteninhalt abgetrennt. Um das zu erreichen, wird in herkömmlichen Publikationen auf Linien gesetzt, die Inhalt und Fußnote trennen. Wie so etwas aussehen kann, zeigt **Abbildung 8.33**.

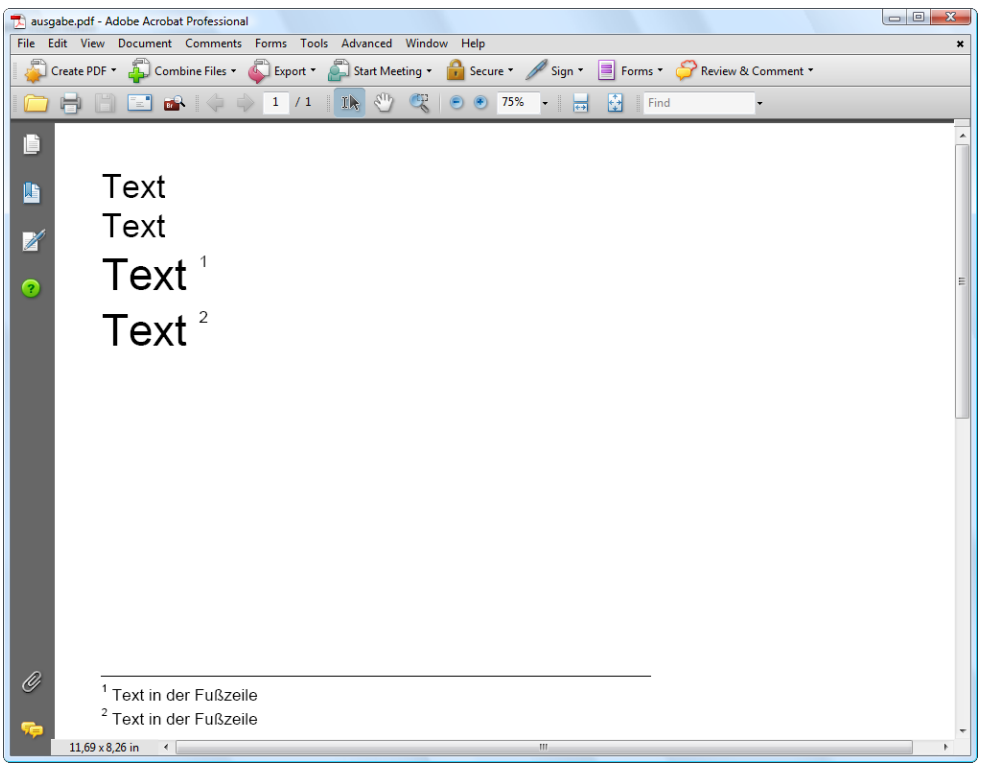

Abbildung 8.33 Jetzt sieht es noch mehr nach Fußnoten aus.

So sehen die Fußnoten deutlich ansprechender aus. Verantwortlich für diese Anzeige ist der Wert `xsl-footnote-separator` des Attributs `flow-name`.

Die Linie wird eingefügt.

```
<fo:static-content flow-name="xsl-footnote-separator">
   <fo:block text-align-last="justify">
      <fo:leader leader-length="70%" rule-thickness="0.7pt"
      leader-pattern="rule"/>
   </fo:block>
</fo:static-content>
```

Für die eigentliche Linie wird auf `fo:leader` zurückgegriffen. Beachten Sie, dass die gezeigte Syntax nicht in allen Formatierern funktioniert.

8.10 Grafiken einbinden

Selbstverständlich lassen sich auch Grafiken einbinden. Verwendet wird dafür das Element `fo:external-graphic`. Als Attribut muss zumindest `src` angegeben werden. Diesem Attribut weist man den Pfad zur und den Namen der Grafik zu, die angezeigt werden soll. Hier gelten die gleichen Regeln, die vom Einbinden von Grafiken aus HTML her bekannt sind.

Gibt man nur `src` an, wird das Bild in seiner Originalgröße angezeigt. Anhand zweier Attribute können Sie Einfluss auf die Bildergröße nehmen.

- `content-width` – Hierüber gibt man die Breite der Grafik an. Als Standardwert wird `auto` angenommen. Dadurch wird die Grafik in ihrer Originalbreite angezeigt. Durch `scale-to-fit` wird die Grafik an die Größe eines vorgegebenen Rechtecks angepasst. Ebenso sind auch prozentuale und absolute Breitenangaben möglich.
- `content-height` – Durch dieses Attribut legt man die Höhe der Grafik fest. Ansonsten gilt das Gleiche, was zuvor im Zusammenhang mit `content-width` beschrieben wurde.

Wenn man lediglich für eine Dimension eine Größenangabe macht, wird die andere Dimension proportional angepasst.

Listing 8.59 Die gleiche Grafik wird drei Mal eingebunden.

```
<fo:block>

<fo:inline>
  <fo:external-graphic src="image.jpg"/>
 </fo:inline>

 <fo:inline>
   <fo:external-graphic src="image.jpg" content-width="40mm"/>
 </fo:inline>

 <fo:inline>
  <fo:external-graphic src="image.jpg" content-width="40mm"
  content-height="40mm"/>
 </fo:inline>

</fo:block>
```

Abbildung 8.34 zeigt, wie die Grafiken jeweils eingebunden werden.

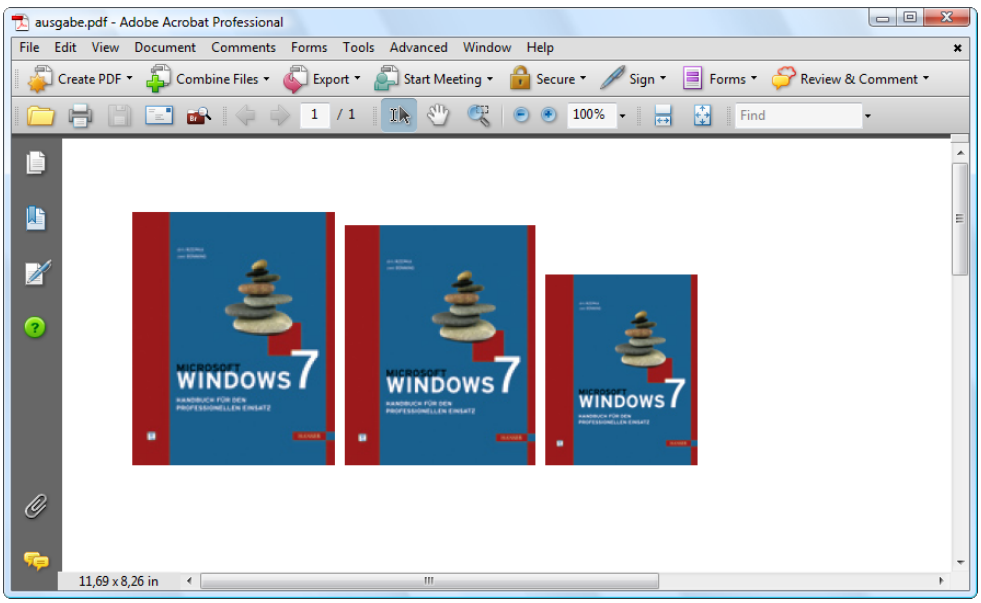

Abbildung 8.34 Die gleiche Grafik, aber in unterschiedlichen Größen

Um Verzerrungen zu vermeiden, sollte man in jedem Fall die Maße exakt angeben oder nur die Breite oder nur die Höhe des Bildes definieren.

Die Grafik muss korrekt referenziert wird, da der Formatierer ansonsten die Generierung des Dokuments mit einer Fehlermeldung abbricht.

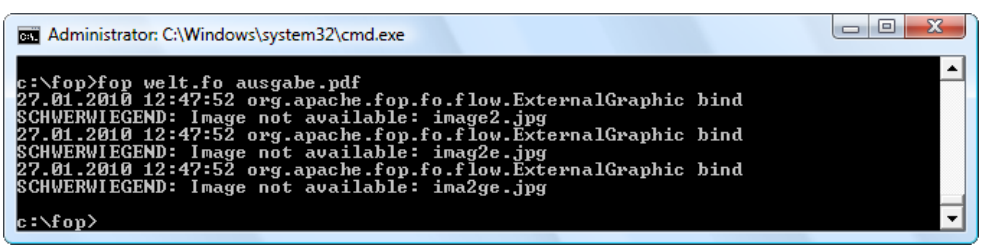

Abbildung 8.35 Die Grafiken wurden nicht gefunden.

Hier konnte die Grafik nicht eingebunden werden.

8.10.1 Hintergrundbilder definieren

Auch Hintergrundbilder lassen sich einbinden. Für diesen Zweck stellt die Spezifikation zahlreiche Attribute zur Verfügung, die Ihnen so vielleicht bereits von CSS her bekannt sind. Bevor die Attribute aber im Einzelnen vorgestellt werden, ein einfaches Beispiel, in dem eine Hintergrundgrafik eingebunden wird.

Listing 8.60 Die Hintergrundgrafik wird eingebunden.

```
<fo:block background-image="background.jpg"
background-repeat="x-repeat" font-size="40pt">
   Herzlich Willkommen
</fo:block>
```

Und jetzt die verfügbaren Attribute.

- `background-image` – Dieses Attribut erwartet als Wert den Pfad zur und den Namen der Grafik.
- `background-repeat` – Hierüber wird festgelegt, ob und wie die Hintergrundgrafik wiederholt werden soll. Der Vorgabewert ist `repeat`. Dadurch wird die Grafik vertikal und horizontal so oft wiederholt, bis der gesamte Platz, für den die Grafik gelten soll, ausgefüllt ist. Durch `x-repeat` erfolgt, wenn es nötig ist, eine horizontale Wiederholung der Grafik. Setzt man hingegen `y-repeat` ein, wird die Grafik vertikal wiederholt. Will man die Wiederholung sowohl vertikal wie auch horizontal unterbinden, setzt man `no-repeat` ein. Beachten Sie, dass FOP die beiden Werte `x-repeat` und `y-repeat` nicht korrekt interpretiert.
- `background-position-vertical` – Über dieses Attribut wird die vertikale Positionierung innerhalb des Block-Rechtecks ggf. inklusive der `padding`-Fläche angegeben. Der Vorgabewert ist `0%`, der am oberen Rand des Rechtecks beginnt. Durch andere Werte kann man das Hintergrundbild innerhalb des Rechtecks verschieben. Neben numerischen Werten sind auch noch `top` (entspricht `0%`), `center` (entspricht `50%`) und `bottom` (entspricht `100%`) möglich.
- `background-position-horizontal` – Das ist das Gegenstück zu `background-position-vertical`. Allerdings wird bei `background-position-horizontal` die horizontale Positionierung innerhalb des gegebenen Block-Rechtecks bestimmt. Auch hier sind wieder numerische Angaben oder `top`, `center` und `bottom` möglich.
- `background-position` – Dabei handelt es sich um eine verkürzte Syntax für die Positionierung der Hintergrundgrafik. Der erste Wert entspricht `background-position-vertical`, der zweite `background-position-horizontal`.
- `background-color` – Hiermit geben Sie die Hintergrundfarbe an. Interessant ist diese z.B., wenn die definierte Hintergrundgrafik – aus welchem Grund auch immer – nicht angezeigt werden kann.

Mit dem Einsatz von Hintergrundgrafiken sollte man sehr sparsam umgehen. Das gilt insbesondere, wenn die Grafik hinter Text gelegt werden soll. Oftmals ist dann der eigentliche Text nämlich nur noch schwer oder überhaupt nicht mehr zu lesen.

Abbildung 8.36 zeigt eindrucksvoll, wie man es nicht machen sollte.

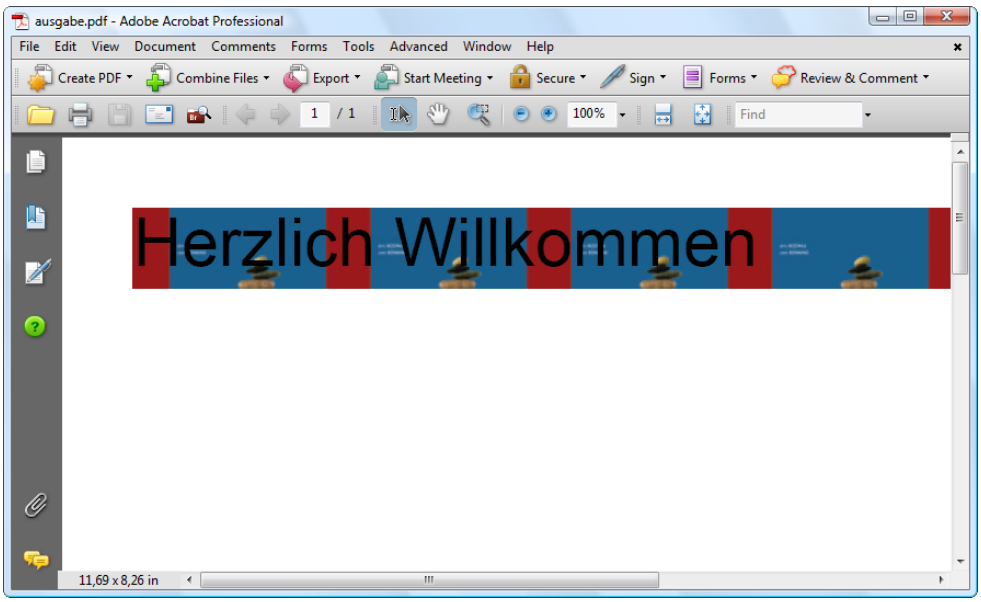

Abbildung 8.36 Hier kann man den Text kaum lesen.

8.10.2 SVG einbinden

Die Formatting Objects wurden für die Anzeige von Textdokumenten entwickelt. Es besteht allerdings auch die Möglichkeit, Grafiken, die im Binärformat vorliegen, einzubinden. Der entsprechende Elementtyp lautet `external-graphic`. Wie dieser einzusetzen ist, wurde auf den vorherigen Seiten beschrieben.

Anders sieht es aber aus, wenn Grafiken eingebunden werden sollen, die in XML-Formaten vorliegen. Ein typisches Anwendungsbeispiel dafür ist SVG. Aber auch solche Grafiken können eingebunden werden. Dabei wird allerdings der entsprechende XML-Quelltext in das XSL-FO-Dokument übernommen. Zum Einsatz kommt hier das Element `instream-foreign-object`. Die Kinder eines solchen Elements kommen aus anderen Namensräumen als dem XSL-Namensraum. Welche Formate letztendlich unterstützt werden, hängt vom verwendeten XSL-Prozessor ab.

Integrieren lassen sich z.B. SVG-Grafiken. Wie das funktioniert, zeigt das folgende Beispiel. Zum Einbinden der XML- bzw. SVG-Syntax wird `fo:instream-foreign-object` verwendet.

Listing 8.61 SVG wird integriert.

```
<?xml version="1.0" encoding="UTF-8"?>
<fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format" >
     <fo:layout-master-set>
          <fo:simple-page-master master-name="simple">
               <fo:region-body/>
          </fo:simple-page-master>
     </fo:layout-master-set>
```

```
        <fo:page-sequence master-reference="simple">
             <fo:flow flow-name="xsl-region-body">
                  <fo:block>

                       <fo:instream-foreign-object>
                            <svg:svg width="200pt"
                            height="100pt"
                            xmlns:svg="http://www.w3.org/2000/svg" >

                                 <svg:circle cx="60pt"
                                 cy="50pt"
                                 r="50pt"
                                 style="fill:red;"/>
                                 </svg:svg>


                       </fo:instream-foreign-object>
                  </fo:block>
             </fo:flow>
        </fo:page-sequence>
</fo:root>
```

Innerhalb des `fo:instream-foreign-object`-Elements steht herkömmliche SVG-Syntax. Im aktuellen Beispiel wird einfach ein roter Kreis generiert. Das Ergebnis sieht folgendermaßen aus:

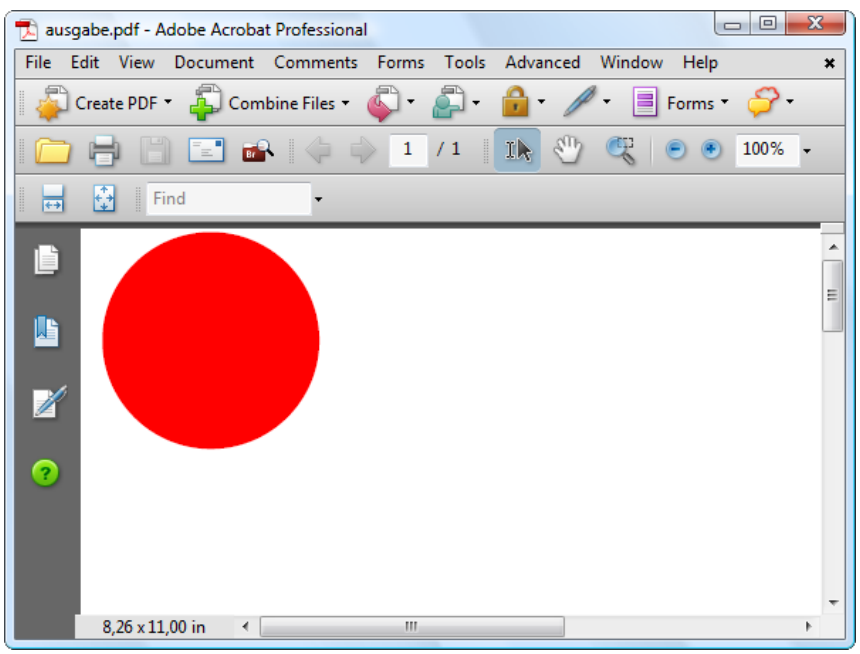

Abbildung 8.37 Ein SVG-Element wurde integriert.

Die Sache hat natürlich einen entscheidenden Haken. Animationen, wie sie sonst mittels SVG möglich sind, lassen sich in PDF- oder anderen „statischen“ Elementen leider nicht realisieren.

FOP erlaubt es übrigens, dass Sie Ihre normalen XSL-FO-Dokumente in das SVG-Format umwandeln. Dafür muss bei FOP lediglich das Ausgabeformat SVG angegeben werden.

8.11 Mit Tabellen arbeiten

Tabellen kennen Sie aus Textverarbeitungsprogrammen und z.B. aus HTML. Auch in XSL-FO lassen sich Tabellen definieren. Die Tabellenstruktur in XSL-FO ist allerdings nicht mit der aus HTML vergleichbar.

Listing 8.62 Eine einfache Tabelle wird angelegt.

```
<fo:table>
 <fo:table-body>
  <fo:table-row>

   <fo:table-cell>
    <fo:block>Zelle 1</fo:block>
   </fo:table-cell>

   <fo:table-cell>
    <fo:block>Zelle 2</fo:block>
   </fo:table-cell>

   <fo:table-cell>
    <fo:block>Zelle 3</fo:block>
   </fo:table-cell>

   <fo:table-cell>
    <fo:block>Zelle 4</fo:block>
   </fo:table-cell>

  </fo:table-row>
 </fo:table-body>
</fo:table>
```

Für die Definition von Tabellen sind mehrere Elemente zuständig.

- `fo:table` – Hierüber wird die Tabelle erzeugt.
- `fo:table-header` – Erzeugt den Kopfbereich der Tabelle. Dieser Kopfbereich wird, wenn sich die Tabelle über mehrere Seiten erstreckt, wiederholt.
- `fo:table-body` – Darin wird der eigentliche Inhalt der Tabelle definiert.
- `fo:table-footer` – Hierüber wird der Fußbereich der Tabelle erzeugt.
- `fo:table-and-caption` – Wenn die Tabelle eine Überschrift oder Legende besitzt, handelt es sich bei diesem Element um das oberste und obligatorische Tabellenelement.
- `fo:table-caption` – Nimmt die Überschrift oder Legende auf, die als oberstes Element `fo:table-and-caption` besitzt.
- `fo:table-row` – Darüber werden Tabellenzeilen definiert.
- `fo:table-cell` – Hierüber bestimmt man die Tabellenzellen.

Abbildung 8.38 zeigt das Ergebnis der zuvor definierten Syntax.

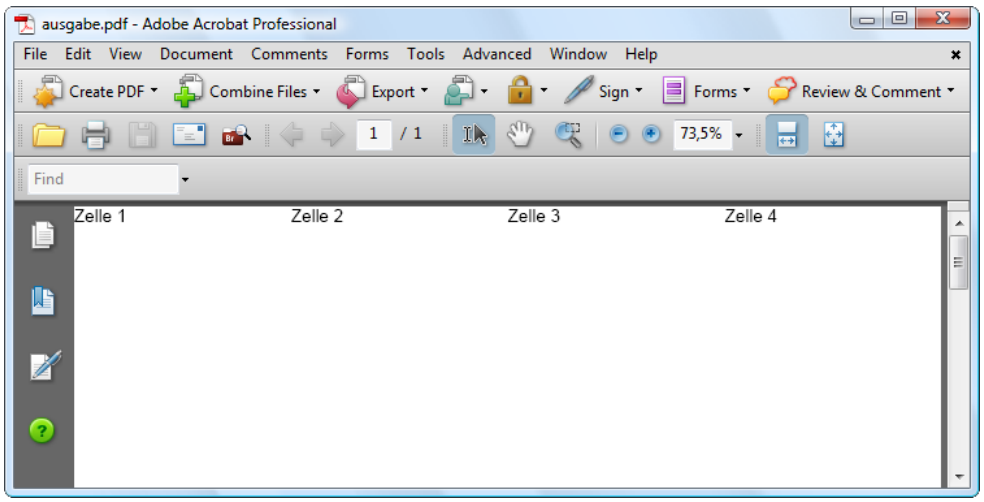

Abbildung 8.38 Eine sehr einfache Tabelle wurde angelegt.

Es handelt sich hier um eine sehr einfache Tabelle. Tabellendefinitionen lassen sich aber durch zahlreiche Attribute erweitern. So kann man u.a. Rahmen, Abstände und Farben einsetzen.

Listing 8.63 Die Tabelle wurde erweitert.

```
<fo:block>
<fo:table width="160mm" border-style="ridge" border-width="5pt">
 <fo:table-body>
  <fo:table-row>

   <fo:table-cell width="40mm" border-style="solid"
   border-width="2pt">
    <fo:block>Zelle 1</fo:block>
   </fo:table-cell>

   <fo:table-cell width="40mm" border-style="solid"
   border-width="2pt">
    <fo:block>Zelle 2</fo:block>
   </fo:table-cell>

   <fo:table-cell width="40mm" border-style="solid"
   border-width="2pt">
    <fo:block>Zelle 3</fo:block>
   </fo:table-cell>

   <fo:table-cell width="40mm" border-style="solid"
   border-width="2pt">
    <fo:block>Zelle 4</fo:block>

   </fo:table-cell>
  </fo:table-row>
  <fo:table-row>

   <fo:table-cell border-style="solid" border-width="1pt">
    <fo:block>Zelle 5</fo:block>
   </fo:table-cell>

   <fo:table-cell border-style="solid" border-width="1pt">
    <fo:block>Zelle 6</fo:block>
   </fo:table-cell>
```

```
    <fo:table-cell border-style="solid" border-width="1pt">
     <fo:block>Zelle 7</fo:block>
    </fo:table-cell>

    <fo:table-cell border-style="solid" border-width="1pt">
     <fo:block>Zelle 8</fo:block>
    </fo:table-cell>

   </fo:table-row>
  </fo:table-body>
 </fo:table>
 </fo:block>
```

Wie das Ergebnis dieser Syntax aussieht, zeigt **Abbildung 8.39**.

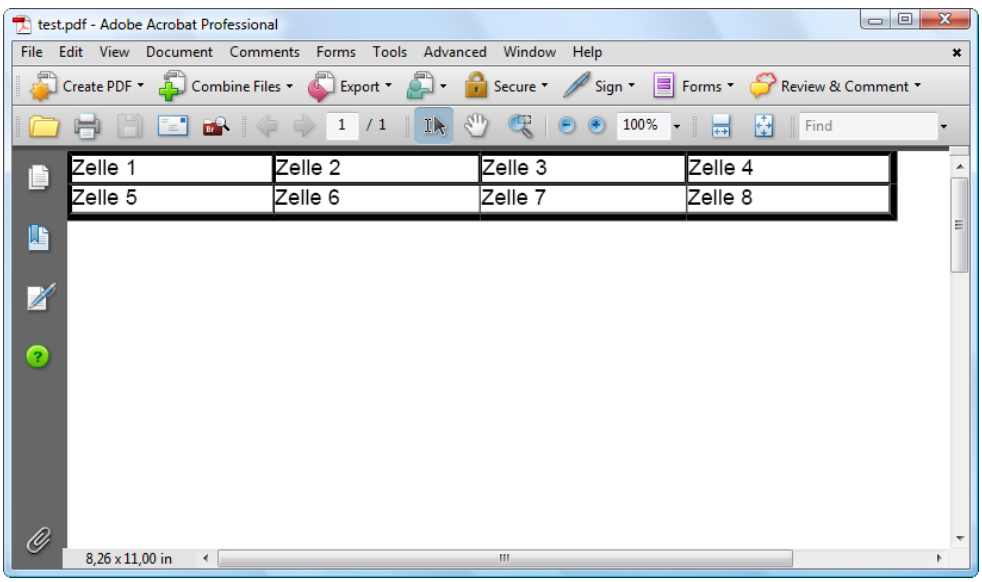

Abbildung 8.39 Das sieht schon eher nach einer Tabelle aus.

8.11.1 Zellen und Zeilen überspannen

In den bisherigen Beispielen wurde noch nicht die Möglichkeit des Überspannens von Spalten und Zeilen genutzt.

Mit dem Attribut `number-columns-spanned` bei `fo:table-column` und bei `fo:table-cell` wird das Überspannen von Spalten realisiert. Will man hingegen Zeilen überspannen, verwendet man das Attribut `number-rows-spanned` bei `fo:table-cell`.

Das folgende Beispiel zeigt, wie sich diese Elemente einsetzen lassen.

Listing 8.64 Eine etwas aufwendigere Tabelle

```
<fo:block>
<fo:table width="110mm" border-style="outset" border-width="1pt">
<fo:table-column column-number="1" column-width="25%"
border-style="solid" border-width="1pt"/>
<fo:table-column column-number="2" column-width="25%"
border-style="solid" border-width="1pt"/>
<fo:table-column column-number="3" column-width="25%"
```

```
border-style="solid" border-width="1pt"/>
<fo:table-column column-number="4" column-width="25%"
border-style="solid" border-width="1pt"/>

<fo:table-header>
   <fo:table-row>

      <fo:table-cell number-columns-spanned="4">
         <fo:block>Kopfzeile</fo:block>
      </fo:table-cell>

   </fo:table-row>
</fo:table-header>

<fo:table-footer>
   <fo:table-row>

      <fo:table-cell number-columns-spanned="4">
         <fo:block> Fusszeile</fo:block>
      </fo:table-cell>

   </fo:table-row>
</fo:table-footer>

<fo:table-body>
   <fo:table-row>

      <fo:table-cell column-number="1">
         <fo:block>1. Zelle</fo:block>
      </fo:table-cell>

      <fo:table-cell column-number="2">
         <fo:block>2. Zelle</fo:block>
      </fo:table-cell>

      <fo:table-cell column-number="3" number-columns-spanned="2">
         <fo:block>3. Zelle</fo:block>
      </fo:table-cell>

   </fo:table-row>

   <fo:table-row>

      <fo:table-cell column-number="1" number-columns-spanned="4">
         <fo:block>Mittelteil</fo:block>
      </fo:table-cell>

   </fo:table-row>

   <fo:table-row>

      <fo:table-cell column-number="1" number-columns-spanned="2">
         <fo:block>1. Zelle</fo:block>
      </fo:table-cell>

      <fo:table-cell column-number="3">
         <fo:block>2. Zelle</fo:block>
      </fo:table-cell>

      <fo:table-cell column-number="4">
         <fo:block>3. Zelle</fo:block>
      </fo:table-cell>

   </fo:table-row>
</fo:table-body>
</fo:table>
</fo:block>
```

Und auch hier natürlich wieder das Ergebnisdokument.

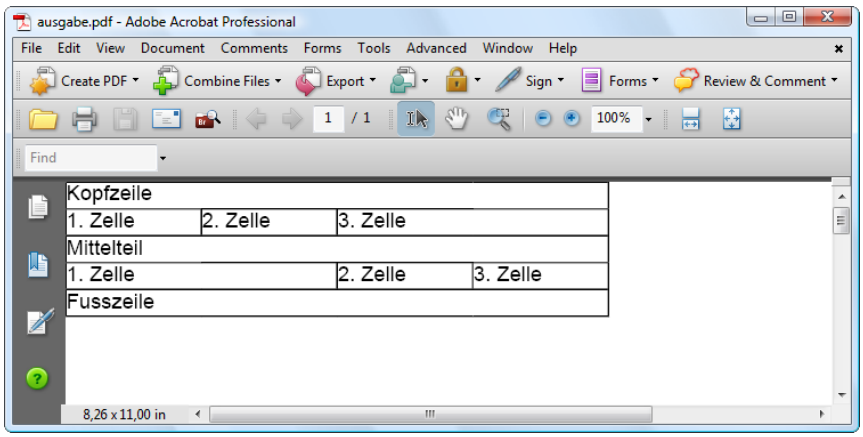

Abbildung 8.40 Zellen wurde verbunden.

8.12 Das Float-Konzept

Den Namen “Float-Konzept” haben Sie vielleicht bereits im Zusammenhang mit CSS gehört. Einfach gesagt, geht es dabei um das Fließverhalten von Elementen. Wie soll sich also z.B. ein Text verhalten, in dessen Mitte plötzlich eine Grafik erscheint? Soll der Text nun links oder rechts um die Grafik fließen oder erst unter dem Bild anfangen?

Im Zusammenhang mit XSL-FO wird das Float-Konzept allerdings noch in einem anderen Zusammenhang eingesetzt. Denn da es in XSL-FO keine andere Möglichkeit gibt, kommt das Float-Konzept auch für die Generierung von Marginalien zum Einsatz.

Das Float-Konzept bzw. `fo:float` ist längst noch nicht in allen Formatierern implementiert. So meldet sich FOP beispielsweise folgendermaßen:

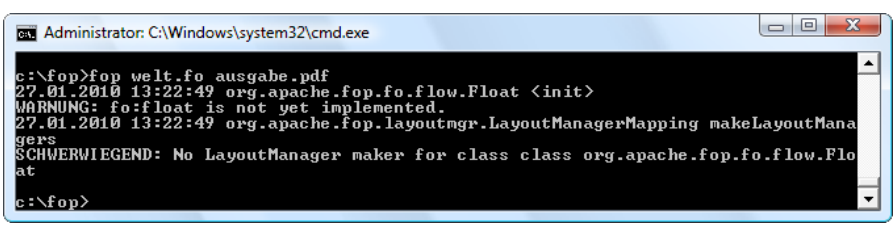

Abbildung 8.41 FOP unterstützt das Float-Konzept nicht.

FOP gibt die Meldung `fo:float ist not yet implemented` aus. Hier kann man dann in FOP schlichtweg nichts machen.

`fo:float` wird innerhalb von `fo:block` notiert.

Listing 8.65 Die Grundstruktur wird angelegt.

```
<fo:block intrusion-displace="line">
   Inhalt
```

```
      <fo:float float="left">
         Inhalt
      </fo:float>
</fo:block>
```

Um die Fließrichtung zu bestimmen, wird das `float`-Attribut verwendet. Dieses Attribut kennt die folgenden Werte:

- `start` – Das Fließobjekt wird an den Start-Rand gesetzt. (Zumindest wenn die Schreibrichtung von links nach rechts läuft.)
- `end` – Hierdurch wird das Fließobjekt an den End-Rand gesetzt. (Auch das gilt nur, wenn die Schreibrichtung von rechts nach links läuft.)
- `left` – Das Fließobjekt wird an den linken Rand gesetzt.
- `right` – Hierdurch wird das Fließobjekt an den rechten Rand gesetzt.

Neben dem `float`-Attribut im `fo:float`-Element muss auch dem `fo:block`-Element das Attribut `intrusion-displace` zugewiesen werden. Durch dieses Attribut reserviert man innerhalb des Blocks den für das `fo:float`-Element benötigten Platz,

- `block` – Hierdurch wird ein Block unter Berücksichtigung von `text-indent` erzeugt.
- `line` – Umfließt das Element unter Berücksichtigung von `start-indent`.
- `indent` – Umfließt das Element unter Berücksichtigung von `text-indent` und `start-indent`.

Zunächst ein Beispiel für eine links fließende Variante:

Listing 8.66 Die Inhalte fließen links.

```
<fo:block intrusion-displace="block">
   <fo:float float="right">
      <fo:block margin-left="3mm">
         Marginalspalte rechts
      </fo:block>
   </fo:float>
Das ist der Inhalt
</fo:block>
```

Und jetzt noch einmal für eine rechts fließende Variante:

Listing 8.67 Der Inhalt fließt rechts.

```
<fo:block intrusion-displace="line">
   <fo:float float="right">
      <fo:block>
         <fo:external-graphic src="image.jpg"
         content-width="30mm"/>
      </fo:block>
   </fo:float>
Hallo, Welt!
</fo:block>
```

Informieren Sie sich im Vorfeld darüber, ob Ihr Formatierer `fo:float` unterstützt. Denn FOP generiert das Dokument dann trotzdem, auch wenn dieser Formatierer `fo:float` eigentlich nicht unterstützt. Das Ergebnisdokument entspricht dann allerdings mit ziemlicher Sicherheit nicht dem Gewünschten.

Register

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X

Z

VORHANG AUF FÜR XML //

- Speziell aus der Sicht von Webentwicklern geschrieben
- Praktische Beispiele ebnen den Einstieg
- XML Schema, Linking-Sprachen und Transformation von XML-Dokumenten
- Ergänzungen und Beispiele unter www.downloads.hanser.de und www.medienwerke.de

XML FÜR WEBENTWICKLER // Moderne Web-Anwendungen sind ohne XML nicht mehr denkbar. Ob Portale, Content Management Systeme oder die Anbindung von Webapplikationen an Datenbanken - an XML führt im Web kein Weg vorbei. Was aber verbirgt sich hinter XML, und wie kann der Webentwickler die verschiedenen XML-basierten Sprachen effizient einsetzen? Hier setzt dieses Buch an.

Der Webentwickler lernt hier XML von Grund auf kennen. Er erfährt alles Wissenswerte zu Dokumenttypen, Elementen und Namensräumen. Nach einem Einstieg in die XML-Syntax geht es mit XML Schema, der Sprache für Dokumenttypdefinitionen, weiter. XPath, XPointer und XLink als die wichtigsten Linking-Sprachen fehlen ebenso wenig wie zwei weitere wichtige Eckpfeiler des Buches, nämlich XSLT und XSL-FO.

Das Buch zeichnet sich durch eine Fülle praxisnaher Tipps und Informationen aus, die beispielweise zeigen, wie sich mittels Transformation aus einfachen XML-Strukturen PDF-Dateien generieren, Grafiken einbinden oder XML-Dateien flexibel formatieren lassen. Aber auch Themen wie XML Schema, DTDs und die zahlreichen XML-basierten Linking-Sprachen spielen in diesem Buch eine wichtige Rolle.

AUS DEM INHALT // Geschichte und Zukunft von XML // XML-Grundlagen // Dokumenttypen // XML Schema // XPath, XPointer und XLink // Ausgabe mit CSS // Transformation mit XSLT // Formatierungen mit XSL-FO //

daniel **KOCH** ist Autor zahlreicher Fachbücher zu Webtechnologien und freiberuflicher Webentwickler. XML gehört für ihn in seiner Arbeit seit Jahren zu den unverzichtbaren Werkzeugen, um leistungsfähige und moderne Webanwendungen zu erstellen.

HANSER

www.hanser.de/computer

ISBN 978-3-446-42256-8

9 783446 422568

Webentwickler und Webprogrammierer