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.

Bei der Herstellung des Werkes haben wir uns zukunftsbewusst für umweltverträgliche und wiederverwertbare Materialien entschieden.

Der Inhalt ist auf elementar chlorfreiem Papier gedruckt.

ISBN 978-3-95845-755-3

1. Auflage 2018

www.mitp.de

E-Mail: mitp-verlag@sigloch.de

Telefon: +49 7953 / 7189 - 079

Telefax: +49 7953 / 7189 - 082

Lektorat: Sabine Janatschek

Sprachkorrektorat: Petra Heubach-Erdmann

Covergestaltung: Christian Kalkert, www.kalkert.de

Satz: Wilfried Grupe

Druck: Plump Medienhaus GmbH, Rheinbreitbach

Bildnachweis Cover: istock.com/lena_serditova

Inhalt

Kapitel 1

Einleitung

XML hat Konjunktur. XML wird in zahlreichen Unternehmen, Behörden, Verwaltungen, Verlagen, Gerichten, Universitäten, Fachhochschulen, im weltweiten Web täglich millionenfach eingesetzt.

Wenn Sie eine Banküberweisung tätigen, Ihre Steuerklärung machen oder Bescheide erhalten, wenn Sie einen Zeitungsartikel oder ein Buch wie dieses lesen, so hat XML mit einiger Wahrscheinlichkeit einen Anteil an dessen technischem Zustandekommen. Es mag sein, dass Sie davon nichts sehen, denn dieser Anteil wird wieder ausgeblendet. Aber XML war vermutlich beteiligt, als unentbehrlicher Helfer.

Mit XML geht einfach alles. Die Verwendbarkeit ist scheinbar unbegrenzt. XML ist äußerst flexibel, bietet alle Voraussetzungen zu höchster Präzision in jeder Branche, über die verschiedensten IT-Systeme hinweg, ist unabhängig von internationalen Zeichensätzen und daher wichtig für den globalisierten Datenaustausch.

XML-Technologien gehören zu den grundlegenden Qualifikationen in der IT. Die fachlichen Kenntnisse werden immer häufiger als selbstverständlich vorausgesetzt. Daher wendet sich das Buch an alle Leser, die sich mit XML-Technologien befassen.

Das Hauptanliegen der Arbeit ist ein Überblick über die aktuellen XML-Technologien XML, XML-Schema, XPath (1.0, 2.0, 3.0, 3.1), XSLT (1.0, 1.1, 2.0, 3.0), XSL-FO, XQuery, XProc, Schematron und XSpec sowie die Unterstützung, die XML-Technologien in Java, C#.NET oder Datenbanken finden.

Zu Beginn eines jeden größeren Kapitels finden Sie einen Abschnitt "Grundlagen". Auf den jeweils folgenden Seiten gibt es vertiefende Details dazu, die vor allem bei den Abschnitten über XML-Schema, XPath und XSLT den Charakter einer Kurzreferenz haben.

Sie können das Buch so lesen, dass Sie zunächst die "Grundlagen" nacheinander durcharbeiten. Die Themen sind so aufgebaut, dass Sie sie aufgrund der bisherigen Lektüre verstehen können. Dann sind Sie je nach Lesetempo in wenig mehr als einer Stunde durch und haben einen generellen Überblick.

- Was[1)] ist ein XML-Dokument?
- Wie[2)] können Sie eine einheitliche Datenstruktur absichern?

1) Seite: 13
2) Seite: 70

- Wie[3)] können Sie XML-Dokumente via XPath auswerten?
- Wie[4)] können Sie XML-Dokumente via XSL in andere Datenstrukturen transformieren?
- Inwiefern stellt XQuery[5)] eine gute Alternative zu XSL dar?
- Wie[6)] können Sie zahlreiche XML-Dokumente in XML-Datenbanken sichern und via XQuery auswerten?
- Wie[7)] können Sie mit Schematron detailgenau testen, ob die erwarteten Ergebnisse stimmen?
- Inwiefern[8)] ist XML ein ideales Datenaustauschformat?

Im Anschluß an die "Grundlagen" finden Sie reichlich Gelegenheit zum Stöbern in den fachlichen Details, die konkrete Hilfe für häufige Programmierprobleme bieten. Da jene Details fachlich ineinandergreifen, ist es sinnvoll, diesen Teil als Nachschlagewerk zu betrachten.

3) Seite: 139
4) Seite: 363
5) Seite: 532
6) Seite: 564
7) Seite: 588
8) Seite: 607

Kapitel 2

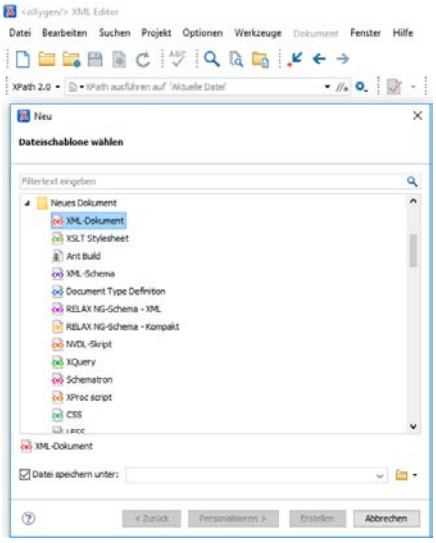
XML

Elemente, Attribute, Kommentare, Entitäten, Prolog, Processing Instruction, Namespaces sind zentrale Grundbegriffe in XML-Technologien.

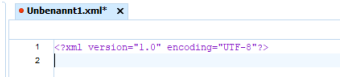
XML-Grundlagen

XML-Dateien sind simple Textdokumente, die einige strukturelle Voraussetzungen erfüllen müssen. Zwar ist es möglich, XML-Dateien mit sehr einfachen Texteditoren zu schreiben. Diese bieten jedoch häufig nur eine geringe fachgerechte Unterstützung.

Weit effizienter ist daher die Arbeit mit professionellen Editoren, die Sie in jeder Hinsicht unterstützen. Einer dieser Editoren ist der Oxygen XML Editor 19.1. Über DATEI|NEU erhalten Sie dieses Fenster:



Mit der Auswahl *XML-Dokument* generiert der Editor eine Datei, in der der sogenannte XML-Prolog bereits enthalten ist.



Diesen Prolog können Sie zunächst löschen, um sich mit den Grundlagen zu befassen.

Elemente

Grundsätzlich hat jedes XML-Dokument genau ein Root-Element. Dieses Element kann so aussehen:

```
<Personen></Personen>
```

Das Element besteht aus einem Start-Tag <Personen> und einem Ende-Tag </Personen>.

Wichtig ist dabei die korrekte Schreibweise. Haben Sie das Start-Tag <Personen> genannt, dann muss das Ende-Tag genau so geschrieben werden, lediglich mit dem "/"-Slash-Zeichen versehen. Das Ende-Tag anders zu schreiben, etwa </personen> oder </PERSONEN>, wäre von vornherein falsch: Das XML-Dokument wäre nicht wohlgeformt.

Für die Benennung der Elemente gibt es einige Einschränkungen, auf die ich später[9)] noch einmal eingehe.

Nun hindert Sie niemand daran, zwischen Start- und Ende-Tag eines Elements weitere Elemente einzufügen. Zum Beispiel so:

```
<Personen>
   <Person></Person>
   <Person></Person>
</Personen>
```

Das Verfahren können Sie beliebig ausbauen, zudem können Sie zwischen den jeweiligen Start- und Ende-Tags auch normalen Text einfügen.

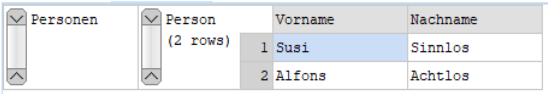
```
<Personen>
  <Person>
    <Vorname>Susi</Vorname>
    <Nachname>Sinnlos</Nachname>
  </Person>
  <Person>
    <Vorname>Alfons</Vorname>
    <Nachname>Achtlos</Nachname>
  </Person>
</Personen>
```

Wenn Sie dieses XML-Dokument in eine Datei "Personen.xml" abspeichern, dann können Sie diese Datei mit einem normalen Internet-Browser öffnen. Der Firefox zeigt die Datei beispielsweise so an:

9) Seite: 23

```
−<Personen>
  −<Person>
      <Vorname>Susi</Vorname>
      <Nachname>Sinnlos</Nachname>
    </Person>
  −<Person>
      <Vorname>Alfons</Vorname>
      <Nachname>Achtlos</Nachname>
    </Person>
  </Personen>
```

Andere Programme können das genauso gut. Gute Power-Tools zeigen darüber hinaus noch eine andere Darstellung.



Personen	Person (2 rows)	Vorname	Nachname
	1	Susi	Sinnlos
	2	Alfons	Achtlos

Angenommen, Sie kennen von einer Person weder den Vornamen noch den Nachnamen. Wenn zwischen dem Start- und Ende-Tag "Person" kein Inhalt abgebildet werden kann, wenn das Element also leer ist, dann handelt es sich sinnigerweise um ein "leeres Element", bei dem Sie sich das Ende-Tag sparen können, sofern der "/" am Ende des Start-Tags erscheint. Beispiel:

```
<Personen>
   <Person/>
</Personen>
```

Attribute

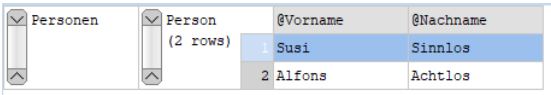
Alternativ zur eben dargestellten Elementschreibweise können Sie Inhalte auch als Attribute einfügen. Ein Attribut wird grundsätzlich in das Start-Tag eines Elements geschrieben. Da die ursprünglichen Kind-Elemente entfallen können, schreiben Sie die Information kurzerhand als leeres Element, aber mit Attributen.

```
<Personen>
  <Person Vorname="Susi"
          Nachname="Sinnlos"/>
  <Person Vorname="Alfons"
          Nachname="Achtlos"/>
</Personen>
```

Speichern Sie das in einer Datei "Personen2.xml" ab, so zeigt der Internetbrowser das so an:

```
−<Personen>
    <Person Vorname="Susi" Nachname="Sinnlos"/>
    <Person Vorname="Alfons" Nachname="Achtlos"/>
  </Personen>
```

Auch das XML-Power-Tool macht mit und zeigt diese Darstellung, die sich nur in einer scheinbar winzigen Kleinigkeit von der vorherigen unterscheidet: das @" in *@Vorname* und *@Nachname*.



Personen	Person (2 rows)	@Vorname	@Nachname
	1	Susi	Sinnlos
	2	Alfons	Achtlos

Die Attributschreibweise hat gegenüber der Elementschreibweise nur einen Nachteil: Jedes Attribut darf nur ein einziges Mal vorkommen. Hat eine Person also mehrere Vornamen, so bietet es sich an, den Nachnamen als Attribut und die Vornamen in Elementschreibweise darzustellen.

```
<Personen>
  <Person Nachname="Holzflos">
    <Vorname>Hugo</Vorname>
    <Vorname>Helmut</Vorname>
    <Vorname>Horst</Vorname>
  </Person>
  <Person Nachname="Nixlos">
    <Vorname>Tanja</Vorname>
    <Vorname>Theodora</Vorname>
  </Person>
</Personen>
```

Kommentare

Ein großer Vorteil von XML-Dokumenten ist, dass Sie auch Kommentare einfügen können. Das erleichtert die Lesbarkeit sehr und ist auch in der praktischen Arbeit von erheblicher Bedeutung, nicht zuletzt bei der Fehlersuche. XML-Kommentare[10)] beginnen mit "<!--" und enden mit "-->".

```
<Personen>
  <!-- Attributschreibweise -->
  <Person Vorname="Resi"
          Nachname="Denzschlos"/>
  <!-- Elementschreibweise -->
  <Person>
    <Vorname>Lotte</Vorname>
    <Nachname>Rielos</Nachname>
  </Person>
</Personen>
```

Entitäten

Sie haben schon bemerkt, dass jedes Tag mit einem "<" beginnt und mit einem ">" endet. Diese beiden Zeichen haben eine zentrale Bedeutung in XML. Das kann Sie jedoch in

10) Seite: 29

einige Verlegenheit bringen, wenn Sie die hochwichtige Information "3 < 4" in XML abbilden möchten.

```
<Info>3 < 4</Info>
```

Dieser Versuch geht schief. Jedes XML-Tool, das etwas auf sich hält und nicht außergewöhnlich leidensfähig ist, meckert Sie an:

```
XML-Verarbeitungsfehler: nicht wohlgeformt
```

oder im schönsten IT-Deutsch:

```
System-Fehlerlevel: error
The content of elements must consist
of well-formed character data or markup.
```

Hier bleibt Ihnen nur übrig, das "<" als Entität zu deklarieren und entsprechend zu kennzeichnen.

```
<Info>3 &lt; 4</Info>
```

Es gibt mehrere Standard-Entitäten[11)] und auch die Möglichkeit, eigene Entitäten zu definieren. Dazu später mehr.

Prolog

Häufig finden Sie am Anfang eines XML-Dokuments einige Zusatzinformationen, die fast durchweg so aussehen:

```
<?xml version="1.0"?>
<Personen>
   <Person/>
</Personen>
```

<?xml version="1.0"?> ist die Minimalinformation. Sie besagt, dass es sich um ein XML-Dokument in der Version 1.0[12)] handelt.

Hin und wieder finden Sie dort auch Informationen zum verwendeten Encoding[13)]. Darin ist ein Hinweis auf den im Dokument verwendeten Zeichensatz enthalten. Fehlt diese Angabe, dann handelt es sich per Default um den Zeichensatz "UTF-8".

11) Seite: 26
12) Seite: 23
13) Seite: 24

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

Processing Instruction

Neben dem Prolog können Sie noch Verarbeitungsanweisungen[14] mitgeben für das Programm, das das XML-Dokument auswertet. Beispiel:

```
<?versandperMail dort@woderpfefferwaechst.de"?>
```

Das verarbeitende Programm wertet diese Information aus und nimmt (hoffentlich) wohlwollend zur Kenntnis, was hier angemerkt wird.

```
<?xml version="1.0"
      encoding="ISO-8859-1"?>
<?sorry ich@habnichtsgemacht.gov"?>
<Personen>
   <Person/>
</Personen>
```

Namespace

Namespaces[15] sind ebenfalls ein zentrales Thema in XML. Sie eröffnen weitreichende Möglichkeiten, ein XML-Dokument einem Namensraum zuzuordnen und dabei auch versionsbedingte Informationen einzubinden.

```
<Personen xmlns="www.Kundenliste.de/2018">
  <k:Person xmlns:k="www.besondererKunde.de">
    <k:Vorname>Wanja</k:Vorname>
    <k:Nachname>Wunschlos</k:Nachname>
  </k:Person>
</Personen>
```

- Zu unterscheiden sind hier Namespace-Präfixe, die ein Kürzel für einen Namensraum definieren, sowie Default-Namespaces.
- Im obigen Beispiel ist *xmlns="www.Kundenliste.de/2018"* der Default-Namespace.
- *xmlns:k="www.besondererKunde.de"* definiert einen Namensraum mit einem Kürzel *k*, dem sogenannen Namespace-Präfix, das bei den Elementen *k:Person*, *k:Vorname* und *k:Nachname* zum Einsatz kommt.

Stören Sie sich nicht daran, wenn einzelne Internet-Browser sich weigern, die Namespaces anzuzeigen, nachdem Sie das Dokument in eine XML-Datei gespeichert

14) Seite: 36
15) Seite: 48

haben und den Browser auffordern, diese anzuzeigen. Die Tools arbeiten da unterschiedlich, und mit einem guten XML-Editor finden Sie ohnehin alles wieder.

XML-Datenstrukturen

Bereits die wenigen Anmerkungen verdeutlichen die enorme Flexibilität, die XML zu bieten hat. Es ist praktisch alles darstellbar, was in das menschliche Hirn hineinpasst, egal wie komplex ein Sachverhalt auch sein mag. Vorausgesetzt, Sie halten sich an einige Grundregeln der Wohlgeformtheit.

Das begrenzt sich durchaus nicht auf die klar strukturierten Daten, die ich bisher beschrieben habe. Auch Folgendes[16] ist nicht nur möglich, sondern kommt recht häufig vor:

```
<?xml version="1.0"
      encoding="ISO-8859-1"?>
<Meldung>Mit XML durch das Weltall,
zum <fett>Mars</fett>,
<kursiv>Jupiter</kursiv>
und in die Milchstrasse.</Meldung>
```

Die Kernfrage bei dieser überwältigenden Flexibilität lautet: Wie schaffen Sie es, Programme zu schreiben, die damit klarkommen? Darum geht es in diesem Buch.

XML-Basics

XML ist eine erweiterbare, flexible, stukturierte Markup-Sprache, die in unterschiedlichen Bereichen zum Einsatz kommt, etwa bei Transformation zu HTML, XML, Text, SVG, RTF, PNG, TIFF, PDF.

```
<?xml version="1.0" encoding="UTF-8"?>
<?lernen was="XML XSLT XSD XQuery"?>
<r:root xmlns:r="Namespaces">
    <child attribut="JA"/>
    <![CDATA[ <Spezial/> ]]>
</r:root>
```

Was ist das für eine Sprache,

- die so einfach strukturiert ist, dass man ihre Grundlagen leicht in einer halben Stunde lernen kann?

16) Seite: 40

- die nicht auf einem eigenen Compiler, Interpreter, anderen Übersetzer basiert, der sprachspezifische Kommandos in Maschinensprache umwandelt?
- die (abgesehen von einer äußerst knappen Formalstruktur) über fast keine Schlüsselworte oder Vokabular verfügt, aus denen eine Sprache normalerweise mindestens besteht?
- die von sich aus praktisch gar nichts mitbringt, aber dennoch eine äußerst präzise, versionsbezogene Datenstrukturdefinition für die unterschiedlichsten Branchen ermöglicht?
- die zudem alle Möglichkeiten für eine leichthändige Datenkonvertierung zwischen den unterschiedlichsten Datenformaten und Zeichensätzen bietet?
- die selbst de facto überhaupt nichts "tut" oder "kann", sondern sämtliche Möglichkeiten aus der breiten Unterstützung anderer Technologien bzw. Sprachen bezieht?
- die von sich aus weder Technologien zu Systemkonfiguration, Datenaustausch, GUI, Datenbankzugriffen, Prozessdefinition, Automatisierung, präzisen Definition hochkomplexer Datenstrukturen, Transformation beliebiger Datenstrukturen in beliebige Zielformate, leistungsfähiger Publishing-Standards und vielem Anderen mehr bereitstellt, aber in all diesen Bereichen hervorragend zum Einsatz kommt?

Auswertung, Transformation	XPath, XSL, XQuery
Webseiten	HTML
Verknüpfung	XInclude, XLink, XPointer
Validierung, Testen	DTD, XML-Schema (XSD), Relax NG, Schematron, XSpec, NVDL
Signatur, Verschlüsselung	XML Signature, XML Encryption
Arbeit in verteilten Systemen	XML-RPC
EDI	XML, daneben Übertragungen von EDIFACT, ANSI X.12, SAP IDoc, CSV, CargoImp u.a.m.
Finanzberichte	XBRL
Formatierung	FO, XSL-FO, MathML, SVG, EPUB, WordML
Publishing-Standards	DITA, DocBook, TEI
Grafiken	SVG, X3D
Formatting Object	TIFF, PNG, PDF, PS, PCL
Geodaten	CityGML (City Geography Markup Language), GML (Geography Markup Language), GPX (GPS Exchange Format),

	KML (Google Earth: Keyhole Markup Language)
Landwirtschaft	AgroXML
Prozessdefinition	ANT, XProc
Webservices	SOAP, WSDL, REST
Office-Anwendungen	OASIS Open Document Format for Office Applications, RTF

XML: erweiterbar, flexibel

Zunächst ist XML eXtensible, also eine erweiterbare Markup Language. Sie besteht ausdrücklich nicht aus einer endlichen Menge von einigen Dutzend Schlüsselwörtern, die man (in diversen Programmiersprachen) kennen muss, um Code schreiben zu können, den der PC dann ausführt. Sondern XML ist erweiterbar. Potenziell können unendlich viele Begriffe definiert werden, in den unterschiedlichsten Sprachen, Zeichensätzen und Schreibweisen. Die einzige Bedingung ist, einige wenige grundlegende Anforderungen einzuhalten.

Zweitens handelt es sich um eine äußerst flexible Markierungssprache, die wohlgeformte, strukturierte Daten definiert und obendrein eine präzise Zuordnung zu einem fachlichen Kontext, auch mit Versionsunterschieden erlaubt. XML zieht seine Effizienz jedoch aus einer breiten Unterstützung durch andere Technologien.

Dabei kann XML sehr schwach strukturiert (dokumentzentriert: Die XML-Elemente dienen zur semantischen Strukturierung des Textes, was eine maschinelle Verarbeitung erschwert), sehr stark strukturiert (datenzentriert: XML-Elemente, Attribute etc. folgen einer klaren Strukturdefinition zur effizienten maschinellen Verarbeitung), aber auch *mixed content* haben (semistrukturiert: XML-Dokumente als Mischung aus starker und schwacher Strukturierung; die Ansätze zur effizienten Auswertung sind hier andere als bei Datenzentrierung).

Die vorliegende Arbeit hat ihren Schwerpunkt auf datenzentrierten Dokumenten. XML ist hier nie Selbst- oder Endzweck, sondern Teil einer Verarbeitungskette: Immer ist ein Folgeprogramm nötig, das mit der jeweiligen XML-Datenstruktur umzugehen weiß. Das mag ein Webbrowser sein, der skalierbare Vektorgrafiken (SVG, eine Spezialform von XML) anzeigt. Ebenso kann es ein Systemprogramm sein, das eine in XML definierte System- oder Serverkonfiguration auswertet. Auch kann es sich um ein in Java[17], C#.NET[18], VisualBasic.NET[19], C++ oder in einer anderen Sprache geschriebenes Programm handeln, das in XML definierte Prozesse schrittweise abarbeitet.

17) Seite: 627
18) Seite: 616
19) Seite: 623

Die Struktur der in XML vorliegenden Daten und die Programme, die sie auswerten, müssen also Hand in Hand gehen. XML-Datenstrukturen, die ein auswertendes Programm nicht verarbeiten kann, sind wirkungslos.

Die Programme folgen unterschiedlichen Verarbeitungsmodellen. SAX verarbeitet XML als sequenziellen Datenstrom und hält für bestimmte Ereignisse spezielle *callback functions* bereit. Das sehr speicherintensive DOM betrachtet XML dagegen auf der Baumstruktur und gewährt wahlfreien Zugriff mit Manipulationsmöglichkeiten. Daneben stehen noch die Pull-API (sequenzielle Verarbeitung mit Iterator) oder die Verarbeitung auf Byte-Ebene bereit. Häufig werden Objekte in XML-Dokumente umgewandelt (Serialisierung) oder umgekehrt (marshalling); siehe JAXB oder XML-Schema-Definition-Toolkit in .NET.

Trotzdem ist es nicht notwendig, die interne Logik jener Programme, die XML-Dokumente auswerten, zu kennen, um XML-Dokumente schreiben zu können, die das Programm auswerten kann. Es reicht aus, klare Vorgaben hinsichtlich der Struktur und Detailtypen der zu übermittelnden Daten verfügbar zu haben. Hier helfen diverse Standards weiter, die eine Strukturdefinition der XML-Dokumente erlauben, unter anderem DTD und XML-Schema (XSD).

XML-Schema erlaubt, die grundsätzlich extrem gestaltungsflexiblen Möglichkeiten von XML-Dokumenten auf eine endliche Anzahl zulässiger Element- und Attributnamen einzugrenzen, verbunden mit einer begrifflichen Zuordnung zu bestimmten Namespaces. Auf diese Weise wird eine Datenstruktur definierbar, die von Folgeprogrammen ausgewertet werden kann. Es besteht die Möglichkeit einer Vorprüfung (Validierung), ob das jeweilige XML-Dokument diesen Vorgaben entspricht; falls nicht, kann die Weiterverarbeitung gestoppt werden.

Freilich kommt es recht häufig vor, dass die verfügbaren Daten in einer Struktur vorliegen, die die Folgeprogramme nicht auswerten können. Dann wird eine Konvertierung erforderlich. Hier hilft XSLT, ab XSLT 2.0 auch dann, wenn die ursprünglichen Daten gar nicht in XML vorliegen, sondern beispielsweise in Textformaten wie CSV.

XSL ist selbst auch ein XML-Dokument, jedoch mit einem exakt definierten Aufbau sowie einer Reihe klar definierter Schlüsselbegriffe. XSL bietet (in Kombination mit XPath) sehr effiziente Ansätze zur Transformation vorliegender Daten in andere Strukturen.

XSL und XPath bieten effiziente Möglichkeiten zur Transformation von (strukturierten) XML-Dokumenten in diverse Zielformate, z.B. HTML, XML, Text, SVG, RTF, PNG, TIFF, PDF und andere mehr. Dabei können mehrere XML-Quelldokumente ebenso berücksichtigt werden wie mehrere Ausgabedokumente.

Grundvoraussetzung für effizientes Programmieren mit XSL ist, dass sowohl die Struktur des XML-Quelldokuments als auch die Struktur des Zielformats zweifelsfrei klar ist. Während das XML-Inputdokument die (hoffentlich) klar strukturierten Daten liefert, stehen in XSL/XPath jene Programmieranweisungen, die die Struktur des Quelldokuments in die gewünschte Zielstruktur konvertiert. Das Kind der Ehe von XML und XSL ist das gewünschte Dokument.

XML: Wohlgeformte Dokumente

XML-Dokumente sind wohlgeformt, sofern sie nicht gegen eine XML-Regel verstoßen, z.B.:

- Jedes XML-Dokument hat genau ein Root-Element.
- Elemente dürfen nicht mehrere Attibute mit demselben Namen haben.
- Alle Elemente mit Child-Elementen (auch Textknoten) haben ein Start- und ein Ende-Tag, z.B. *<content>Inhalt</content>*, die innerhalb ihres Parent-Knotens abgeschlossen sein müssen.
- Elemente ohne Childnodes sind leer, sie benötigen kein Ende-Tag, sondern können einfach geschlossen werden mit "/>" (z.B. *<content/>*. Eventuell weisen sie Attribute auf, wie im folgenden Beispiel:

```
<content mycontent="XML is my favourite"/>
```

Für die Benennung von Elementen und Attributen gibt es einige Einschränkungen. So darf das erste Zeichen keine Zahl sein. Ebenso ist eine ganze Reihe von Sonderzeichen ganz oder eingeschränkt verboten, die als logische oder Rechenoperatoren oder Bestandteile von XPath-Statements zum Einsatz kommen können:

```
+ - * / \ & " ' % # ! = ; () [] {} @ $ § ? .
```

: kommt im Zusammenhang mit Namespaces, Blanks kommen bei Attributdeklaration zum Einsatz.

<fe-ld/> oder *<fe.ld/>* ist erlaubt, als erstes Zeichen *<-feld/>* oder *<.feld/>* aber nicht. Problemlos: *<_1feld/>*.

XML-Version

Im Normalfall gehört zu jedem XML-Dokument ein Prolog, der mindestens über die XML-Version informiert.

In den allermeisten Fällen hat die Version den Wert "1.0".

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

Nur in sehr seltenen Ausnahmefällen (die Daten enthalten Steuerzeichen wie den vertikalen Tabulator, Zeilenvorschub oder Inhalte in selten verwendeten Sprachen), und wenn die verwendeten Parser damit umgehen können, kann die Versionsbenennung von 1.1 vorteilhaft sein.

XML Encoding

Speziell im internationalen Datenaustausch werden unterschiedliche Codierungen verwendet. Sofern die in XML verwendeten Zeichen nicht aus dem UTF-8-Encoding stammen, ist im XML-Prolog das verwendete Encoding anzugeben.

Wer konsequent mit UTF-8 arbeitet, erspart sich den Umgang mit Sonderzeichen. Das ist aber nicht immer möglich: Auch in ISO-8859-1 gibt es eine längere Reihe von Sonderzeichen, die in XML nicht durch HTML-Entitätsreferenzen abgedeckt sind, etwa für das EURO-Zeichen.

Ein Aufruf der HTML-Referenz *€* führt in XML zu einem Fehler, hier muss mit *€* bzw. deren Hexwert *€* gearbeitet werden. In

- *http://www.w3schools.com/charsets/ref_html_8859.asp*
- *https://wiki.selfhtml.org/wiki/Referenz:HTML/Zeichenreferenz*
- *http://docstore.mik.ua/orelly/xml/xmlnut/ch26_01.htm*

finden Sie sehr brauchbare Übersichten.

Das Encoding definiert die Zeichencodierung, die im Dokument verwendet werden soll. UTF-8 ist dabei die Standard-Codierung. Bitte beachten Sie, dass nicht alle Parser auch sämtliche Encodings unterstützen. Einige Tools ignorieren das angegebene Encoding und arbeiten grundsätzlich mit UTF-8.

UTF-8	Standard-Encoding in XML-Dokumenten. UTF-8 ist so designt, dass alle ASCII Dokumente legale UTF-8-Dokumente darstellen, was bei UTF-16 und Latin1 nicht der Fall ist.
UTF-16	Ein Zwei-Byte-Encoding von Unicode, das alle Zeichen von Unicode 3.0 und früher umfasst.
ISO-10646-UCS-2	Die Basis-Multilingual-Version von Unicode; der Zeichensatz ist weitgehend identisch mit UTF-16. Der Unterschied betrifft lediglich Unicode 3.1 und höher.
ISO-10646-UCS-4	Ein Vier-Byte-Encoding von Unicode.
ISO-8859-1	Latin-1, ASCII plus jene Zeichen, die für die meisten westeuropäischen Sprachen verwendet werden, inkl. Dänisch, Deutsch, Holländisch, Englisch, Finnisch, Flämisch, Irisch, Isländisch, Italienisch, Norwegisch, Portugiesisch, Spanisch und Schwedisch.

ISO-8859-2	Latin-2, ASCII plus jene Zeichen, die für die meisten zentraleuropäischen Sprachen verwendet werden, inkl. Kroatisch, Tschechisch, Ungarisch, Polnisch, Slowakisch, Slowenisch.
ISO-8859-3	Latin-3, ASCII plus jene Zeichen für Esperanto, Galizisch, Maltesisch, Türkisch.
ISO-8859-4	Latin-4, ASCII plus jene Zeichen für Lappländisch, Lettisch, Litauisch, Grönländisch. Wurde weitgehend ersetzt durch ISO-8859-10, Latin-6.
ISO-8859-5	ASCII plus die kyrillischen Zeichen für Belorussisch, Bulgarisch, Mazedonisch, Russisch, Serbisch, Ukrainisch.
ISO-8859-6	ASCII plus Arabisch.
ISO-8859-7	ASCII plus Griechisch.
ISO-8859-8	ASCII plus Hebräisch.
ISO-8859-9	Latin-5, weitgehend identisch mit Latin-1 (ASCII plus westeuropäisch), aber ohne bestimmte türkische und isländische Zeichen.
ISO-8859-10	Latin-6: Zeichen für nordeuropäische Sprachen wie Grönländisch, Isländisch, Lappländisch, Litauisch. Ähnlich wie Latin-4, ergänzt in ISO-8859-13.
ISO-8859-11	ASCII plus Thai. Die Unterstützung von XML-Prozessoren ist nicht optimal.
ISO-8859-12	Nicht benötigt.
ISO-8859-13	Alternativer Zeichensatz für baltische Sprachen. Vgl. Latin 6.
ISO-8859-14	Latin-8; eine Variante für Latin-1 mit Zusatzzeichen für Gälisch und Welsch.
ISO-8859-15	Latin-9; Revision von Latin-1. Weitestgehend identisch mit ISO-8859-1.
ISO-8859-16	Latin-10; für Rumänisch.

ISO-2022-JP	Sieben-Bit-Encoding mit japanischem Zeichensatz JIS X-0208-1997, in E-Mails und im Web verwendet, siehe RFC 1468.
Shift_JIS	Japanischer Zeichensatz JIS X-0208-1997, in Microsoft Windows verwendet.
EUC-JP	Japanischer Zeichensatz JIS X-0208-1997, in den meisten UNIX-Varianten verwendet.

XML-Entitäten

XML-Entitäten finden Verwendung in Form von Standard-, selbst definierten Entitäten, Einbindung separater XML-Dokumente sowie zur Entitätsdeklaration in XSLT.

Folgende Standard-Entitäten sollten geläufig sein:

```
&lt; für <
&gt; für >
&quot; für "
&amp; für &
```

Selbst definierte Entitäten

Ergänzend können selbst definierte Entitäten zum Einsatz kommen:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE mytext [
<!ENTITY LvH "Leute von heute" >
]>
<mytext>
Hallo, &LvH;
</mytext>
```

Das Ergebnis sieht im Browser dann so aus:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE mytext>
<mytext> Hallo, Leute von heute </mytext>
```

Entitäten als externe Dokumente

Darüber hinaus besteht die Möglichkeit, mittels selbst definierter Entitäten externe XML-Dokumente in ein größeres Dokument einzubinden. Auf diese Weise können mehrere

Autoren parallel an unterschiedlichen Dokumenten arbeiten; nach abgeschlossener Arbeit werden die XML-Dokumente in ein umfassenderes Dokument eingefügt.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE article [
<!ENTITY KAP1 SYSTEM "Kapitel1.xml">
]>
<article>
  <chapter>
    <title>XML-Basics</title>
    &KAP1;
  </chapter>
</article>
```

Wobei das externe Dokument etwa so aussehen könnte:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<section>
  <title>Attribute</title>
  <para>Attribute sind ebenso sinnvoll wie Elemente.</para>
</section>
```

Im Arbeitsspeicher werden die Dokumente dann zusammengefügt zu:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<article>
  <chapter>
    <title>XML-Basics</title>
    <section>
      <title>Attribute</title>
      <para>Attribute sind ebenso sinnvoll wie Elemente.</para>
    </section>
  </chapter>
</article>
```

Entitätsdeklaration in XSLT

Je nach XSLT-Prozessor[20)] können im Vorspann zu XSLT auch Entitäten deklariert werden (bei diversen Prozessorvarianten habe ich hier temporäre Probleme gefunden, die teilweise bereits beseitigt sind).

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE stylesheet [
<!ENTITY euro  "&#8364;" >
<!ENTITY auml  "&#228;" >
<!ENTITY Auml  "&#196;" >
<!ENTITY ouml  "&#246;" >
<!ENTITY Ouml  "&#214;" >
<!ENTITY uuml  "&#252;" >
<!ENTITY Uuml  "&#220;" >
<!ENTITY szlig  "&#223;" >
```

20) Seite: 369

```
]>
<xsl:stylesheet version="2.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
 <xsl:output method="html" />
    <xsl:decimal-format
        name="df"
        decimal-separator=","
        grouping-separator="."
        minus-sign="-"
        digit="#"/>
 <xsl:template match="/">
  <html>
   <head>
     <meta
      http-equiv="Content-Type"
      content="text/html; charset=UTF-8"/>
   </head>
   <body>
    <xsl:value-of
         select="format-number(
                 sum(//Gehalt),
                 '#.##0,00 &euro;',
                 'df')" />
    <br />
    <table>
     <tr>
      <td>ae</td>
      <td>&auml;</td>
      <td>
       <xsl:text>&auml;</xsl:text>
      </td>
     </tr>
     <tr>
      <td>Ae</td>
      <td>&Auml;</td>
      <td>
       <xsl:text>&auml;</xsl:text>
      </td>
     </tr>
     <tr>
      <td>oe</td>
      <td>&ouml;</td>
      <td>
       <xsl:text>&ouml;</xsl:text>
      </td>
     </tr>
     <tr>
      <td>Oe</td>
      <td>&Ouml;</td>
      <td>
       <xsl:text>&Ouml;</xsl:text>
      </td>
     </tr>
     <tr>
      <td>ue</td>
      <td>&uuml;</td>
      <td>
       <xsl:text>&uuml;</xsl:text>
      </td>
     </tr>
     <tr>
      <td>Ue</td>
      <td>&Uuml;</td>
      <td>
```

```
        <xsl:text>&Uuml;</xsl:text>
       </td>
      </tr>
      <tr>
       <td>szlig</td>
       <td>&szlig;</td>
       <td>
        <xsl:text>&szlig;</xsl:text>
       </td>
      </tr>
     </table>
    </body>
   </html>
  </xsl:template>
</xsl:stylesheet>
```

Das Ergebnis sieht im Browser dann so aus:

23.816,77 €

ae	ä	ä
Ae	Ä	ä
oe	ö	ö
Oe	Ö	Ö
ue	ü	ü
Ue	Ü	Ü
szlig	ß	ß

XML-Kommentare

In XML, XSLT, XML-Schema und anderen Standards sind ergänzende Kommentare hilfreich, um die Wartung komplexer Anwendungen zu erleichtern. Neben Standard-Kommentaren stehen auch *CDATA*-Kommentare bereit.

Ergänzende Kommentare können weitere Hilfestellung geben. Ein Standard-XML-Kommentar beginnt mit <!-- und endet mit -->.

```
<root>
   <!-- this is a comment, you are free to write down your CV -->
</root>
```

XML-Kommentare in XSL erzeugen

In XSL ist es angebracht, zum besseren Verständnis der Programmlogik lokale Kommentare einzubauen, die nicht im Ergebnisdokument erscheinen. Sinnvolle

Kommentare im Quelltext können die Wartung der Programme sehr erleichtern, daher sind sie unbedingt zu empfehlen.

```
<erg>
  <!-- dieser Kommentar gilt nur lokal in XSL,
       wird nicht im Ergebnisdokument erscheinen -->
</erg>
```

Die Ausgabe im Ergebnisdokument lautet wie beabsichtigt ohne Kommentar:

```
<erg/>
```

Um im Ergebnisdokument einen Kommentar sichtbar zu machen, können Sie mit *<xsl:comment>* arbeiten.

```
<erg>
  <xsl:comment>Dieser Kommentar wird im Ergebnis erscheinen</xsl:comment>
</erg>
```

Die Ausgabe im Ergebnisdokument lautet:

```
<erg>
  <!--Dieser Kommentar wird im Ergebnis erscheinen-->
</erg>
```

XML-Kommentare in XSL auswerten

Umgekehrt ist es auch möglich, mit XSLT die Kommentare in den Input-Dokumenten auszuwerten. Betrachten Sie folgendes XML-Input-Dokument, das (abgesehen von einem "root"-Node) ausschließlich XML-Kommentare aufweist.

```
<root>
<!--Kommentar 1-->
<!--Kommentar 2-->
<!--Kommentar 3-->
<!--Kommentar 4-->
</root>
```

Nun soll versucht werden, diese Kommentare in XSLT auszuwerten. Das funktioniert recht einfach:

```
<xsl:stylesheet version="1.0"
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="xml"
              version="1.0"
              encoding="UTF-8"
              indent="yes"/>
  <xsl:template match="/">
```

```
    <Kommentare>
      <xsl:for-each select="/root/comment()">
        <info>
          <xsl:value-of select="."/>
        </info>
      </xsl:for-each>
    </Kommentare>
  </xsl:template>
</xsl:stylesheet>
```

Das Ergebnis ist wenig überraschend:

```
<?xml version="1.0" encoding="UTF-8"?>
<Kommentare>
 <info>Kommentar 1</info>
 <info>Kommentar 2</info>
 <info>Kommentar 3</info>
 <info>Kommentar 4</info>
</Kommentare>
```

Das obige XSL-Stylesheet konzentriert sich jedoch nur auf jene Kommentare, die unmittelbar unterhalb des "root"-Elements stehen. In der Regel sieht die Datenstruktur jedoch etwas komplexer aus:

```
<root>
 <!--Kommentar 1-->
 <Ebenea>
  <ka>
   <!--Kommentar 2-->
  </ka>
  <Ebeneb>
   <kb>
    <!--Kommentar 3-->
   </kb>
   <Ebenec>
    <kc>
     <!--Kommentar 4-->
    </kc>
   </Ebenec>
  </Ebeneb>
 </Ebenea>
</root>
```

Möchten Sie nun nicht nur sämtliche Kommentare auf unterschiedlichen Ebenen auslesen, sondern auch den XPath zu dieser Ebene wissen, so können Sie sich hiermit behelfen:

```
<xsl:stylesheet   version="1.0"   xmlns:xsl="http://www.w3.org/1999/XSL/
Transform">
 <xsl:output method="xml" version="1.0" encoding="UTF-8" indent="yes"/>
 <xsl:template match="/">
  <Kommentare>
   <xsl:for-each select="/*/descendant-or-self::*/comment()">
    <info>
     <xsl:attribute name="xpath">
      <xsl:for-each select="ancestor-or-self::*">
       <xsl:text>/</xsl:text>
```

```
       <xsl:value-of select="name()"/>
      </xsl:for-each>
     </xsl:attribute>
     <xsl:value-of select="."/>
    </info>
   </xsl:for-each>
  </Kommentare>
 </xsl:template>
</xsl:stylesheet>
```

Das Ergebnis hilft Ihnen weiter:

```
<Kommentare>
 <info
    xpath="/root">Kommentar 1</info>
 <info
    xpath="/root/Ebenea/ka">Kommentar 2</info>
 <info
    xpath="/root/Ebenea/Ebeneb/kb">Kommentar 3</info>
 <info
    xpath="/root/Ebenea/Ebeneb/Ebenec/kc">Kommentar 4</info>
</Kommentare>
```

CDATA in Scripting-Dateien

Neben Standard-Kommentaren gibt es noch *CDATA*-Kommentare. *CDATA*-Kommentare beginnen mit *<![CDATA[* und enden mit *]]>*.

Im Unterschied zu den Parsed Character Data (*PCDATA*), die durch einen Parser verarbeitet werden, beinhalten *CDATA*-Kommentare Daten, die nicht geparst werden. *CDATA*-Kommentare erlauben beispielsweise, zusätzliche XML-Elemente einzufügen, die im XML-Schema nicht definiert sind. Durch *CDATA* wird es möglich, diese nicht-validen Inhhalte vor etwaiger Validierung gegen DTD oder XML-Schema zu schützen.

Einen Einsatzbereich für *CDATA*-Kommentare finden Sie beispielsweise in Windows-Scripting-Dateien, die im Kern XML-Dokumente darstellen, deren Scriptanteile jedoch über den Windows Script Host ausgeführt werden können.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<job id="T1">
    <comment>
        <c1>File: dosomething.wsf</c1>
        <c1>Author: Wilfried Grupe</c1>
        <c1>Datum: 01.01.2018</c1>
        <c1>Description: writes Hello</c1>
    </comment>
    <script language="VBScript">
<![CDATA[
sub dosomething
    wscript.echo "Hallo"
end sub
]]>
    </script>
    <script language="JScript">
<![CDATA[
try {
```

```
    // calling vbscript from JScript:
    dosomething();
}
catch(e) {
    // Exceptionhandling in case something is going wrong
    WScript.echo(e.description);
}
]]>
    </script>
</job>
```

Einen vergleichbaren Einsatzzweck findet der *CDATA*-Kommentar in ANT, wo mittels JavaScript mehrere ANT-Echoaufrufe generiert werden, die im Ergebnis das unten stehende Bild ergeben.

```
  <target name="jstest">
    <script language="javascript">
    <![CDATA[
      for(var z=0; z<15;z++){
        var str="";
        for(var s=0; s<48; s++) {
          if(    z===s || s+z===14
            || s===17 || s===31  || s===33
            || (z==14 && s > 33)
            || (z < 8) &&(s===17+z || z+s===31 )){
            str = str + "#";
          }
          else str = str + " ";
        }
        var ve = MYANTPROJECT.createTask("echo");
        ve.setMessage(str);
        ve.perform();
      }
    ]]>
    </script>
  </target>
```

```
jstest:
     [echo] #              #  #              #  #
     [echo]  #            #   ##            ##  #
     [echo]   #          #    # #          # #  #
     [echo]    #        #     #  #        #  #  #
     [echo]     #      #      #   #      #   #  #
     [echo]      #    #       #    #    #    #  #
     [echo]       # #         #     # #     #  #
     [echo]        #          #      #      #  #
     [echo]       # #         #             #  #
     [echo]      #   #        #             #  #
     [echo]     #     #       #             #  #
     [echo]    #       #      #             #  #
     [echo]   #         #     #             #  #
     [echo]  #           #    #             #  #
     [echo] #             #   #             #  ################
BUILD SUCCESSFUL
Total time: 1 second
```

In *xsl:output* kann definiert werden, welche Elemente in *CDATA*-Section dargestellt werden sollen:

```
<xsl:output method="xml" indent="yes"
     cdata-section-elements="erg wert" />
```

CDATA-Sections und XML-Schema-Validierung

Häufig validieren Datenempfänger ihre XML-Dokumente "sehr scharf" gegen ein XML-Schema. Mitunter sind in den XML-Dokumenten jedoch *CDATA*-Sections enthalten, die Inhalte "maskieren", einer XSD-Validierung entziehen und so eine geforderte, eindeutige Strukturdefinition aushebeln.

Es bleibt dem Developer überlassen, bei eventuellen strukturellen Änderungen in den *CDATA*-Sections die Verarbeitungslogik (XSL, XQuery) anzupassen. Es kann aber vorkommen, dass diese Anpassung unterbleibt oder verzögert vorgenommen wird. Dann droht hier ein Informationsverlust, der sehr teuer werden kann.

Hierzu möchte ich ein einfaches Beispiel geben. Das folgende XML-Schema definiert ein Root-Element *Person* mit drei *xs:string*-Childnodes *Vorname*, *Nachname* und *info*.

```
<xs:schema
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  attributeFormDefault="unqualified"
  elementFormDefault="unqualified" version="1.0" >
  <xs:element name="Person" type="PersonTYP"/>
  <xs:complexType name="PersonTYP">
    <xs:sequence>
      <xs:element ref="Vorname" />
      <xs:element ref="Nachname" />
      <xs:element ref="info" />
    </xs:sequence>
  </xs:complexType>
  <xs:element name="Vorname" type="xs:string" />
  <xs:element name="Nachname" type="xs:string" />
  <xs:element name="info" type="xs:string" />
</xs:schema>
```

Das folgende XML-Dokument übernimmt *Vorname* und *Nachname*, weist der *info* aber einen zusätzlichen Childnode *Hobby* zu, das im XML-Schema nicht vorgesehen ist; ihre Existenz im XML-Dokument sollte daher zu einem Validierungsfehler führen.

```
<Person
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="Person.xsd">
  <Vorname>Vorname</Vorname>
  <Nachname>Nachname</Nachname>
  <info><Hobby>XML</Hobby></info>
</Person>
```

Der Validierungsfehler lässt auch nicht auf sich warten:

```
Element 'info' ist Simple Type und
darf daher keine Elementinformationselemente
[untergeordnete Elemente] haben.
Person.xml is not a valid XML document
```

Dagegen geht die *CDATA*-"Maskierung" glatt durch:

```
<Person
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="Person.xsd">
  <Vorname>Vorname</Vorname>
  <Nachname>Nachname</Nachname>
  <info><![CDATA[ <Hobby>XML</Hobby>]]> </info>
</Person>
```

Alternativ denkbar wäre, die Daten ohne *CDATA*-Section als Text mit Entitäten einzubinden; auch hier würde eine Validierung gegen das vorher beschriebene XML-Schema problemlos durchlaufen:

```
<Person
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="Person.xsd">
  <Vorname>Vorname</Vorname>
  <Nachname>Nachname</Nachname>
  <info>&lt;Hobby&gt;XML&lt;/Hobby&gt;</info>
</Person>
```

In beiden Fällen stellen sich aus meiner Sicht einige Fragen:

- Wozu diese Trickserei?
- Stimmt das XML-Schema (XSD) mit den aktuellen Anforderungen überein?
- Geht es um die provisorische Weiterverwendung eines veralteten XML-Schemas, das überarbeitet wird?
- Welche Konsequenzen ergeben sich für die Folgeprogrammierung?
- Kann der durch *CDATA* "maskierte" Bereich ignoriert werden?
- Falls nicht: Wie wird die Einhaltung der geforderten Datenqualität sichergestellt?
- Ist es notwendig, die Inhalte der *CDATA*-Section zu validieren?
- Werden die XML-Schemata zur automatischen Generierung objektorientierter Klassen (z.B. xsd.exe in .NET, xjc.exe für JAXB) verwendet? Welcher Zusatzaufwand ergibt sich für die Verarbeitung der *CDATA*-Infos?
- Droht Informationsverlust?

parse-xml-fragment

Obwohl *CDATA*-Kommentare nicht geparst werden und Inhalte betreffen können, die einer XML-Schema-Validierung widersprechen, ist es doch möglich, dass diese Inhalte von Belang sind. Die XPath-Funktion *parse-xml-fragment* erlaubt deren Auswertung.

```
 <xsl:variable name="vfeld">
  <![CDATA[
  <root>
    <i>Straße2</i>
    <i>Strasse1</i>
    <i>Weg</i>
    <i>Straße1</i>
    <i>Strasse2</i>
    <i>Pfad</i>
    </root>
   ]]>
 </xsl:variable>
```

Variablen wie *vfeld* können Sie mit der XPath-Funktion *parse-xml-fragment* in ein XML-Dokument überführen und dieses auswerten.

```
<ergebnis>
  <xsl:for-each
       select="parse-xml-fragment($vfeld)/root/i">
    <wert>
     <xsl:value-of select="."/>
    </wert>
  </xsl:for-each>
</ergebnis>
```

Daß dies gelungen ist, sehen Sie hier:

```
<ergebnis>
   <wert>Straße2</wert>
   <wert>Strasse1</wert>
   <wert>Weg</wert>
   <wert>Straße1</wert>
   <wert>Strasse2</wert>
   <wert>Pfad</wert>
</ergebnis>
```

XML: Processing-Instruction

Anweisungen zur Processing-Instruction helfen, XML-Dokumente mit temporären Zusatzinformationen zu versehen, die für die Weiterverarbeitung wichtig sind, ohne selbst Teil des XML-Dokuments zu sein. Beispielhaft sind hier browserseitige Transformationen von XML und XSL bzw. CSS.

Speziell bei komplexen XML-Konvertierungsstrecken ist es oft erforderlich, XML-Dokumente mit temporären Zusatzinformationen zu versehen, die bei der

Weiterverarbeitung benötigt werden, z.B. Headerinformationen, Verweise und Bezüge zu anderen Dokumenten, Empfängeradressen u.v.a.m., die aber nicht Bestandteile des XML-Dokuments werden sollen.

Da das XML-Dokument selbst inhaltlich nicht verändert werden darf (Dokumentschutz) oder bei der XML-Schema-Validierung ggf. auf einen Fehler laufen würde, behilft man sich oft damit, die Zusatzinformationen vor den Root-Node zu setzen. Dadurch verliert das XML-Dokument seine Wohlgeformtheit.

Im weiteren Konvertierungsprozess müssen daher jene Zusatzinformationen erst wieder entfernt werden, damit das XML-Dokument wieder wohlgeformt und valide ist und regulär verarbeitet werden kann.

Processing-Instructions (PI) machen diese Datenschnippelei durch externe Programme überflüssig. Mit ganz legalen Mitteln können einem XML-Dokument Zusatzinformationen mitgegeben und später ausgelesen werden, ohne das Dokument wiederholt verändern und die Wohlgeformtheit gefährden zu müssen.

```
<xsl:processing-instruction
  name="receiver">info10@wilfried-grupe.de</xsl:processing-instruction>
```

erzeugt folgenden Aufruf:

```
<?receiver info10@wilfried-grupe.de?>
```

PIs gehören selbst nicht zum XML-Dokument. Auch wenn sie optisch ähnlich aussehen mögen wie Attribute oder Prologe, so haben sie damit nichts zu tun. XML-Prozessoren ignorieren die PIs und leiten den Inhalt weiter. Um die PIs in XSLT wieder auslesen zu können, kann *processing-instruction()* verwendet werden.

```
<?receiver info10@wilfried-grupe.de?> <Books/>
```

Die Auswertung der PI ist in XSLT dann ganz einfach:

```
<xsl:value-of
     select="/processing-instruction('receiver')"/>
```

Der Output ist dann

```
info10@wilfried-grupe.de
```

Processing-Instructions sind in der XML Specification *https://www.w3.org/TR/REC-xml/#sec-pi* näher erläutert: "Processing instructions (PIs) allow documents to contain instructions for applications. ... PIs are not part of the document's character data, but MUST be passed through to the application."

Zahlreiche Anwendungen arbeiten mit diesem Konzept. Beeindruckend finde ich die Umsetzung in den XSLTForms, die clientseitig XForms mit XSLT kombinieren. Der Besuch von *http://www.agencexml.com/xsltforms.htm*, das Nachvollziehen der dort aufgeführten Beispiele sowie die ergänzenden Erläuterungen in *https://en.wikibooks.org/wiki/Category:XSLTForms* kann ich nur empfehlen.

```
<?xml-stylesheet href="xsltforms/xsltforms.xsl" type="text/xsl"?>
<?xsltforms-options debug="yes"?>
```

Verwendung finden die Processing-Instructions nicht zuletzt auch in der Verlagsbranche, etwa Zeitungsverlagen, wo bestimmte Platzhalter (Reiter, Dachzeile, Hauptzeile, Titel, Bildunterzeile, Quelle, Bildergalerie, Schlagwort, Vorspann, Autor) bereitgestellt werden, die im weiteren Konvertierungsverlauf verwendet werden. Eine Variante ist, diese Platzhalter durch Processing-Instructions zu definieren, die im Input-Text an verschiedenen Stellen auftauchen können.

Einen Überblick über die verwendeten Processing-Instructions eines XML-Dokuments kann man durch folgendes XSL gewinnen:

```
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="xml" indent="yes" />
  <xsl:template match="/">
    <root>
      <xsl:for-each
        select="descendant::*/processing-instruction()">
        <pi>
          <xsl:value-of select="name()" />|<xsl:value-of select="." />
        </pi>
      </xsl:for-each>
    </root>
  </xsl:template>
</xsl:stylesheet>
```

Soll der Name der Processing-Instruction selbst zu einem Element im Zieldokument werden, andere Elemente jedoch erhalten bleiben, so empfiehlt sich folgender Ansatz:

```
  <xsl:template match="p | stichwort | foto-quelle">
    <xsl:element name="{name()}">
      <xsl:apply-templates />
    </xsl:element>
  </xsl:template>
  <xsl:template match="processing-instruction()">
    <xsl:element name="{name()}">
      <xsl:value-of select="."></xsl:value-of>
    </xsl:element>
  </xsl:template>
  <xsl:template match="/">
    <erg>
      <xsl:apply-templates />
    </erg>
  </xsl:template>
```

Browserseitige Transformation XML und XSL

Alternativ zur direkten Generierung von HTML ist es auch möglich, die XSL-Transformation im Webbrowser durchführen zu lassen. Das wird möglich, wenn dem XML-Dokument über die Einbindung von Processing-Instructions ein Hinweis auf das XSL-Dokument zugewiesen wird (hier z.B. unter dem Namen "HTMLTabelle.xsl"). Der Browser führt die XSL-Transformation durch und zeigt das XML-Dokument entsprechend modifiziert an.

```
<?xml-stylesheet
    type="text/xsl"
    href="HTMLTabelle.xsl"
    version="1.0"
    encoding="iso-8859-1"?>
<Orte>
  <Ort>
    <id>1</id>
    <name>Neustadt</name>
    <!-- weitere Inhalte -->
  </Ort>
</Orte>
```

Das funktioniert freilich nicht in jedem Browser identisch. So habe ich diverse XML-Dokumente mit Hinweis auf das oben beschriebene XSL-Stylesheet in verschiedenen Browsern aufgerufen. Microsofts Internet Explorer (verschiedene Versionen) kam damit problemlos klar. Firefox führte die XSL-Transformation grundsätzlich zwar durch, ohne jedoch Templates auszuführen, die ihrerseits in das eingebundene XSL-Stylesheet "HTMLTabelle.xsl" importiert wurden.

Browserseitige Transformation XML und CSS

Ganz analog funktioniert der Browseraufruf bei der Kombination von XML und CSS. Durch die Processing-Instruction wird ein externes CSS-Dokument eingebunden, der Browser stellt das XML-Dokument nun entsprechend dar.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet
      type="text/css"
      href="Abfrageergebnis.css" ?>
<ROOT>
  <DS nr="1">
    <ID>5</ID>
    <Vorname>Willi</Vorname>
    <Nachname>Wasistlos</Nachname>
    <Wohnort>Kapstadt</Wohnort>
    <Artikel>Hemd</Artikel>
    <anzahl>44</anzahl>
    <EP>12,99</EP>
    <NETTO>571,56</NETTO>
  </DS>
  <DS nr="2">
    <ID>9</ID>
    <Vorname>Stefan</Vorname>
    <Nachname>Sprachlos</Nachname>
```

```
    <Wohnort>Neustadt</Wohnort>
    <Artikel>Hemd</Artikel>
    <anzahl>33</anzahl>
    <EP>12,99</EP>
    <NETTO>428,67</NETTO>
  </DS>
<ROOT>
```

Die dazu passende CSS-Datei lautet:

```
ROOT
 { position:absolute;
   top:45px;
   left:45px;
   background-color:#C0C0C0;
   padding:90px; }
DS
 { position:relative;
   display:block;
   width:600px;
   background-color:#FFFF80;
   color:#000000;
   font-family:Tahoma,Arial,Helvetica,sans-serif;
   font-size:12pt;
   padding:2px;
   vertical-align:top; }
ID,Vorname,Nachname
 { position:relative;
   width:70px; }
Wohnort
 { position:relative;
   color:#000FFF;
   width:360px;  }
Artikel, anzahl, EP, NETTO
 { font-weight:bold;
   color:0000E0; }
```

XML-Datenstrukturen

Das Datenformat XML ist außerordentlich gestaltungsflexibel. Das hat Konsequenzen für die Art der automatischen Verarbeitung.

Grundsätzlich lassen sich XML-Dokumente unterteilen in

- starke, datenzentrierte Strukturierung: XML-Elemente, Attribute etc. folgen einer klaren Strukturdefinition zur effizienten maschinellen Verarbeitung.
- schwache, dokumentenzentrierte Strukturierung, wo XML-Elemente und Attribute der semantischen Strukturierung dienen, was die systematische Auswertung erschwert.
- gemischte Strukturierung mit einem Mix aus klarer und semantischer Strukturdefinition (*mixed content*).

Jede dieser Strukturierungsalternativen erfordert andere Vorgehensweisen bei der systematischen Verarbeitung. Dieses Buch legt seinen Schwerpunkt auf starke, datenzentrierte Strukturierung der XML-Dokumente. Daher möchte ich hier nur kurz auf schwach strukturierte Dokumente oder auf Dokumente mit *mixed content* eingehen.

XML: Schwach strukturierte Dokumente

```
<?xml version="1.0" encoding="iso-8859-1"?>
<root>Die enorme Gestaltungsflexibilität
zwingt zu systematischer Strukturierung,
damit die XML-Dokumente systematisch
ausgewertet werden können.</root>
```

Abgesehen vom XML-Prolog und dem "<root>"-Element, ist das vorstehende XML-Dokument zwar wohlgeformt, aber im Übrigen unstrukturiert. Das wird auch nicht besser, wenn einige Tausend Sätze hinzukommen, die ebenfalls keine strukturierenden Elemente beinhalten. Es bleibt unserer Fantasie oder unserer Bereitschaft zur Spekulation überlassen, hier eine klare Struktur erraten zu wollen, die sich systematisch auswerten ließe. Gerade der völlige Mangel an systematischer Strukturierung zeigt, wie eingeschränkt die anschließende Auswertbarkeit ist.

Eine schwache Strukturierung kann beispielsweise entstehen, wenn einzelne Textteile hervorgehoben werden, etwa durch "... *damit die <hervorheben>XML</hervorheben>-Dokumente systematisch* ...". Dann haben Sie es mit einer gemischten Abfolge aus Textinhalt und Formatierungsanweisungen zu tun, deren Abfolge kaum vorhersehbar ist und die eine hohe Flexibilität bei der Programmierung erfordert.

Das wiederum lässt vermuten, dass Sie in diesem Umfeld mit der Anwendung von *xsl:for-each* Mühe haben könnten. Sinnvoller scheint die Arbeit mit *xsl:template match* bzw. *xsl:apply-templates*, die eine Aufsplittung der Programmierlogik in mehrere Templates mit sich bringt.

Solange der Überblick über die ggf. sehr zahlreichen (Hunderte, Tausende) Templates gewährleistet ist (optimal durch eine effiziente Koordination der Teamarbeit), sind keine Probleme für die flexible Anpassung der Programmlogik zu erwarten. Eine weniger qualitätsbewusste Teamarbeit birgt jedoch Gefahren der Unübersichtlichkeit und hoher Wartungskosten.

XML-Struktur: mixed content

Das folgende Beispiel zeigt einen Mix aus klarer Basisstruktur, die mit XPath gezielt adressiert werden kann (*Abschnitt/para*) und einer gemischten Abfolge aus Textinhalt und Formatierungsanweisungen (innerhalb *para*: *text()*, *kursiv*, *fett*, *link*), für deren systematische Auswertbarkeit eine hohe Flexibilität erforderlich ist.

Während es sich anbietet, die *Abschnitt/para*-Struktur durch xsl:for-each[21] zu verarbeiten, bleibt für die Formatierungsanweisungen innerhalb *para* vorrangig die Arbeit mit "xsl:template match" bzw. xsl:apply-templates[22]. So vielfältig wie die Input-Struktur dürfte auch die Programmierlogik bei deren Auswertung gestaltet werden. Auch hier empfiehlt sich eine effiziente, qualitätsbewusste Koordination der Teamarbeit, um die langfristigen Wartungskosten unter Kontrolle zu behalten.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<Abschnitt>
  <title>Namespaces</title>
  <para>Die enorme Gestaltungsflexibilität der
<kursiv>XML-Dokumente</kursiv> zwingt zu
<fett>systematischer Strukturierung</fett>,
damit die <kursiv>XML-Dokumente</kursiv>
systematisch ausgewertet werden können.</para>
<para>Aber auch eine klare systematische
Strukturierung von <kursiv><fett>Element- und
Attributnamen</fett></kursiv> sowie deren
<fett><kursiv>Datentypen</kursiv></fett>
(etwa durch <link
l="http://www.w3.org/2001/XMLSchema">XML-Schema</link>)
reicht nicht immer aus, begriffliche
Kollisionen zu vermeiden. </para>
</Abschnitt>
```

XML: Stark strukturierte Dokumente

Stark strukturierte XML-Dokumente (das Kernthema dieses Buches) haben häufig einen klaren, hierarchischen Aufbau, der sich entsprechend systematisch auswerten lässt.

Das ist aber keine Selbstverständlichkeit. Es kann schnell passieren, dass die Datenstruktur des XML-Input-Dokuments auf verschiedene Ebenen verteilt ist und gegenseitige Abhängigkeiten aufweist, die in der Programmierlogik mittels XPath mehr oder weniger aufwendig nachvollzogen werden müssen. Das wird spätestens dann ein Problem, wenn die internen Abhängigkeiten nicht im XML-Schema (und auch sonst nirgendwo) dokumentiert sind, sodass die automatisierte Kontrolle erschwert ist.

Ebenso kann es vorkommen, dass der strukturelle Aufbau des XML-Dokuments sich nur aus der sequenziellen Abfolge der einzelnen Elemente ergibt, ohne dass eine hierarchische Strukturierung vorliegt. Abhängig von diesen unterschiedlichen Datenstrukturen müssen auch unterschiedliche Verarbeitungskonzepte bereitstehen.

21) Seite: 416
22) Seite: 381

XML: Hierarchischer Aufbau

Der einfachste Ansatz ist das folgende XML-Dokument: Ein Root-Element *Orte* hat mehrere Child-Elemente *Ort* (jeweils mit *id* und *name*); jeder *Ort* hat mehrere Child-Elemente *Mensch* (mit *id*, *name*, *vorname*, *Gehalt* und *idOrt*), und jeder *Mensch* kann darüber hinaus noch mehrere Child-Elemente *Kauf* haben (mit den Child-Elementen *idMensch*, *anzahl*, *bez*, *preis* und *Gesamt*, die jeweils nur einmal auftreten können).

```
<?xml version="1.0" standalone="yes"?>
<Orte>
  <Ort>
    <id>1</id>
    <name>Neustadt</name>
    <Mensch>
      <id>1</id>
      <name>Holzflos</name>
      <vorname>Hugo</vorname>
      <Gehalt>234.56</Gehalt>
      <idOrt>1</idOrt>
      <Kauf>
        <idMensch>1</idMensch>
        <anzahl>3</anzahl>
        <bez>Hemd</bez>
        <preis>12.99</preis>
        <Gesamt>38.97</Gesamt>
      </Kauf>
    </Mensch>
  </Ort>
</Orte>
```

Diese komfortable hierarchische Struktur lässt sich gut mit XPath auswerten. Sie verdeutlicht das Zusammenspiel von XPath mit XSLT und deren jeweiligen Funktionen sehr gut. In diesem Beispiel lassen sich XPath-Statements wie

```
/Orte/Ort[name='Neustadt']/Mensch/Kauf[bez='Hemd']/Gesamt
```

auf einfache Weise nachvollziehen. Sie erbringt eine Elementliste sämtlicher *Gesamt*-Felder von Menschen, die in Neustadt wohnen und sich ein oder mehrere Hemden gekauft haben.

Das XML-Dokument entspricht einem XML-Schema, das Constraints (Beziehungen von Primär- und Fremdschlüsseln) sichtbar macht.

Index

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Z