

Kapitel 7

Umwandlungen mit XSLT

Bevor XML-Daten in einer bestimmten Form präsentiert werden, sind häufig Transformationen erwünscht, etwa um die Originaldaten neu zu ordnen oder zu filtern, um zwischen XML-Vokabularen zu übersetzen oder um XML-Daten in HTML- oder XHTML-Seiten zu übertragen.

Das W3C hat schon früh mit der Ausarbeitung eines Stylesheet-Standards begonnen, der für die Präsentation von XML-Dokumenten geeigneter sein sollte als die im letzten Kapitel beschriebenen Cascading Stylesheets. Im August 1997 entstand ein erster formaler Vorschlag für eine Extensible Stylesheet Language – XSL –, die die Erfahrungen mit der auf SGML basierenden Stylesheet-Sprache DSSSL nutzen sollte.

7.1 Sprache für Transformationen

Die Anforderungen an den neuen Standard wurden im Mai 1998 unter www.w3.org/TR/WD-XSLReq fixiert. Wie schon in DSSSL realisiert, sollten in dieser Sprache zwei Prozesse miteinander gekoppelt werden: die Transformation des Quelldokuments in das Ergebnisdokument und die Formatierung für die Ausgabe in dem gewünschten Medium.

7.1.1 Bedarf an Transformationen

Bedarf an Umwandlungen gegenüber dem XML-Quelldokument kann aus unterschiedlichen Gründen entstehen. Dazu zählt die Übersetzung von XML-Inhalten in HTML, XHTML oder auch in ein einfaches Textformat, die Übersetzung zwischen unterschiedlichen XML-Vokabularen oder Vorgänge wie die Sortierung, Abfrage und Filterung von Daten nach bestimmten Kriterien etc.

Die Art und Weise, wie die Daten in einem Quelldokument organisiert sind, muss nicht schon der Organisation entsprechen, die für eine Veröffentlichung in einem bestimmten Medium oder gegenüber einem bestimmten Publikum erforderlich ist. Wer mit Datenbanken arbeitet, kennt diese Aufgabenstellung im Zusammenhang mit Abfragesprachen und Reportgeneratoren.

Eine bestimmte Transformation von Inhalten ist immer noch völlig unabhängig von Formatierungsaspekten, wenn sie verwendet wird, um zwischen XML-Vokabularen zu übersetzen, oder wenn es nur darum geht, eine bestimmte Auswahl, Neuordnung oder auch Erweiterung in Bezug auf die Quelldaten vorzunehmen. Bei der Umwandlung von XML in HTML vermischen sich allerdings inhaltliche und formale Elemente in einem gewissen Umfang.

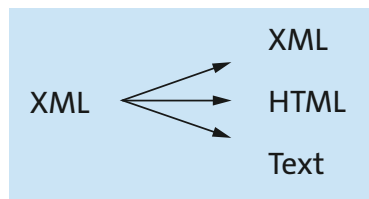


Abbildung 7.1 Transformationsrichtungen, die XSLT ermöglicht

Die für solche Aufbereitungen und Umwandlungen benötigten Sprachelemente wurden relativ schnell zur Reife entwickelt und in der Empfehlung für *XSL Transformations (XSLT) Version 1.0* bereits im November 1999 verabschiedet. Dagegen dauerte das ehrgeizige Projekt, alle möglichen Formatierungsoptionen in einem einheitlichen Vokabular zu beschreiben, wesentlich länger. Es firmierte längere Zeit unter dem Arbeitstitel *XSL Formatting Objects – XSL-FO*. Seit Oktober 2001 liegt die Empfehlung für XSL, gewissermaßen das Elternelement von XSLT und XSL Formatting Objects, auf dem Tisch. Im Dezember 2006 folgte die Version *XSL 1.1*, die noch einige Erweiterungen wie Änderungsmarkierungen, Lesezeichen etc. bereitstellt.

Im Januar 2007 hat das W3C schließlich eine neue Empfehlung für *XSL Transformations (XSLT)* herausgebracht. *XSLT 2.0* wurde entwickelt, um die Gestaltungsmöglichkeiten durch Stylesheets zu erweitern und um einige Schwächen von XSLT 1.0 zu beseitigen. Da die Unterstützung für XSLT 1.0 immer noch wesentlich umfangreicher ist als die der neuen Version, wird in diesem Kapitel zunächst XSLT 1.0 vorgestellt, daran anschließend die Erweiterungen, die die Version 2.0 bereitstellt. XSLT 2.0 ist zudem in hohem Grade rückwärtskompatibel.

Die Arbeit an XSLT 3.0 wurde im Juni 2017 von dem sehr fleißigen Michael Kay abgeschlossen, der über seine 2004 gegründete Firma Saxonica auch maßgeblich an der Entwicklung entsprechender Prozessoren beteiligt ist.

Der wesentliche Schritt, den diese neue Version bringt, ist die Möglichkeit, Transformationen im Streaming-Modus auszuführen. Dabei muss weder das Quelldokument noch das Zieldokument komplett im Hauptspeicher gehalten werden, was bei sehr großen Dokumenten natürlich vorteilhaft ist. Außerdem lassen sich umfangreiche Stylesheets nun aus Modulen zusammensetzen, die an unterschiedlichen Stellen entwickelt werden können. XSLT 3.0 unterstützt die für XPath 3.0 und XPath 3.1

beschriebenen Funktionen, Operatoren und Datentypen, ebenso das entsprechende Datenmodell.

Obwohl XSLT in XSL als notwendiger Bestandteil eingeschlossen ist, kann die Sprache auch auf eigenen Füßen stehen und hat seit ihrer Implementierung als Standard auch eine entsprechende Rolle gespielt. Werkzeuge, die XSLT unterstützen, sind längst etabliert, während die Unterstützung der XSL insgesamt immer noch eher bescheiden ausfällt. Das verwundert auch nicht, wenn man einen Blick auf den Umfang des Unternehmens wirft. In diesem Kapitel soll XSLT in mehreren Anwendungsmöglichkeiten vorgestellt werden; die speziellen Formatierungsoptionen von XSL werden darauf aufbauend im folgenden Kapitel behandelt.

Den Schwerpunkt in den folgenden Abschnitten bildet zunächst die Arbeit mit XSLT 1.0 in Kombination mit XPath 1.0. Die beiden Standards werden seit Jahren von zahlreichen Prozessoren und Tools zuverlässig unterstützt.

7.1.2 Grundlegende Merkmale von XSLT

XSLT ist eine in XML definierte Sprache, die ihrerseits den Prozess der Umwandlung eines XML-Dokuments in ein anderes XML-Dokument oder ein anderes Ausgabeformat beschreibt. Das Quelldokument und das Ergebnisdokument, das mit Hilfe des Stylesheets erzeugt wird, können dabei beliebig weit voneinander abweichen. Im Extremfall könnte ein Stylesheet sogar ganz auf eine Quelldatei verzichten und einfach selbstherrlich festlegen, was in das Ergebnisdokument geschrieben werden soll.

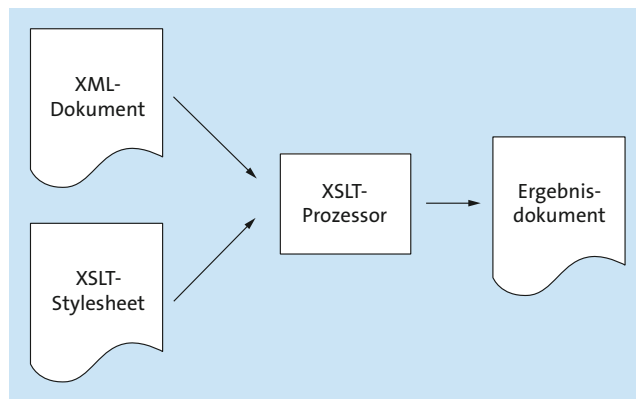


Abbildung 7.2 Generelles Schema der Transformation durch XSLT

Das Stylesheet mag beispielsweise nur einen Teil der Informationen aus dem Quelldokument beachten und das Ergebnisdokument um beliebig viele Informationen erweitern, die nicht im Quelldokument vorhanden sind. Dabei arbeitet XSLT immer nur mit dem in einem XML-Dokument vorhandenen Markup; DTDs und Schemas spielen für XSLT keine Rolle. Die Quelldatei muss wohlgeformt sein, mehr nicht.

XSLT-Stylesheets können deshalb nicht auf HTML-Dateien angewendet werden, wohl aber auf XHTML-Dateien, weil diese ja dem XML-Format entsprechen. Es kann also sinnvoll sein, entsprechende Konvertierungen von Websites nach XHTML vorzunehmen, um die Möglichkeiten von XSLT zu nutzen. Die Transformation wird codiert in Form von XSLT-Stylesheets, die selbst wohlgeformte XML-Dokumente sind.

7.1.3 XSLT-Prozessoren

Damit die Transformation stattfindet, muss ein spezieller XSLT-Prozessor vorhanden sein, der das Stylesheet und die Quelldaten einliest, die im Stylesheet enthaltenen Anweisungen ausführt und das dementsprechende Ergebnisdokument erzeugt. Dieser Prozessor setzt die Anweisungen des Stylesheets selbstständig in die dafür notwendigen Bearbeitungsschritte um. XSLT ist in diesem Sinne also eine High-Level-Language, bei der sich der Entwickler nicht um die kruden Details zu kümmern braucht, anders als bei den Lösungen, die in Kapitel 10, »Programmierschnittstellen für XML«, zu DOM und SAX behandelt werden.

Dabei sind mehrere Wege möglich. Die Anwendung des Stylesheets auf das XML-Quelldokument kann bereits auf einem Webserver ausgeführt werden, der dann das fertige Ergebnisdokument für die Clientseite bereitstellt. Das Stylesheet kann aber auch direkt auf der Clientseite mit Hilfe eines geeigneten Browsers oder eines separaten Programms angewendet werden.

Ein bekannter Open-Source-XSLT-Prozessor ist *Xalan*, der von der *Apache Software Foundation* in einer Version für Java und einer Version für C++ über xalan.apache.org/ bereitgestellt wird. Xalan unterstützt nur die W3C-Empfehlung zu XSLT 1.0 und XPath 1.0 vollständig.

Teilweise frei verfügbar sind auch die XSLT-Prozessoren mit dem Namen *Saxon*, die von Michael Kay entwickelt wurden. Die Open-Source-Variante wird in der aktuellen Version Saxon XSLT 3.0, XQuery 3.1 und XPath 3.1 über <http://sourceforge.net/projects/saxon/> sowohl für die Java- als auch für die .NET-Plattform angeboten.

XML-Entwicklungsumgebungen wie XMLSpy, Stylus Studio oder Oxygen können auch mit eigenen, integrierten XSLT-Prozessoren arbeiten oder vorhandene Prozessoren einbinden.

Am einfachsten ist die Situation, wenn der verwendete Internetbrowser einen solchen Prozessor bereitstellen kann. Dann lassen sich XSLT-Stylesheets genauso einer XML-Datei zuordnen, wie es bei HTML und CSS möglich ist, und direkt im Browser testen.

Microsoft Core XML Services MSXML unterstützt XSLT 1.0 seit der Version 3. Seit der Version 6 des Internet Explorers, die als Vorgabe noch MSXML 3.0 verwendet, las-

sen sich XML-Dateien direkt darstellen, die mit XSLT-Stylesheets verknüpft sind. Seit Windows 7 ist MSXML bereits integriert, muss also nicht zusätzlich installiert werden.

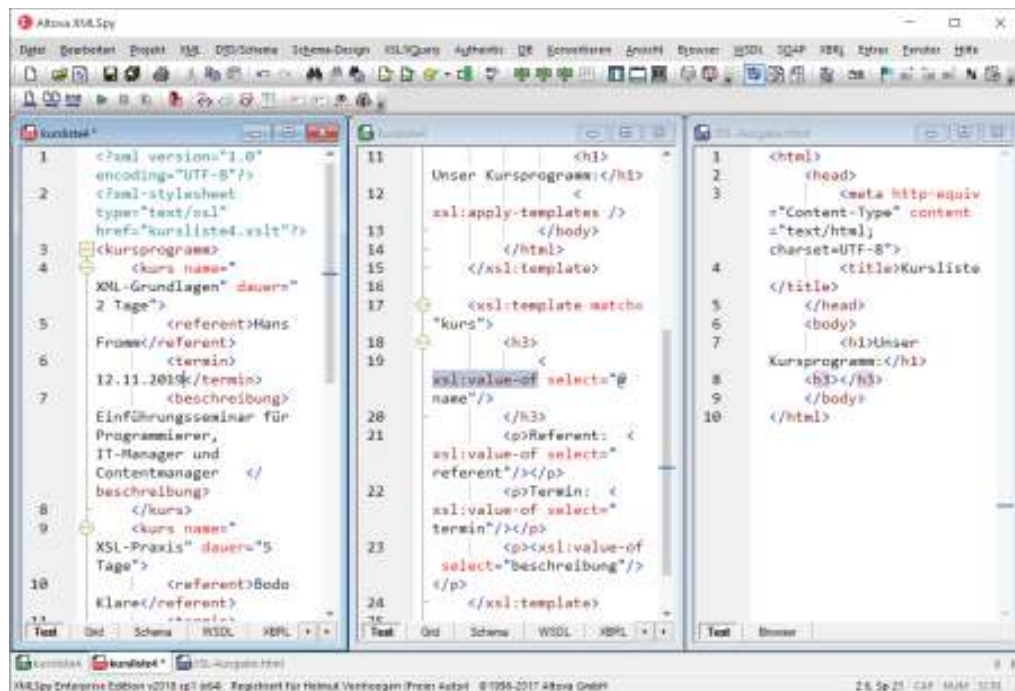


Abbildung 7.3 Beim Debuggen eines Stylesheets mit XMLSpy kann der Aufbau des Ergebnisdokuments Schritt für Schritt beobachtet werden.

7.1.4 Die Elemente und Attribute von XSLT

Die Elemente und Attribute, die XSLT verwendet, um dem XSLT-Prozessor seine Anweisungen zu geben, müssen mit dem speziellen Namensraum übereinstimmen, der für XSLT gesetzt ist – <http://www.w3.org/1999/XSL/Transform>. Das Stylesheet kann zusätzliche Elemente und Attribute enthalten, die nicht zu diesem Namensraum gehören. Diese Komponenten werden dann in der Regel vom XSLT-Prozessor direkt zum Ergebnisbaum hinzugefügt.

Zusätzlich sind spezielle Erweiterungselemente und Erweiterungsfunktionen möglich, die in einem eigenen Namensraum definiert sein müssen, damit sie vom XSLT-Prozessor ebenfalls als Anweisungen interpretiert werden. Allerdings kommt es in diesem Fall darauf an, ob diese Erweiterungen auch in dem verwendeten XSLT-Prozessor implementiert sind.

Eine beschreibende Sprache

Obwohl XSLT eine Reihe von Elementen enthält, die sich auch in allgemeinen Programmiersprachen finden – etwa Elemente für die Bildung von Verzweigungen oder Schleifen –, handelt es sich doch im Wesentlichen um eine beschreibende Sprache, die dem XSLT-Prozessor gewissermaßen in Form von Beispielen sagt, was er tun soll: »Wenn du auf ein Element mit dem Namen ›Thema‹ triffst, dann schreibe es in das Ergebnisdokument.« Das Wie bleibt dabei den eingebauten Funktionen des XSLT-Prozessors überlassen, derer sich das Stylesheet einfach bedient.

Vom XML-Dokument zur Webseite

Wir nehmen hier als erstes Beispiel eine etwas abgewandelte Version der bereits verwendeten Kursliste:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="kursliste4.xslt"?>
<kursprogramm>
  <kurs name="XML-Grundlagen" dauer="2 Tage">
    <referent>Hans Fromm</referent>
    <termin>12.11.2019</termin>
    <beschreibung>Einführungsseminar für Programmierer,
      IT-Manager und Contentmanager</beschreibung>
  </kurs>
  <kurs name="XSL-Praxis" dauer="5 Tage">
    <referent>Bodo Klare</referent>
    <termin>10.10.2019</termin>
    <beschreibung>Praktische Übungen für Programmierer und
      Contentmanager</beschreibung>
  </kurs>
  <kurs name="XSLT-Einstieg" dauer="2 Tage">
    <referent>Hanna Horn</referent>
    <termin>12.03.2019</termin>
    <beschreibung>Einführung in die Transformation von
      XML-Dokumenten für Programmierer und Contentmanager
    </beschreibung>
  </kurs>
</kursprogramm>
```

Listing 7.1 kursliste4.xml

Um daraus eine einfache Liste auf einer Webseite zu erzeugen, lässt sich das folgende XSLT-Stylesheet verwenden:

```

<?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>
      <head>
        <title>Kursliste</title>
      </head>
      <body>
        <h1>Unser Kursprogramm:</h1>
        <xsl:apply-templates select="/kursprogramm/kurs"/>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="kurs">
    <h3>
      <xsl:value-of select="@name"/>
    </h3>
    <p>Referent: <xsl:value-of select="referent"/></p>
    <p>Termin: <xsl:value-of select="termin"/></p>
    <p><xsl:value-of select="beschreibung"/></p>
  </xsl:template>

</xsl:stylesheet>

```

Listing 7.2 kursliste4.xslt

7.1.5 Verknüpfung zwischen Stylesheet und Dokument

Damit ein Browser die XML-Quelldaten so ausgibt, wie im Stylesheet vorgesehen, muss in die Quelldatei eine entsprechende Verknüpfung eingebaut werden. Dies geschieht mit Hilfe einer entsprechenden Verarbeitungsanweisung, die in diesem Fall so aussieht:

```

<?xml-stylesheet type="text/xsl" href="kursliste4.xslt"?>

```

Der Browser gibt das formatierte XML-Dokument, wie in [Abbildung 7.4](#) zu sehen, als Liste aus.

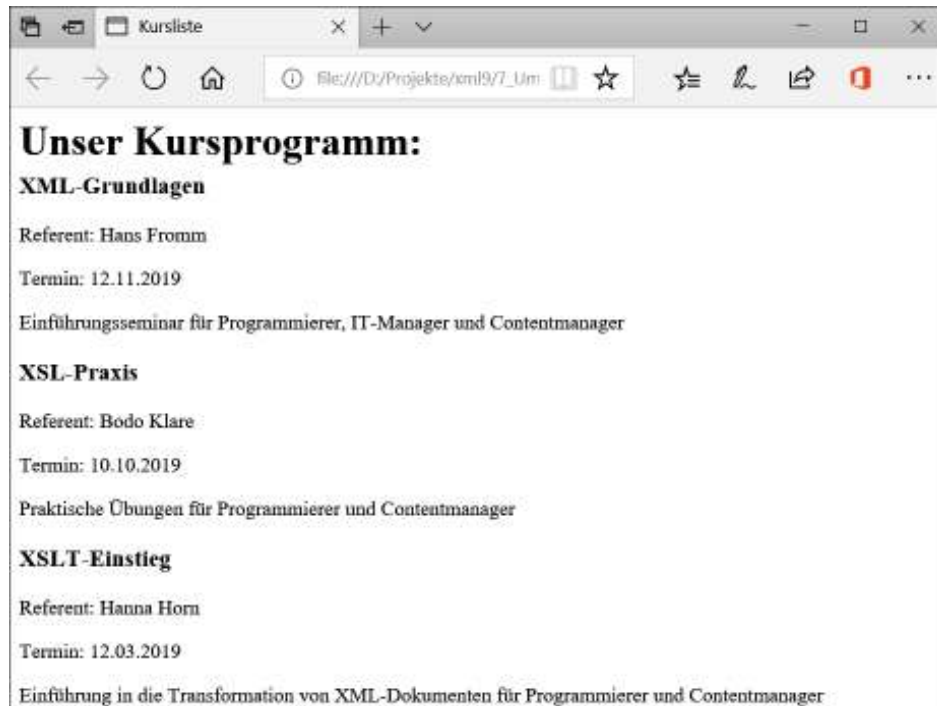


Abbildung 7.4 Ausgabe des formatierten Dokuments in Microsoft Edge

7.1.6 Das Element `<stylesheet>`

Das Stylesheet beginnt, da es sich um ein XML-Dokument handelt, mit der XML-Deklaration. Anschließend werden sämtliche Komponenten in das vorgegebene Wurzelement `<xsl:stylesheet>` eingefügt. Als Synonym kann auch `<xsl:transform>` verwendet werden, was sinnvoll sein kann, um Anwendungen, die nur Daten transformieren, von solchen zu unterscheiden, die auch auf die Formatierung ausgerichtet sind. Es gilt für XSLT die Konvention, dass alle zu XSLT gehörenden Element-, Attribut- und Funktionsnamen kleingeschrieben werden.

Das Element `<xsl:stylesheet>` muss ein `version`-Attribut enthalten, für XSLT 1.0 gilt also der Wert 1.0. Ebenso notwendig ist, wie schon angesprochen, die Zuordnung des Namensraums, der die vorgegebenen Elemente und Attribute umfasst. Der URI des XSLT-Namensraums ist `http://www.w3.org/1999/XSL/Transform`; als Vorgabe wird das Präfix `xsl` verwendet, aber es steht Ihnen frei, auch andere Abkürzungen zu nutzen.

7.1.7 Top-Level-Elemente

Abbildung 7.5 zeigt die möglichen Kinder des Elements `<xsl:stylesheet>`. Sie werden *Top-Level-Elemente* genannt.


```

<xsl:stylesheet>
  <xsl:import/>
  <xsl:include/>
  <xsl:namespace-alias/>
  <xsl:strip-space/>
  <xsl:preserve-space/>
  <xsl:output/>
  <xsl:key/>
  <xsl:decimal-format/>
  <xsl:attribute-set/>
  <xsl:variable/>
  <xsl:param/>
  <xsl:template/>
</xsl:stylesheet>

```

Abbildung 7.5 Die möglichen Kinder des Elements `<stylesheet>`

Diese Elemente dürfen in beliebiger Reihenfolge im Stylesheet auftauchen, ausgenommen sind allerdings `<xsl:import>`-Elemente, die immer am Anfang stehen müssen. Der Anwender kann auch eigene Top-Level-Erweiterungselemente verwenden, etwa für Metadaten zu einem Stylesheet, wenn dafür ein eigener Namensraum definiert wird. Der XSLT-Prozessor kann solche Elemente aber ignorieren.

Häufig wird am Anfang des Stylesheets das Element `<xsl:output>` eingefügt, um die Methode festzulegen, die bei der Erzeugung des Ergebnisdokuments beachtet werden soll. Im Beispiel wird `method = "html"` benutzt. Allerdings ist das nur eine Aufforderung an den XSLT-Prozessor, die nicht bindend sein muss. XSLT 1.0 unterstützt die in [Tabelle 7.1](#) genannten Werte für das Attribut `method`.

Wert	Beschreibung
xml	Erzeugt wohlgeformtes XML.
html	Sorgt dafür, dass korrekt verwendete HTML-Elemente und -Attribute gemäß der HTML-Version 4.0 erkannt werden.
text	Gibt die String-Werte aller Textknoten aus, die im Ausgabebaum enthalten sind, und zwar in der Reihenfolge, die der Dokumentreihenfolge entspricht.

Tabelle 7.1 Mögliche Werte für das Attribut »method«

Wird das Element `<xsl:output>` nicht verwendet, benutzt der XSLT-Prozessor entweder `xml` oder `html` als Vorgabe, je nachdem, ob er `<html>` als erstes Element findet oder nicht.

7.1.8 Template-Regeln

Die wichtigsten Anweisungen an den XSLT-Prozessor sind in dem XSLT-Stylesheet in einer freien Abfolge von Template-Regeln zusammengestellt, die jeweils in das Element `<xsl:template>` eingepackt sind. Die im Beispiel verwendeten Regeln bestehen aus zwei Komponenten: einem Suchmuster oder Pattern, das dafür verwendet wird, im Quelldokument die Komponenten zu finden, mit denen etwas gemacht werden soll, und dem Template selbst, das aus einer oder mehreren Anweisungen besteht, die in allen Fällen, wo das Suchmuster passt, ausgeführt werden sollen.

Alternativ dazu können Template-Regeln auch mit einem Namen versehen werden, über den sie dann von anderen Templates aus aufgerufen werden können.

In dem Beispiel wird in der ersten Template-Regel als Suchmuster `match="/"` verwendet, das heißt, der Prozessor soll diese Regel anwenden, wenn er den Wurzelknoten der Quelldatei vor sich hat. Die Zeilen zwischen dem `<xsl:template>`-Start- und -End-Tag sind das eigentliche Template, das dem Prozessor beschreibt, was zu tun ist, wenn beim Durchwandern des Knotenbaums ein Knoten gefunden worden ist, der exakt dem Suchmuster entspricht.

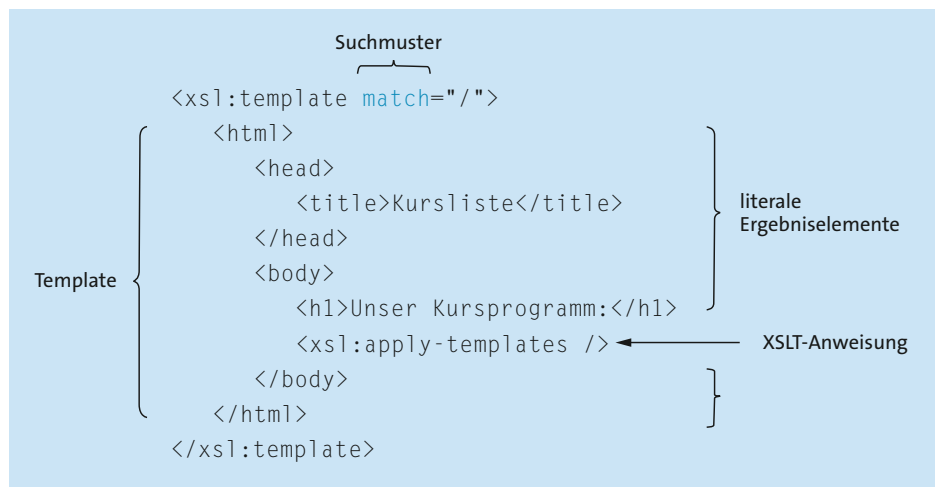


Abbildung 7.6 Beispiel einer Template-Regel

XSLT-Anweisungen

Innerhalb des Templates kommen zwei unterschiedliche Dinge vor. Das eine sind XSLT-Elemente mit Anweisungen an den XSLT-Prozessor, erkennbar daran, dass sie dasselbe Namensraumpräfix haben wie das Elternelement `<xsl: template>`. In dem ersten Template ist es die Anweisung `<xsl:apply-templates/>`, gewissermaßen eine Generalanweisung, die in dem Stylesheet vorhandenen Template-Regeln auszuführen.

Wie bedeutend diese Anweisung ist, lässt sich leicht testen, wenn Sie sie in dem Beispiel einfach einmal vergessen oder aber testweise auskommentieren. In der Ausgabedatei finden Sie dann nur noch die Überschrift, die Liste mit den Daten ist jedoch verschwunden.

Literale Ergebniselemente

Zum anderen enthält das erste Template eine Reihe von HTML-Tags. Im Stylesheet sind in diesem Fall die für die HTML-Seite notwendigen Elemente `<html>`, `<head>`, `<body>` und `<h1>` für die Listenüberschrift bereits in die erste Template-Regel eingefügt, die der Prozessor ausführen wird. Dies geschieht auf sehr schlichte Weise. Die entsprechenden HTML-Elemente und -Attribute werden einfach genau so, wie sie in der späteren HTML-Seite benötigt werden, eingetragen.

Da diese Elemente nicht zum Namensraum von XSLT gehören – das entsprechende Präfix fehlt ja offensichtlich –, schreibt der Prozessor diese Teile des Templates mehr oder weniger unverändert, also wortwörtlich, in das Ergebnisdokument. Diese Elemente werden deshalb auch *literale Ergebniselemente* genannt. Die Einschränkung »mehr oder weniger« bezieht sich darauf, dass literale Ergebniselemente Attribute enthalten können, die anstelle fixer Werte XPath-Ausdrücke enthalten, die erst während der Verarbeitung ausgewertet werden.

7.1.9 Attributwert-Templates

Solche Attribute werden als *Attributwert-Templates* behandelt. Der Prozessor wertet ein solches Template aus und überträgt erst das Ergebnis in die Ausgabe. Der Prozessor erkennt solche Attributwert-Templates daran, dass sie in geschweiften Klammern eingeschlossen sind. Dieses Verfahren erlaubt es beispielsweise, Werte aus der Quelldatei in die Attribute von HTML-Elementen zu übernehmen.

Unser Kursprogramm könnte zum Beispiel mit den Porträts der Referenten verschönert werden, indem die zweite Template-Regel in folgender Weise erweitert wird:

```
<xsl:template match="kurs">
  <h3><xsl:value-of select="@name"/></h3>
  <p></p>
  <p>Referent: <xsl:value-of select="referent"/></p>
  <p>Termin: <xsl:value-of select="termin"/></p>
  <p><xsl:value-of select="beschreibung"/></p>
</xsl:template>
```

Listing 7.3 kursliste4f.xslt

Der Prozessor ersetzt jedes Mal, wenn er die Template-Regel anwendet, in dem `src`-Attribut des ``-Elements den Teil, der in geschweiften Klammern steht. Er sucht also jeweils den String-Wert des jeweiligen Elements `<referent>`.

Voraussetzung dafür ist, dass die Bilddateien jeweils mit Dateinamen belegt sind, die aus dem Wort *portrait* und dem Namen der Person zusammengesetzt sind, zum Beispiel *portrait Hans Fromm.gif*.



Abbildung 7.7 Das Ergebnis des Templates mit einigen Musterbildern

Attributwert-Templates können als Attributwerte bei einer begrenzten Zahl von XSLT-Elementen erscheinen, so bei den Attributen `lang`, `data-type`, `order` und `case-order` in `<xsl:sort>`, bei Attributen von `<xsl:number>` und bei den `name-` oder `namespace-`Attributen von `<xsl:attribute>`, `<xsl:element>` und `<xsl:processing-instruction>`. In der XSLT 1.0-Kurzreferenz weiter unten sind die entsprechenden Attribute durch geschweifte Klammern gekennzeichnet.

7.1.10 Zugriff auf die Quelldaten

Die zweite Template-Regel soll für den Fall angewendet werden, dass der Prozessor Knoten mit dem Namen `kurs` findet. Jedes Mal, wenn dies der Fall ist, wird zunächst mit dem HTML-Element `<h3>` eine Zwischenüberschrift erzeugt. An dieser Stelle

nutzt der XSLT-Prozessor zum ersten Mal Daten aus dem Quelldokument. Er holt sich über das XSLT-Element `<xsl:value-of>` den Namen des jeweiligen Kurses. Ähnlich wie bei dem `match`-Attribut des Elements `<xsl:template>` wird für das hier benutzte Attribut `select` ein XPath-Ausdruck ausgewertet. `select="@name"` greift auf den String-Wert des Attributs `name` zu und kopiert ihn an die Stelle des Ergebnisdokuments, die als Zwischenüberschrift formatiert ist.

Ähnlich wird verfahren, um den Textinhalt der anderen Elemente in die Ausgabe zu übernehmen. Zusätzlich werden einfach die Beschriftungen »Referent« und »Termin« in die `<p>`-Tags eingefügt.

7.2 Ablauf der Transformation

Es ist für die Entwicklung von XSLT-Anwendungen von großer Bedeutung, im Blick zu behalten, wie ein XSLT-Prozessor ein Stylesheet abarbeitet. Wenn ein XSLT-Prozessor beauftragt wird, ein spezielles Stylesheet auf ein XML-Quelldokument anzuwenden, um daraus schließlich ein neues Ergebnisdokument zu erzeugen, liest und parst der Prozessor zunächst beide Dokumente und baut im Speicher für beide Dokumente jeweils eine interne Baumstruktur auf, wie es schon für XPath beschrieben worden ist. Das Stylesheet benutzt ja XPath-Ausdrücke, um Knoten in dieser Baumrepräsentation des Quelldokuments aufzusuchen.

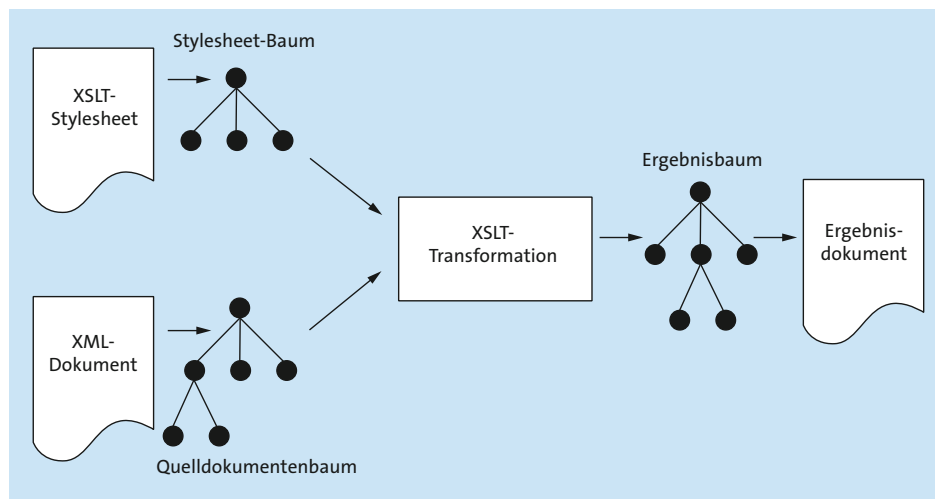


Abbildung 7.8 Schema der Transformation im Detail

Erst dann wird der Transformationsprozess gestartet. Während der Abarbeitung des Stylesheets wird in dem Umfang, den die Anweisungen der Template-Regeln erfordern, auf die Quelldaten zugegriffen. Die verwendeten Quelldaten und die literalen

Ergebniselemente liefern den Stoff, aus dem der Ergebnisbaum aufgebaut wird. Aus diesem Ergebnisbaum werden schließlich durch eine entsprechende Serialisierung die Ergebnisdokumente erzeugt, die das Stylesheet anfordert: eine neue XML-Datei, eine HTML-Datei oder eine einfache Textdatei.

7.2.1 Startpunkt Wurzelknoten

Wenn die Transformation beginnt, nimmt sich der XSLT-Prozessor zunächst immer die Wurzel des XML-Baums vor und verarbeitet die Template-Regel, die für diesen Wurzelknoten vorhanden ist. Es spielt deshalb keine Rolle, in welcher Reihenfolge Template-Regeln in einem Stylesheet eingetragen sind, der Prozessor sucht immer zunächst nach einer Template-Regel, die explizit für den Wurzelknoten definiert worden ist oder sich dafür verwenden lässt.

Findet er keine solche Regel, nutzt er normalerweise eine entsprechende eingebaute Template-Regel, wie es in [Abschnitt 7.4](#) noch beschrieben wird. Sind gleich mehrere konkurrierende Regeln vorhanden, tritt ein Konfliktlösungsmechanismus in Aktion, der in [Abschnitt 7.2.4](#) behandelt wird.

In unserem Beispiel existiert eine explizite Template-Regel für den Wurzelknoten. Der Prozessor beginnt den Aufbau des Ergebnisbaums also damit, die Anweisungen auszuführen, die in dieser Template-Regel enthalten sind. Zuerst werden die HTML-Tags samt Inhalt in den Baum übertragen.

7.2.2 Anwendung von Templates

Nach der Übertragung des Start-Tags für das `<body>`-Element stößt der Prozessor auf die XSLT-Anweisung `<xsl:apply-templates/>`. Erst muss diese Anweisung ausgeführt werden, bevor das `<body>`-Element geschlossen werden kann, denn die Anweisung soll in diesem Fall den Inhalt des `<body>`-Elements liefern.

Da diese Anweisung innerhalb der Template-Regel für den Wurzelknoten auftritt, der damit den maßgeblichen Kontext für diese Anweisung bildet, und sie in diesem Fall keine näheren Angaben durch ein mögliches `select`-Attribut enthält, entspricht sie einer Aufforderung an den XSLT-Prozessor, alle Kindknoten des aktuellen Knotens auszuwählen und jedes Mal nach einer Template-Regel zu suchen, die darauf passt, und diese Regel anzuwenden.

Der Wurzelknoten hat nur ein Kind, den Dokumentknoten, dem das Wurzelement des XML-Dokuments `<kursprogramm>` entspricht.

Für diesen Knoten ist aber in unserem Beispiel – mit einer gewissen Absicht – keine Template-Regel angegeben, sondern erst für dessen Kindelement `<kurs>`. Sie erwarten vielleicht, dass der Prozessor in diesem Fall nicht mehr weiterweiß. Die Situation

erinnert an ein Programm, in dem sich eine Unterroutine befindet, die aber, weil die Entwickler es vergessen haben, niemals aufgerufen wird.

7.2.3 Rückgriff auf versteckte Templates

In XSLT ist das anders. Da eine Template-Regel für das Kind des Wurzelknotens fehlt, verwendet der XSLT-Prozessor – zumindest ist das der Normalfall – eine eingebaute Regel, die nicht nur auf den Wurzelknoten, sondern auf alle Elementknoten zutrifft, und die einfach noch einmal die Anweisung `<xsl:apply-templates/>` aufruft. (Diese eingebauten Regeln werden in [Abschnitt 7.4](#) im Detail besprochen.)

Nun aber hat sich der Kontext für die zweite `<xsl:apply-templates/>`-Anweisung gegenüber der ersten Anweisung geändert. Der Kontextknoten ist jetzt nicht mehr der Wurzelknoten, sondern dessen Kind. Also findet der Prozessor jetzt mehrere Kinder des `<kursprogramm>`-Elements, nämlich die einzelnen `<kurs>`-Elemente. Er baut eine entsprechende Knotenliste auf, um sie abzuarbeiten, und nun findet er auch ein spezielles Template, das auf jeden einzelnen dieser `<kurs>`-Knoten nacheinander angewendet wird.

Da jeder Kursknoten selbst wiederum Kinder hat, wird geprüft, ob es für diese Knoten ebenfalls Templates gibt. Das ist hier nicht der Fall, weil die Daten aus diesen Knoten bereits in der `kurs`-Template-Regel übernommen worden sind.

Wenn der Prozessor diese Prüfung abgeschlossen hat, ist hier zugleich der Knotenbaum des Quelldokuments durchlaufen. Der Prozessor kehrt zu der Zeile in der Template-Regel für den Wurzelknoten zurück, die der Anweisung `<xsl:apply-templates/>` folgt, und kann nun das End-Tag des `<body>`-Elements und des `<html>`-Elements in die Ausgabe einfügen.

7.2.4 Auflösung von Template-Konflikten

Es kommt vor, dass gleich mehrere Templates für denselben Knoten zutreffen. In dieser Situation treten bestimmte Prioritätsregeln in Kraft:

- ▶ Eine Regel für eine spezifischere Information hat Vorrang vor einer Regel für eine allgemeinere Information. `match="/firma/abteilung/person/name"` ist zum Beispiel spezifischer als `match="//name"`.
- ▶ Suchmuster mit Wildcards wie `*` oder `@*` sind allgemeiner als entsprechende Muster ohne Wildcards.
- ▶ Wenn keines der aufgeführten Kriterien hilft (und nur dann), spielt die Reihenfolge der Template-Regeln im Stylesheet die Rolle des Schiedsrichters. Die zuletzt stehende Regel hat dann Vorrang.

- Die Priorität von Templates kann auch gezielt über das Attribut `priority` gesetzt werden. Dabei werden Ganzzahlen oder reelle Zahlen unter oder über dem Standard 0 verwendet, etwa -1.2 oder 2.2. Eine höhere Zahl liefert eine höhere Priorität.

7.3 Stylesheet mit nur einer Template-Regel

Die Template-Regel, die für die Wurzel gilt, enthält meist den Aufruf weiterer Regeln, die für einzelne Teile des Dokuments maßgeblich sind. Ein Stylesheet kann aber auch aus nur einer einzigen Template-Regel bestehen. In diesem Fall ist eine verkürzte Fassung des Stylesheets möglich. Hier ein kleines Beispiel:

```
<?xml version="1.0" encoding="UTF-8"?>
<html xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
      xsl:version="1.0"
      <head>
        <title>Kursliste</title>
      </head>
      <body>
        <h1>Unser Kursprogramm:</h1>
        <xsl:for-each select="/kursprogramm/kurs">
          <h3>
            <xsl:value-of select="."/>
          </h3>
        </xsl:for-each>
      </body>
    </html>
```

Listing 7.4 kursliste6.xslt

In diesem Fall werden die Angabe zu `xsl:version` und die Einbindung des Namensraums direkt in das `<html>`-Start-Tag eingefügt.

7.4 Eingebaute Template-Regeln

Die wichtigste eingebaute Template-Regel ist oben schon angesprochen worden. Wenn eine Regel fehlt, die sich explizit auf den Wurzelknoten bezieht, verwendet der Prozessor automatisch eine für diesen Zweck eingebaute Template-Regel.

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


Sie können dies testen, indem Sie ein Stylesheet auf eine Quelldatei anwenden, das noch gar keine Template-Regeln enthält. Der Prozessor gibt dann einfach die String-Werte aller Elemente des Quelldokuments aus, wobei die Attribute ignoriert werden. Jede explizite Template-Regel zu / überschreibt die eingebaute Regel, soweit sie sich auf den Wurzelknoten bezieht.

Diese Regel trifft nämlich nicht nur auf den Wurzelknoten zu, sondern auf alle beliebigen Elementknoten. Sie sorgt für eine rekursive Verarbeitung, also dafür, dass, wenn eine Template-Regel für bestimmte Knoten existiert, diese auch ausgeführt wird, auch wenn ihre Anwendung nicht explizit aufgerufen wird.

Eine weitere eingebaute Template-Regel ist für die Verwendung von unterschiedlichen Anwendungsmodi vorgesehen, die in [Abschnitt 7.6.4](#) behandelt werden. Sie sorgt dafür, dass die Knoten unabhängig vom gerade gültigen Modus verarbeitet werden:

```
<xsl:template match="*/" mode="m">
  <xsl:apply-templates mode="m"/>
</xsl:template>
```

Die eingebaute Regel für Text- und Attributknoten

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

kopiert den Text aller Text- und Attributknoten in den Ausgabebaum. Diese Regel wird aber nur dann ausgeführt, wenn die Knoten ausgewählt werden.

Die folgende Regel sorgt lediglich dafür, dass Kommentare und Verarbeitungsanweisungen nicht in das Ergebnisdokument übernommen werden. Sie ist leer, das heißt, der Prozessor soll in diesem Fall einfach nichts tun.

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

Wenn Sie die Kommentare im Ergebnisdokument sehen wollen, definieren Sie eine Template-Regel, die die eingebaute einfach überschreibt:

```
<xsl:template match="comment()">
  <xsl:comment>
    <xsl:value-of select="."/>
  </xsl:comment>
</xsl:template>
```

Diese Regel sorgt beispielsweise dafür, dass der Inhalt eines Kommentars im Ergebnisdokument erneut als Kommentar eingefügt wird, also eingeschlossen in `<!--` und `-->`.

All diese Template-Regeln haben eine geringere Priorität als alle explizit im Stylesheet eingefügten Regeln, kommen also nur zum Zuge, wenn entsprechende explizite Regeln fehlen.

7.5 Designalternativen

Die oben beschriebene Version eines Stylesheets für das Kursprogramm ist nur eine der Lösungen, die mit XSLT realisierbar sind. In dem Beispiel sind zwei Methoden für den Zugriff auf den Inhalt von Knoten gemischt. In der ersten Template-Regel wird die Behandlung der Knoten unterhalb des Wurzelknotens gewissermaßen an ein anderes Template abgestoßen, in der zweiten Regel dagegen werden die Daten der Kinder des aktuellen Knotens direkt mit Hilfe von `value-of` in das Template einbezogen. Die erste wird deshalb manchmal auch *Push-Processing*, die andere *Pull-Processing* genannt.

Eine Variante des Stylesheets, die stärker die Push-Methode verwendet, könnte so aussehen:

```
<?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>
      <head>
        <title>Kursliste</title>
      </head>
      <body>
        <h1>Unser Kursprogramm:</h1>
        <xsl:apply-templates />
      </body>
    </html>
  </xsl:template>

  <xsl:template match="kurs">
    <h3><xsl:value-of select="@name" /></h3>
    <xsl:apply-templates />
  </xsl:template>
```

```

<xsl:template match="referent">
  <p>Referent: <xsl:value-of select="."/></p>
</xsl:template>

<xsl:template match="termin">
  <p>Termin: <xsl:value-of select="."/></p>
</xsl:template>

<xsl:template match="beschreibung">
  <p><xsl:value-of select="."/></p>
</xsl:template>

</xsl:stylesheet>

```

Listing 7.5 kursliste8.xslt

In dieser Version ist für jeden Elementknoten im Quelldokument eine eigene Template-Regel definiert. Die Regeln für die jeweiligen Kindknoten werden über die Anweisung `<xsl:apply-templates />` stufenweise aktiviert. Dieses Vorgehen bietet sich an, wenn der Ergebnisbaum weitgehend über dieselbe Struktur wie das Quelldokument verfügt. Die Inhalte der Elemente `<referent>`, `<termin>` und `<beschreibung>` werden jedes Mal mit der Anweisung `<xsl:value-of select="."/>` übernommen, wobei der Punkt als Abkürzung für den Kontextknoten verwendet wird. Da es sich hier um die Knoten am Ende des Baums handelt, liefert diese Anweisung die Blätter des Baums, den Inhalt der Textknoten.

Würde `<xsl:value-of select="."/>` dagegen bei einem Kontextknoten verwendet, der selbst noch Elemente als Kinder hat, ergäbe die Anweisung eine Zeichenkette, die die String-Werte der Textknoten dieser Kinder aneinanderreihet. Eine Instanz einer Template-Regel wie

```

<xsl:template match="kurs">
  <xsl:value-of select="." />
</xsl:template>

```

würde beispielsweise folgende Zeichenkette liefern:

Hans Fromm 12.11.2019 Einführungsseminar für Programmierer, IT-Manager und Contentmanager

Obwohl die Reihenfolge der Template-Regeln im Prinzip keine Bedeutung hat, liegt es nahe, beim Entwurf des Stylesheets der Struktur des Quelldokuments zu folgen, wie es hier geschehen ist. Dies macht das Ganze übersichtlicher und lesbarer. Dies

gilt zumindest in all den Fällen, in denen sich das Ergebnisdokument weitgehend an der Struktur des Quelldokuments orientiert.

7.6 Kontrolle der Knotenverarbeitung

Die in den bisherigen Beispielen verwendete Anweisung `<xsl:apply-templates/>` ist eine sehr generelle Aufforderung an den Prozessor, den Knotenbaum zu durchwandern, um nach Knoten zu sehen, für die Templates definiert sind. Bei einem Quelldokument mit einer tiefen Hierarchie, aus der nur bestimmte Informationen in das Ergebnisdokument übernommen werden sollen, und dazu vielleicht noch in einer anderen Anordnung, ist es effektiver, mit Hilfe des `select`-Attributs die Zielknoten der Anweisung genau oder wenigstens genauer zu bestimmen.

Wenn beispielsweise in der Kursliste die Beschreibung vor die Terminangabe gesetzt werden soll, können Sie die Template-Regel für das Element `<kurs>` folgendermaßen angeben:

```
<xsl:template match="kurs">
  <h3><xsl:value-of select="@name"/></h3>
  <xsl:apply-templates select="referent"/>
  <xsl:apply-templates select="beschreibung"/>
  <xsl:apply-templates select="termin"/>
</xsl:template>
```

Listing 7.6 kursliste9.xslt

Das Attribut `select` benötigt als Wert einen XPath-Ausdruck, der die Knotenmenge beschreibt, die verarbeitet werden soll. Der Ausdruck muss also immer eine Knotenmenge liefern. Ohne `select` werden jeweils alle Kinder des aktuellen Knotens ausgewählt, dies entspricht `select="child::node()"`, wobei zu dieser Knotenmenge alle Knoten vom Typ Element, Text, Verarbeitungsanweisung und Kommentar gehören, aber nicht die Attribute des aktuellen Knotens, die ja von XPath – wie schon erwähnt – nicht als Kindknoten behandelt werden. Dabei können auch Wildcards eingesetzt werden, wie das folgende Listing zeigt.

```
<xsl:apply-templates select="*" />
```

wählt alle Kindknoten vom Typ Element bezogen auf den aktuellen Kontextknoten aus. Der XPath-Ausdruck kann anstelle von relativen Lokalisierungspfaden auch mit absoluten Pfaden arbeiten.

```
<xsl:apply-templates select="/kursprogramm/kurs/referent" />
```

ist ein solcher Pfad, der eine Auswahl von Knoten erlaubt, die unabhängig vom gerade aktuellen Kontextknoten ist. Auch der Operator // lässt sich in einem absoluten Pfad verwenden:

```
<xsl:apply-templates select="/kursprogramm//referent"/>
```

Gesucht wird dann nach einer Knotenmenge `referent`, die den Knoten `kursprogramm` als Vorfahren hat. Mit Hilfe von Funktionen kann die Suche weiter verfeinert werden.

7.6.1 Benannte Templates

Ein Stylesheet kann auch Templates verwenden, die sich nicht mit dem `match`-Attribut gewissermaßen die Opfer suchen, auf die sie sich stürzen, um daraus einen Teil des Ausgabebaums zu fabrizieren, sondern solche, die mit eindeutigen QNames gekennzeichnet sind. Über diese Namen können sie von anderen Templates aus aufgerufen werden. Diese Vorgehensweise ist dem Aufruf von Funktionen in prozeduralen Sprachen ähnlich. Dabei können auch Parameter an das aufgerufene Template übergeben werden. Mehr dazu lesen Sie weiter unten in [Abschnitt 7.11](#), »Parameter und Variablen«.

Ein wichtiger Unterschied zu den Templates, die mit Suchmustern arbeiten, ist, dass der Kontextknoten nicht durch das aufgerufene Template verändert wird. Es bleibt der Kontext erhalten, der zu der aufrufenden Template-Regel gehört. Unser Beispiel kann bei diesem Verfahren so aussehen:

```
<xsl:template match="/">
  <html>
    <head>
      <title>Kursliste</title>
    </head>
    <body>
      <h1>Unser Kursprogramm:</h1>
      <xsl:call-template name="kursliste"/>
    </body>
  </html>
</xsl:template>

<xsl:template name="kursliste">
  <xsl:for-each select="//kurs">
    <h3><xsl:value-of select="@name"/></h3>
    <p>Referent: <xsl:value-of select="referent"/></p>
    <p>Termin: <xsl:value-of select="termin"/></p>
```

```

        <p><xsl:value-of select="beschreibung"/></p>
    </xsl:for-each>
</xsl:template>

```

Listing 7.7 kursliste10.xslt

Die Template-Regel für den Wurzelknoten ruft hier ein Template mit Namen `kursliste` auf, das zunächst vollständig ausgeführt werden muss, bevor die übrigen Anweisungen der ersten Regel zum Zuge kommen. Anstelle einer `apply-templates`-Aufforderung an den Prozessor wird hier eine andere Methode verwendet, um an die Daten der Knoten zu gelangen. Mit `<xsl:for-each>` wird der Prozessor angewiesen, die Knotenliste vollständig durchzugehen, die durch den Lokalisierungspfad bestimmt ist, der den Wert von `select` liefert. Da in diesem Fall der Wurzelknoten immer noch den aktuellen Kontext bildet, wird ein absoluter Lokalisierungspfad `//kurs` angegeben, um die Knoten unabhängig vom aktuellen Kontextknoten zu finden.

7.6.2 Template-Auswahl mit XPath-Mustern

In dem ersten Beispiel haben wir für die beiden Template-Regeln zwei sehr einfache `match`-Attribute benutzt: `match="/"` für den Wurzelknoten und `match="kurs"` für eines der Elemente. Das Suchmuster enthält die Bedingung, die erfüllt sein muss, damit ein Knoten als Kandidat für die Anwendung des Templates ausgewählt wird. Dabei muss jedes Mal der Kontext beachtet werden, in dem die Prüfung auf Übereinstimmung stattfindet.

Für die Formulierung des Musters steht nur eine Untermenge der XPath-Sprache, die in [Kapitel 5](#), »Navigation und Verknüpfung«, beschrieben wurde, zur Verfügung, im Unterschied zu den `select`-Kriterien, die Elemente wie `<apply-templates>` oder `<value-of>` verwenden.

Ein XSLT-Muster kann einen oder auch mehrere Lokalisierungspfade enthalten, allerdings dürfen nur die `child`- und die `attribute`-Achse benutzt werden. Außerdem ist der `//`-Operator erlaubt, nicht aber die `descendent`-Achse insgesamt. Solche Muster werden außer im `match`-Attribut des `<template>`-Elements auch für das `match`-Attribut des `<key>`-Elements und in den `count`- und `from`-Attributen des Elements `<number>` verwendet. [Tabelle 7.2](#) zeigt einige Muster.

Muster	Bedeutung
<code>kurs</code>	Passt auf alle <code><kurs></code> -Elemente.
<code>kurs/referent</code>	Passt auf jedes Element <code><referent></code> , das ein Kind von <code><kurs></code> ist.

Tabelle 7.2 Beispiele für Muster

Muster	Bedeutung
<code>kursprogramm/kurs[1]</code>	Passt auf das erste Element <code><kurs></code> , das Kind von <code><kursprogramm></code> ist.
<code>kurs[referent="Hanna Horn"]</code>	Passt auf jedes Element <code><kurs></code> , bei dem der String-Wert des Kindes <code><referent></code> dem angegebenen Wert entspricht.
<code>kursprogramm/kurs[position()=1]</code>	Passt auf jedes Element <code><kurs></code> , das Kind von <code><kursprogramm></code> ist und nicht das erste Kind ist.
<code>kurs [@name="XSL-Praxis"]</code>	Passt auf das Element <code><kurs></code> , bei dem das Attribut <code>name</code> den angegebenen Wert hat.

Tabelle 7.2 Beispiele für Muster (Forts.)

Mit Hilfe solcher Muster lässt sich die Zuordnung von Templates sehr fein auf die Besonderheiten bestimmter Elemente abstimmen. Beispielsweise erlaubt das folgende Stylesheet-Fragment, bei einigen Kursen zusätzliche Hinweise einzubauen:

```
<xsl:template match="kurs">
  <h3><xsl:value-of select="@name"/></h3>
  <xsl:apply-templates />
</xsl:template>

<xsl:template match="kurs[referent='Hanna Horn']">
  <h3>Neu im Programm: <xsl:value-of select="@name"/></h3>
  <xsl:apply-templates />
</xsl:template>

<xsl:template match="kurs [@name='XSL-Praxis']">
  <h3>Wochenendseminar: <xsl:value-of select="@name"/></h3>
  <xsl:apply-templates />
</xsl:template>
```

Die Templates mit dem spezifischeren Muster werden in diesem Fall vorrangig vor dem Muster für `kurs` verwendet. Wenn sich ein Template gleich für unterschiedliche Knoten verwenden lässt, können Sie auch mit einem `match`-Wert arbeiten, der mehrere Werte zulässt, wie das folgende Listing zeigt.

```
<xsl:template match="referent|termin">
```

wäre ein Beispiel dafür.

7.6.3 Kontext-Templates

Ähnlich wie bei den Kontextselektoren, die in [Abschnitt 6.1](#), »Cascading Stylesheets für XML«, beschrieben worden sind, lassen sich Elementen unterschiedliche Templates zuordnen, je nach dem Kontext, in dem sie auftauchen. Das ist immer dann von Bedeutung, wenn auf unterschiedlichen Ebenen der Hierarchie für Elemente gleiche Namen verwendet werden. Das folgende Listing zeigt einen Ausschnitt aus einer etwas modifizierten Version der Kursliste:

```
<kursprogramm>
  <kurs>
    <name>XML-Grundlagen</name>
    <referent>
      <name>Hans Fromm</name>
      <email/>
    </referent>
    <termin>12.11.2019</termin>
    <beschreibung>Einführungsseminar für Programmierer,
      IT-Manager und Contentmanager</beschreibung>
    </kurs>
  </kursprogramm>
```

Die Anweisung

```
<xsl:template match="name">
```

würde nacheinander auf beide Elemente angewendet. Sollen die beiden Elemente dagegen unterschiedlich ausgegeben werden, muss das `match`-Kriterium entsprechend differenziert werden:

```
<xsl:template match="kurs/name">
<xsl:template match="referent/name">
```

7.6.4 Template-Modi

Es kommt vor, dass dieselben Quelldaten einmal so und einmal anders benötigt werden. An das ausführliche Kursprogramm kann zum Beispiel noch eine kurze Übersicht angehängt werden. Um Templates, die dasselbe `match`-Kriterium nutzen, dennoch unterscheiden zu können, wird das Attribut `mode` eingesetzt. Den Template-Regeln muss über dieses Attribut ein eindeutiger Modus zugeordnet werden, der dann auch beim Aufruf der Template-Regel verwendet wird.

```
<?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>
    <head>
      <title>Kursliste</title>
    </head>
    <body>
      <h1>Unser Kursprogramm:</h1>
      <xsl:apply-templates select="//kurs" mode="langfassung"/>
      <h4>Übersicht</h4>
      <xsl:apply-templates select="//kurs" mode="kurzfassung"/>
    </body>
  </html>
</xsl:template>

<xsl:template match="kurs" mode="langfassung">
  <h3><xsl:value-of select="@name"/></h3>
  <p>Referent: <xsl:value-of select="referent"/></p>
  <p>Termin: <xsl:value-of select="termin"/></p>
  <p><xsl:value-of select="beschreibung"/></p>
</xsl:template>

<xsl:template match="kurs" mode="kurzfassung">
  <table width="180" border="1" cellspacing="2" cellpadding="2">
    <tr>
      <td width="120" height="60">
        <xsl:value-of select="@name"/>
      </td>
      <td width="60" height="60">
        <xsl:value-of select="termin"/>
      </td>
    </tr>
  </table>
</xsl:template>

</xsl:stylesheet>

```

Listing 7.8 kursliste14.xslt

Abbildung 7.9 zeigt das Ergebnis.

Unser Kursprogramm:	
XML-Grundlagen	
Referent: Hans Fromm	
Termin: 12.11.2019	
Einführungseminar für Programmierer, IT-Manager und Contentmanager	
XSL-Praxis	
Referent: Bodo Klare	
Termin: 10.10.2019	
Praktische Übungen für Programmierer und Contentmanager	
Übersicht	
XML-Grundlagen	12.11.2019
XSL-Praxis	10.10.2019

Abbildung 7.9 Kursliste mit angehängter Übersicht

7.7 Datenübernahme aus der Quelldatei

In den bisherigen Beispielen sind Daten aus der Quelldatei hauptsächlich mit Hilfe der Anweisung `<xsl:value-of>` in das Ergebnisdokument übernommen worden. Dabei kann über das `select`-Attribut sehr präzise bestimmt werden, welche Daten übernommen werden sollen. Durch entsprechende XPath-Ausdrücke kann auf jedes beliebige Element oder Attribut innerhalb des Quelldokuments zugegriffen werden. Der Ausdruck wird bezogen auf den gerade aktuellen Kontextknoten ausgewertet, sofern nicht mit absoluten Lokalisierungspfaden gearbeitet wird, und liefert eine entsprechende Zeichenkette, also den String-Wert.

Neben dem notwendigen Attribut `select` kann das Attribut `disable-output-escaping` verwendet werden. Vorgabe ist "no", so dass sichergestellt ist, dass Sonderzeichen wie `<`, `&` und `>` auch in der Ausgabe maskiert bleiben, also als `<`, `&` und `>` ausgegeben werden. Der Wert "yes" bewirkt, dass die Zeichen selbst – unmaskiert – erscheinen, was aber nur sinnvoll ist, wenn das Ergebnisdokument kein gültiges XML-Dokument mehr sein soll.

Sollen gleich ganze Teilbäume aus dem Quelldokument in ein neues Dokument übertragen werden, bietet sich die Anweisung `<xsl:copy-of>` an. Damit kann die Knotenliste, die über das `select`-Attribut der Anweisung bestimmt wird, vollständig aus

dem Quelldokument in die Ausgabe kopiert werden. Das Ergebnis hängt dabei von dem aktuellen Knotentyp ab. Im Unterschied zu `<xsl:value-of>` kopiert `<xsl:copy-of>` die ausgewählte Knotenliste und nicht den String-Wert der Auswahl. Das Markup bleibt also erhalten.

Diese Anweisung wird hauptsächlich bei der Transformation von XML nach XML verwendet, wenn es darum geht, Teile des Quelldokuments unverändert zu übernehmen. In dem folgenden Beispiel werden alle `<kurs>`-Elemente der Quelldatei in eine Datei kopiert, bei der diese Elemente in ein neues Wurzelement `<schulungsprogramm>` eingefügt werden.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <schulungsprogramm>
      <xsl:copy-of select="//kurs"/>
    </schulungsprogramm>
  </xsl:template>
</xsl:stylesheet>
```

Listing 7.9 kursliste4d.xslt

Der folgende Auszug zeigt die kopierten Daten als Abkömmlinge des neuen Wurzelements:

```
<schulungsprogramm>
  <kurs name="XML-Grundlagen" dauer="2 Tage">
    ...
  </kurs>
  ...
</schulungsprogramm>
```

Die verwandte Anweisung `<xsl:copy>` kopiert nur den aktuellen Knoten im Quelldokument, ohne die Attribute oder die Kinder des Knotens zu berücksichtigen.

7.8 Nummerierungen

XSLT gibt Ihnen die Möglichkeit, die aus dem Quelldokument übernommenen Daten in fast jeder denkbaren Weise auch nachträglich zu nummerieren, wobei wiederum zahlreiche Formate zur Verfügung stehen. Das Element `<xsl:number>` enthält dafür eine Reihe von optionalen Attributen, die hier nicht alle im Detail behandelt werden können; sie sind in [Abschnitt 7.18](#), »Kurzreferenz zu XSLT 1.0«, aufgelistet. Die Anweisung kann innerhalb eines Templates verwendet werden.

7.8.1 Einfach

In der einfachsten Form, ohne jedes Attribut, liefert das Element eine fortlaufende Nummer für alle Elemente der Knotenliste, zu der der Kontextknoten gehört.

```
<xsl:template match="kurs">
  <p>
    <xsl:number/>
    <xsl:text> </xsl:text>
    <xsl:value-of select="@name" />
  </p>
</xsl:template>
```

Das liefert die folgende schlichte Liste der Kurse:

- 1 XML-Grundlagen
- 2 XSL-Praxis
- 3 XSLT-Einstieg

Derselbe Effekt kann in diesem Fall übrigens auch erreicht werden, wenn zu jedem Knoten das Ergebnis der Funktion `position()` ausgegeben wird.

7.8.2 Mehrstufig

Die Attribute für `<xsl:number>` erlauben aber auch komplexe Nummerierungen, die mit mehreren Ebenen arbeiten, wie es zum Beispiel bei Textdokumenten oder hierarchischen Verzeichnissen üblich ist. Maßgeblich für die Art, wie der Prozessor beim Durchzählen vorgeht, ist insbesondere das Attribut `level`, das drei Werte kennt (siehe [Tabelle 7.3](#)).

Wert	Beschreibung
single	(Vorgabe) Zählt Geschwisterknoten, die auf einer Ebene liegen.
any	Dieser Wert wird verwendet, um Knoten zu nummerieren, die auf verschiedenen Ebenen vorkommen können, etwa Fußnoten, Tabellen oder Bilder in einem Text.
multiple	Erlaubt die Bildung von zusammengesetzten Nummerierungen, wie sie in Dokumenten verwendet werden, wie etwa 1.2.1 oder I.1.a.

Tabelle 7.3 Werte für das Attribut »level«

Anstatt jeweils alle Elemente einer Ebene der Dokumenthierarchie zu zählen, kann mit dem Attribut `count` auch die Übereinstimmung mit einem XPath-Muster geprüft werden. Die Syntax von XPath-Mustern wurde oben schon beschrieben. Zusätzlich

kann mit `from` ein Muster verwendet werden, das einen anderen Startpunkt für das Abzählen setzt.

Das Verfahren des Abzählens, das der Prozessor bei dieser Anweisung zu bewältigen hat, sieht im Prinzip so aus, dass er bei jeder Nummer, die zu vergeben ist, vom gerade aktuellen Kontextknoten aus die `ancestor-or-self`-Achse zurückgeht, um den ersten Knoten zu finden, der dem expliziten oder vorgegebenen `count`-Muster entspricht.

Falls `from` einen Wert hat, stoppt der Prozessor bereits an der dem `from`-Muster entsprechenden Stelle. Wenn er beispielsweise sechs Knoten auf dieser Achse findet, zählt er 1 hinzu und vergibt die Nummer 7 an den aktuellen Kontextknoten.

Für die Definition des Zahlenformats werden Format-Token verwendet, die den Wert des Attributs `format` liefern. Eine Liste der Format-Token sehen Sie in [Tabelle 7.4](#).

Format-Token	Ausgabesequenz
1	1, 2, 3, 4 ...
01	01, 02 ... 10, 12 ...
a	a, b, c, d ...
A	A, B, C, D ...
i	i, ii, iii, iv ...
I	I, II, III, IV ...

Tabelle 7.4 Werte für das Attribut »format«

7.8.3 Zusammengesetzt

Zur Demonstration wollen wir das Kursbeispiel so erweitern, dass im XML-Dokument Kursprogramme für verschiedene Veranstaltungsorte aufgenommen werden können. Dazu erweitern wir das Element `<kursprogramm>` um ein Attribut `ort` und verklammern die verschiedenen Kursprogramme unter einem neuen Wurzelement `<schulungsprogramm>`:

```
<schulungsprogramm>
  <kursprogramm ort="Hamburg">
    <kurs name="XML-Grundlagen" dauer="2 Tage">
      <referent>Hans Fromm</referent>
      <termin>12.11.2019</termin>
    ...
  </kursprogramm>
</schulungsprogramm>
```

```
...
</kursprogramm>
</schulungsprogramm>
```

Das Stylesheet benötigt dann zwei Template-Regeln, um die doppelte Nummerierung zu leisten:

```
<xsl:template match="kursprogramm">
  <h3>Seminare in <xsl:value-of select="@ort"/>: </h3>
  <xsl:apply-templates/>
</xsl:template>

<xsl:template match="kurs">
  <p>
    <xsl:number level="multiple" count="kursprogramm|kurs"
               format="I.1" />
    <xsl:value-of select="@name"/>
  </p>
</xsl:template>
```

Listing 7.10 kursliste16.xslt

Das Attribut `count` enthält in diesem Fall eine Musterverknüpfung, die dafür sorgt, dass sowohl die Kursprogramme als auch die einzelnen Kurse gezählt und die Nummerierungen jeweils aus beiden Werten zusammengesetzt werden. Im Ergebnisdokument erscheint die in [Abbildung 7.10](#) dargestellte Liste.

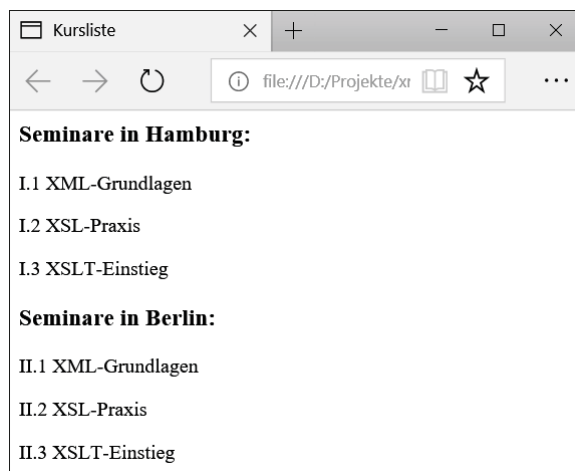


Abbildung 7.10 Liste mit Mehrfachnummerierung im Edge-Browser

7.9 Verzweigungen und Wiederholungen

Mit den Anweisungen `<xsl:if>`, `<xsl:choose>` und `<xsl:when>` lässt sich der Ablauf der Operationen, die der Prozessor vornimmt, zusätzlich beeinflussen.

7.9.1 Bedingte Ausführung von Templates

Die `<xsl:if>`-Anweisung ist sehr einfach gehalten. Sie wird verwendet, um die Ausführung eines Templates an eine Bedingung zu knüpfen. Dieses Template wird direkt in Form von Kindelementen des Elements `<xsl:if>` eingefügt. Das so gleichsam eingeklammerte Template wird nur ausgeführt, wenn die angegebene Bedingung erfüllt ist. Diese Bedingung wird mit Hilfe des Attributs `test` formuliert. Der dabei genutzte XPath-Ausdruck muss einen logischen Wert liefern.

Soll zum Beispiel eine Liste ausgegeben werden, die nur die Kurse eines bestimmten Referenten enthält, kann eine entsprechende Bedingung in die Template-Regel eingefügt werden:

```
<xsl:template match="kurs">
  <xsl:if test="referent='Bodo Klare'">
    <h3><xsl:value-of select="@name"/></h3>
    <p>Referent: <xsl:value-of select="referent"/></p>
    <p>Termin: <xsl:value-of select="termin"/></p>
    <p><xsl:value-of select="beschreibung"/></p>
  </xsl:if>
</xsl:template>
```

Für die Formulierung der Testkriterien können insbesondere auch die zahlreichen Funktionen genutzt werden, die in XPath-Ausdrücken verwendet werden können. Um beispielsweise die Kurse eines bestimmten Jahrgangs gesondert auszugeben, könnte mit einer Anweisung wie

```
<xsl:if test="substring(termin,10,1)='3'">
```

die letzte Stelle des Datums abgefragt werden, wenn es als String vorliegt.

7.9.2 Wahlmöglichkeiten

Die in den meisten Sprachen mögliche Variante `if-else` ist in XSLT nicht vorhanden. Wenn weitere Wahlmöglichkeiten benötigt werden, kann mit der Anweisung `<xsl:choose>` gearbeitet werden, die als Kindelemente für die verschiedenen Fälle mindestens ein oder mehrere `<xsl:when>` Elemente verwendet. Zusätzlich kann ein Default-Fall mit dem Element `<xsl:otherwise>` definiert werden, das ohne `test`-Attribut arbeitet.

`<xsl:choose>` ähnelt der `switch`-Anweisung in Java oder C#. Jedes `<xsl:when>`-Element enthält ein `test`-Attribut, das einen logischen XPath-Ausdruck auswertet, und als Kindelemente die Template-Anweisungen, die ausgeführt werden sollen, wenn der Test den Wert `true` liefert.

Im Unterschied zur angesprochenen `switch`-Anweisung wird es zugelassen, dass mehrere Testausdrücke in `<xsl:when>`-Elementen gleichzeitig wahr sein können. Es wird aber nur das Template des ersten Elements ausgeführt, das den Test besteht. Deshalb ist es in diesem Fall von Bedeutung, in welcher Reihenfolge die `<xsl:when>`-Elemente angeordnet sind. Eine Anordnung wie

```
<xsl:when test="count(//referent) <4">...
<xsl:when test="count(//referent) =1">...
```

würde dazu führen, dass der zweite Test niemals erreicht wird. Die umgekehrte Reihenfolge ließe dagegen beide Tests zu. Wird keiner der `when`-Tests bestanden, wird ausgeführt, was innerhalb des Elements `<xsl:otherwise>` genannt ist, falls dieses Element angegeben ist. Andernfalls geschieht nichts, und die Anweisung hinter `</xsl:choose>` wird bearbeitet.

In unserem Kursbeispiel könnten die Kurse zum Beispiel entsprechend nach der dafür vorgesehenen Dauer unterschiedlich behandelt werden, um je nach Anzahl der Seminartage verschiedene zusätzliche Informationen in das Ergebnisdokument einzufügen:

```
<xsl:template match="kurs">
  <h3><xsl:value-of select="@name"/></h3>
  <p>Referent: <xsl:value-of select="referent"/></p>
  <p>Termin: <xsl:value-of select="termin"/></p>
  <p><xsl:value-of select="beschreibung"/></p>
  <xsl:choose>
    <xsl:when test="@dauer='2 Tage'">
      Das Seminar dauert 2 Tage.
    </xsl:when>
    <xsl:when test="@dauer='5 Tage'">
      Das Seminar dauert 5 Tage.
    </xsl:when>
    <xsl:otherwise>
      Eintägiges Seminar
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

Listing 7.11 kursliste4c.xslt

Unser Kursprogramm:
XML-Grundlagen
Referent: Hans Fromm
Termin: 12.11.2019
Einführungseminar für Programmierer, IT-Manager und Contentmanager
Das Seminar dauert 2 Tage.
XSL-Praxis
Referent: Bodo Klare
Termin: 10.10.2019
Praktische Übungen für Programmierer und Contentmanager
Das Seminar dauert 5 Tage.

Abbildung 7.11 Liste mit bedingten Hinweisen

7.9.3 Schleifen

Neben Verzweigungen sind Schleifen übliche Konstrukte in Programmiersprachen. Das Angebot von XSLT ist hier etwas eingeschränkt, was damit zusammenhängt, dass zwar Variablen verwendet werden können – wie in [Abschnitt 7.11](#), »Parameter und Variablen«, beschrieben –, diesen Variablen aber immer nur jeweils einmal ein Wert zugewiesen werden kann. Die den Entwicklern vertraute Möglichkeit, den Ablauf einer Schleife über einen Schleifenzähler zu kontrollieren, kommt deshalb in dieser Form nicht vor.

Die Alternative zu dieser Art von Schleifen sind in XSLT die Iterationen, die mit Hilfe der schon genutzten Anweisung `<xsl:for-each>` gestartet werden können. Dabei wird der Umfang der Iteration durch das `select`-Attribut gesteuert, das mittels eines XPath-Ausdrucks eine bestimmte Knotenmenge aus dem Quelldokument auswählt. Auf jeden einzelnen Knoten aus dieser Knotenmenge werden dann nacheinander die als Kinder des Elements aufgeführten Anweisungen angewendet.

Der Prozessor baut dabei eine entsprechende Knotenliste auf und fügt dem aktuellen Kontext die Information über die Größe dieser Liste, also die Anzahl ihrer Mitglieder, bei.

Bei jeder Wiederholung ändert sich der aktuelle Kontextknoten und damit die Position des aktuellen Knotens in der Knotenliste, so dass XPath-Ausdrücke, die von Kindelementen der Anweisung verwendet werden, jedes Mal einen entsprechend geänderten Kontext vorfinden und die Daten über diesen Kontext auswerten kön-

nen. Das ist sehr praktisch, wenn es etwa darum geht, die Zellen einer Tabelle schrittweise mit Daten aus dem Quelldokument zu füllen.

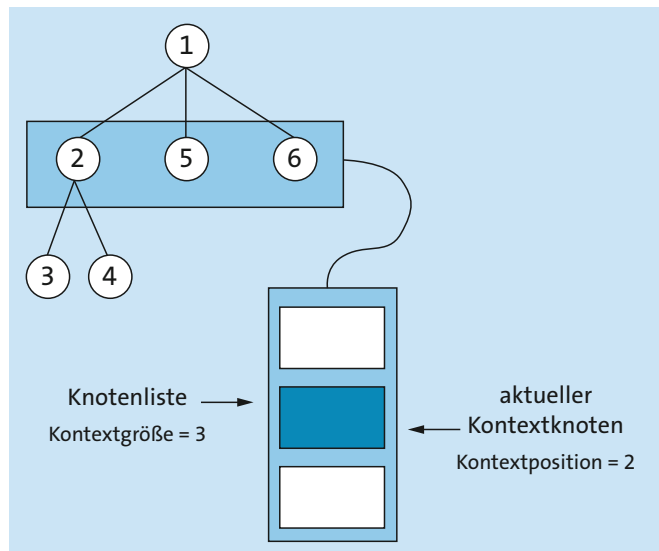


Abbildung 7.12 Knotenliste und Kontextinformationen

Mit dem folgenden Stylesheet wird eine kurze Übersicht in Form einer Tabelle ausgegeben:

```
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="html"/>
  <xsl:template match="/">
    <html>
      <head>
        <title>Kursliste</title>
      </head>
      <body>
        <h3>Kursübersicht</h3>
        <table width="420" border="1" cellspacing="1" cellpadding="20">
          <tr>
            <th>Nr</th>
            <th>Bezeichnung</th>
            <th>Termin</th>
          </tr>
          <xsl:call-template name="kurstabelle"/>
        </table>
      </body>
    </html>
  </template>
</xsl:stylesheet>
```

```

</html>
</xsl:template>
<xsl:template name="kurstabelle">
  <xsl:for-each select="//kurs">
    <tr>
      <td width="30" height="60">
        <xsl:value-of select="position()"/>
      </td>
      <td width="300" height="60">
        <xsl:value-of select="@name"/>
      </td>
      <td width="90" height="60">
        <xsl:value-of select="termin"/>
      </td>
    </tr>
  </xsl:for-each>
</xsl:template>
</xsl:stylesheet>

```

Listing 7.12 kursliste14a.xslt

Die Überschriften der Tabelle werden bereits in der aufrufenden Template-Regel zum Wurzelknoten gesetzt. Innerhalb des `kurstabelle` genannten Templates sind die Anweisungen zum Füllen der einzelnen Tabellenzeilen Abkömmlinge des Elements `<xsl:for-each>`. Da sich bei jedem neuen `<kurs>`-Knoten der Kontext entsprechend der erreichten Position in der Knotenliste ändert, die durch `select="//kurs"` ausgewählt ist, kann mit der Funktion `position()` eine einfache Nummerierung der Kurse erreicht werden, die die aktuelle Position in der Knotenliste wiedergibt.

Kursübersicht		
Nr	Bezeichnung	Termin
1	XML-Grundlagen	12.11.2019
2	XSL-Praxis	10.10.2019
3	XSLT-Einstieg	12.03.2019

Abbildung 7.13 Die Kursübersicht in Form einer Tabelle

7.10 Sortieren und Gruppieren von Quelldaten

Gewöhnlich verarbeitet der XSLT-Prozessor eine durch ein `select`-Attribut ausgewählte Knotenliste in der Reihenfolge, die durch das Quelldokument vorgegeben ist, also in der Dokumentreihenfolge, wie es in [Abschnitt 5.1.3](#) beschrieben ist. Wenn eine andere Anordnung im Ergebnisdokument benötigt wird, kann mit `<xsl:sort>`-Elementen gearbeitet werden. Diese Elemente können als Kindelement von `<xsl:apply-templates>` oder von `<xsl:for-each>` auftreten.

7.10.1 Sortierschlüssel

Wenn Sie mit mehreren Schlüsseln arbeiten wollen, um Duplikate innerhalb des vorgehenden Schlüssels nach einem weiteren Kriterium zu sortieren, nehmen Sie einfach für jeden Schlüssel ein eigenes `<xsl:sort>`-Element. Die Zahl der Schlüssel ist nicht begrenzt. Allerdings muss die Reihenfolge von der größeren zur kleineren Knotenmenge beachtet werden.

```
<xsl:sort select="name"/>
<xsl:sort select="vorname"/>
```

sortiert beispielsweise die üblichen Kandidaten Müller, Meier, Schmitz im zweiten Durchlauf nach den Vornamen.

Innerhalb von `<xsl:for-each>` müssen `<xsl:sort>`-Elemente immer vor den Anweisungen stehen, die auf die sortierten Knoten angewendet werden sollen. Im Zusammenhang mit einer `<xsl:apply-templates>`-Anweisung erscheinen die `<xsl:sort>`-Elemente vor oder auch nach möglichen `<xsl:with-param>`-Elementen.

In dem im letzten Abschnitt verwendeten Stylesheet könnte die Reihenfolge der Tabellenzeilen zum Beispiel nach dem Termin der Kurse aufsteigend geordnet werden. Das Template `kurstabelle` müsste dann folgendermaßen erweitert werden:

```
<xsl:template name="kurstabelle">
  <xsl:for-each select="//kurs">
    <xsl:sort select="concat(substring(termin,7,4),
                          substring(termin,4,2),substring(termin,1,2))"
              order="ascending"
              data-type="text"/>
    <tr>
      <td width="30" height="60">
        <xsl:value-of select="position()"/>
      </td>

      <td width="300" height="60">
```

```

        <xsl:value-of select="@name"/>
      </td>
      <td width="90" height="60">
        <xsl:value-of select="termin"/>
      </td>
    </tr>
  </xsl:for-each>
</xsl:template>

```

Listing 7.13 kursliste14b.xslt

Soll die Sortierung nicht einfach mit dem String-Wert des aktuellen Knotens erfolgen, was die Vorgabe ist, benutzen Sie wie üblich das Attribut `select`, um den Schlüssel festzulegen, nach dem die aktuelle Knotenliste zu sortieren ist. Der angegebene XPath-Ausdruck muss dabei immer einen String-Wert liefern.

Wir haben hier ein Beispiel eines ziemlich komplexen XPath-Ausdrucks, der in diesem Fall notwendig ist, weil die Termine in einem Format angegeben sind, das sich nicht so ohne Weiteres für eine Sortierung eignet. Dabei werden mit Hilfe der Funktion `concat()` die einzelnen Bestandteile des im Element `<termin>` abgelegten Datums, die über die Funktion `substring()` gefunden werden, umgestellt, wobei die Punkte einfach weggelassen werden.

7.10.2 Sortierreihenfolge

Das Attribut `order` bestimmt, ob auf- oder absteigend sortiert wird. `ascending` ist die Vorgabe, hätte hier also auch weggelassen werden können. Auch der Datentyp des Sortierschlüssels kann angegeben werden, wobei `data-type="text"` wiederum Vorgabe ist. Wird `datatype="number"` verwendet, wird der Sortierschlüssel, wo es möglich ist, in einen numerischen Wert konvertiert. Dann reihen sich Werte wie 10, 20, 120 korrekt nach dem Zahlenwert, während sie sonst in der Reihenfolge 10, 120, 20 erscheinen würden.

Wäre das Datum im Format »Jahr-Monat-Tag« – 2019-11-12 – angegeben worden, hätte sich die Sortierung einfach über einen String-Vergleich erledigen lassen. [Abbildung 7.14](#) zeigt die Kurse nach der erfolgten Sortierung.

Wenn Sie [Abbildung 7.14](#) mit [Abbildung 7.13](#) vergleichen, stellen Sie fest, dass die Nummern der Kurse, die jeweils über die Positionsnummer erzeugt wurden, einfach an die neue Reihenfolge nach der Sortierung angepasst wurden. Darin wird noch einmal deutlich, dass der Prozessor die zunächst in der Dokumentreihenfolge vorliegende Knotenliste tatsächlich zunächst dem Sortierschlüssel entsprechend umbaut und erst auf diese Liste die Template-Anweisungen anwendet, die als Kinder des Elements `<xsl:for-each>` vorhanden sind.

Kursübersicht		
Nr	Bezeichnung	Termin
1	XSLT-Einstieg	12.03.2019
2	XSL-Praxis	10.10.2019
3	XML-Grundlagen	12.11.2019

Abbildung 7.14 Die Kursübersicht nach dem Termin aufsteigend sortiert

Die neue Ordnung der Knoten gilt aber immer nur für den Moment; eine andere Template-Regel kann wieder in der üblichen Reihenfolge auf dieselben Knoten zugreifen, wenn es gewünscht wird. Wird mit einem String als Schlüssel gearbeitet, können über die Attribute `lang` und `case-order` länderspezifische Sortierfolgen bestimmt werden. Mit `case-order="upper-first"` werden Großbuchstaben vor Kleinbuchstaben eingeordnet, also »Regen« vor »regen«.

7.11 Parameter und Variablen

Parameter können an eine Template-Regel übergeben werden, wenn sie mit `<xsl:apply-templates>` oder `<xsl:call-template>` aufgerufen wird. Die Parameter werden in beiden Fällen jeweils als Kindelemente mit `<xsl:with-param>`-Anweisungen festgelegt. Der Parameterwert kann dabei entweder über ein `select`-Attribut festgelegt werden, dann bleibt das Element selbst leer, oder der Parameterwert ist der Inhalt des Elements. Als Inhalt enthält das Element selbst ein Template. Das Ergebnis des `select`-Ausdrucks kann ein String, eine Zahl, ein logischer Wert oder eine Knotenliste sein.

7.11.1 Parameterübergabe

Damit das aufgerufene Template die angebotenen Parameter nutzen kann, müssen entsprechende `<xsl:param>`-Anweisungen als erste Kinder des Elements `<xsl:template>` eingefügt werden. Dabei lassen sich Listen von Parametern mit vorgegebenen Werten definieren, die verwendet werden, wenn der Aufruf der Template-Regel ohne entsprechende Parameterwerte erfolgt. Um den Vorgabewert festzulegen, kann wie bei `<xsl:with-param>` wieder entweder ein `select`-Attribut verwendet werden oder

ein Template als Inhalt des Elements. Fehlt zu einem Aufrufparameter ein entsprechendes `<xsl:param>`-Element, führt dies allerdings nicht zu einem Fehler, der Parameter wird bei der Durchführung der Template-Regel einfach ignoriert.

Die Parameter, die an ein Template übergeben worden sind, gehören zum aktuellen Kontext des Templates und lassen sich innerhalb von XPath-Ausdrücken in `select`-Attributen verwenden, wenn ein `$`-Zeichen vor den Namen gesetzt wird.

Ein einfaches Beispiel ist die Übergabe eines bestimmten Format-Tokens an eine Template-Regel für Nummerierung, wie Sie es oben bereits kennengelernt haben:

```
<xsl:template match="kursprogramm">
  <h3>Seminare in <xsl:value-of select="@ort"/>: </h3>
  <xsl:apply-templates select="kurs">
    <xsl:with-param name="form">A.1 </xsl:with-param>
  </xsl:apply-templates>
</xsl:template>

<xsl:template match="kurs">
  <xsl:param name="form" select="I.1" />
  <p><xsl:number level="multiple" count="kursprogramm|kurs"
    format="{form}"/>
    <xsl:value-of select="@name"/></p>
</xsl:template>
```

Die innerhalb des Templates definierte Vorgabe wird in diesem Fall überschrieben, und die Liste wird an der ersten Stelle mit Buchstaben gelistet.

7.11.2 Globale Parameter

Auch an ein Stylesheet insgesamt können Parameter übergeben werden, wenn der verwendete Prozessor dies unterstützt. In diesem Fall werden die Parameter als Top-Level-Elemente, das heißt als Kinder des Elements `<xsl:stylesheet>` definiert. Eine denkbare Anwendung wäre etwa die Auswahl des Modus, wenn für die Ausgabe am Bildschirm oder für den Ausdruck Template-Regeln unterschiedliche Modi definiert sind.

Im Stylesheet kann dies zum Beispiel so aussehen:

```
<xsl:stylesheet ...>
  <xsl:param name="modus" select="'screen'"/>
  ...
  <xsl:template match="/" mode="{modus}">
```

Wenn Sie als XSLT-Prozessor Xalan nutzen, kann ein Parameter für den Modus so übergeben werden:

```
java org.apache.xalan.xslt.Process -in kurs.xml
    -xsl kursparam.xsl -param modus print
```

7.11.3 Lokale und globale Variablen

Anders als in üblichen Programmiersprachen kennt XSLT Variablen nur in einem sehr eingeschränkten Sinn. Sie können zwar eine Variable deklarieren und ihr einen Wert zuweisen, dieser Wert kann aber dann nicht mehr geändert werden. Verwendet wird dafür das Element `<xsl:variable>`, das als Top-Level-Element erscheinen kann, um eine globale Konstante zu definieren, oder lokal innerhalb eines Templates.

Die Konsequenz aus dieser Einschränkung ist, dass ein Template keinen variablen Wert an ein anderes Template übergeben kann. Die einzelnen Template-Regeln haben in diesem Sinne keine Nebeneffekte – eine Eigenschaft, die bei der Konzeption von XSLT ausdrücklich angestrebt wurde, um die Arbeit mit Templates zu vereinfachen. Denn auf diese Weise ist es möglich, die verschiedenen Templates innerhalb eines Stylesheets wahlweise in dieser oder jener Reihenfolge aufzurufen.

Die Syntax von `<xsl:variable>` entspricht ansonsten der von `<xsl:param>`, allerdings können hier keine Default-Werte angegeben werden. Außerdem darf ein `<xsl:variable>`-Element nicht nur als erstes Kindelement innerhalb des Templates erscheinen, sondern überall da, wo es benötigt wird.

Wenn eine Variable innerhalb eines Templates eingefügt wird, muss auf den Geltungsbereich geachtet werden. Der Wert der Variable kann nur von den nachfolgenden Geschwisterelementen und deren Nachkommen verwendet werden, wie [Abbildung 7.15](#) verdeutlichen soll. Das Element 2 stellt hier eine Variable dar, die beispielsweise Kind eines `<xsl:for-each>`-Elements ist. Enthält das Variablenelement selbst Kindelemente, können sie sich nicht auf den Variablenwert beziehen.

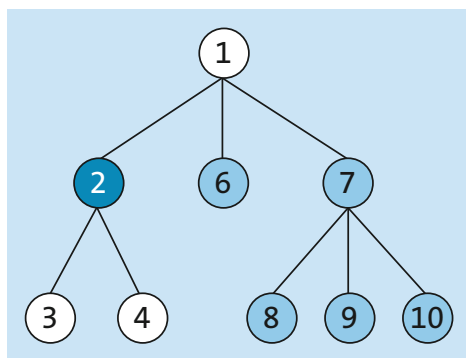


Abbildung 7.15 Nur die Knoten 6–10 können den Wert der Variablen in 2 verwenden.

Die lokale Variable verliert ihre Gültigkeit, sobald das End-Tag des Elternelements der Variablen erreicht ist, also etwa bei `</xsl:for-each>`.

7.11.4 Eindeutige Namen

Variablen- und Parameternamen dürfen in einem Template nur jeweils einmal benutzt werden. Das folgende Template ist nicht erlaubt:

```
<xsl:template name="liste">
  <xsl:param name="startwert"/>
  <xsl:variable name="startwert"/>
  ...
</xsl:template>
```

Allerdings ist es möglich, in einem Template eine globale Definition einer Variablen oder eines Parameters zu überdecken, indem für die lokale Variable oder den lokalen Parameter derselbe Name verwendet wird. Dann kommt die globale Definition innerhalb dieses Templates nicht zum Zuge.

7.11.5 Typische Anwendungen von Variablen in XSLT

Variablen werden gerne eingesetzt, um Wiederholungen gleicher Ausdrücke zu vermeiden und so ein Stylesheet lesbarer und pflegbarer zu machen.

Werte mehrfach verwenden

Ein einfaches Beispiel ist eine globale Variable, die eine bestimmte Ausrichtung definiert:

```
<xsl:template match="kurs">
  <h3><xsl:value-of select="@name"/></h3>
  <p align="{a}">Referent: <xsl:value-of select="referent"/></p>
  <p align="{a}">Termin: <xsl:value-of select="termin"/></p>
  <p align="{a}"><xsl:value-of select="beschreibung"/></p>
</xsl:template>

<xsl:variable name="a" select="'center'"/>
```

Die Variable `a` wird als Top-Level-Element definiert. Es spielt deshalb keine Rolle, ob das `<xsl:variable>`-Element vor oder – wie hier – hinter den Template-Regeln auftaucht, die den Wert der Variablen verwenden. Der String, der der Variablen als Wert zugewiesen wird, ist in einfache Anführungszeichen gesetzt, damit er nicht vom Prozessor als das Element `<center>` missinterpretiert wird. Bei Zahlenwerten kann auf

die einfachen Anführungszeichen verzichtet werden, weil der Prozessor die Regel beachtet, dass Elementnamen nicht mit einer Zahl beginnen dürfen.

Innerhalb der Template-Regel wird der Wert der Variablen mehrfach dafür verwendet, den Wert des Attributs `align` zu setzen. Dabei wird vor den Variablennamen das `$`-Zeichen gesetzt und gleichzeitig der ganze Ausdruck in geschweifte Klammern eingefasst, damit er als Attributwert-Template ausgewertet wird, wie bereits in [Abschnitt 7.1.9](#), »Attributwert-Templates«, beschrieben. Der Nutzen dieses Verfahrens ist natürlich umso größer, je komplexer der Ausdruck ist, der den an die Variable gebundenen Wert liefert.

Ein `<xsl:variable>`-Element kann auch selbst Kindelemente enthalten, etwa um eine Wahl zwischen verschiedenen Alternativen zu treffen. Dann entfällt das `select`-Attribut. In dem folgenden Beispiel wird die Textfarbe der `<h3>`-Überschriften davon abhängig gemacht, welche Dauer der Kurs hat, der gerade bearbeitet wird.

```
<xsl:template match="kurs">
  <xsl:variable name="fc">
    <xsl:choose>
      <xsl:when test="@dauer='2 Tage'">blue</xsl:when>
      <xsl:when test="@dauer='5 Tage'">gray</xsl:when>
      <xsl:otherwise>red</xsl:otherwise>
    </xsl:choose>
  </xsl:variable>
  <h3><font color="{ $fc }">
    <xsl:value-of select="@name" /></font>
  </h3>
  ...
</xsl:template>
```

Listing 7.14 kursliste17.xslt

Die Farbnamen können diesmal ohne Anführungszeichen eingetragen werden, weil sie Inhalt des jeweiligen Elements sind und nicht Teil des `select`-Attributs.

Zwischenspeichern von Kontextinformationen

Eine weitere typische Verwendung ist das Zwischenspeichern von kontextabhängigen Daten, die sonst verloren gingen, weil sich inzwischen der Kontext geändert hat. Das gilt insbesondere für die Verwendung von `<xsl:for-each>`-Iterationen. Um dies zu demonstrieren, erweitern wir unsere Quelldatei um einen Abschnitt, in dem die Namen und Telefonnummern der Referenten aufgelistet sind:

```

<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="kursliste18.xslt"?>
<kursprogramm>
  <kurs name="XML-Grundlagen" dauer="2 Tage">
    <referent>Hans Fromm</referent>
    <termin>12.11.2019</termin>
  </kurs>
  <kurs name="XSL-Praxis" dauer="5 Tage">
    <referent>Bodo Klare</referent>
    <termin>10.10.2019</termin>
  </kurs>
  <kurs name="XSLT-Einstieg" dauer="2 Tage">
    <referent>Hanna Horn</referent>
    <termin>12.03.2019</termin>
  </kurs>
  <kurs name="XML-Schnupperkurs">
    <referent>Hanna Horn</referent>
    <termin>12.03.2019</termin>
  </kurs>
  <referent name="Bodo Klare">
    <fon>766684</fon>
  </referent>
  <referent name="Hanna Horn">
    <fon>676689</fon>
  </referent>
  <referent name="Hans Fromm">
    <fon>476688</fon>
  </referent>
</kursprogramm>

```

Listing 7.15 kursliste18.xml

Mit dem folgenden Stylesheet kann eine Liste erzeugt werden, die für jeden Referenten die Kurse anzeigt, die er halten wird.

```

<?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>
      <body>
        <h2>Die Kurse der Referenten:</h2>
        <xsl:call-template name="kursliste"/>
      </body>
    </html>
  </template>

```

```

        </body>
    </html>
</xsl:template>
<xsl:template name="kursliste">
    <xsl:for-each select="/kursprogramm/referent">
        <h3><xsl:value-of select="@name"/>
            <xsl:text> - </xsl:text>Tel: <xsl:value-of select="fon"/>
        </h3>
        <xsl:variable name="ref" select="."/>

        <xsl:for-each select="/kursprogramm/kurs[referent=$ref/@name]">
            <h4>
                <xsl:value-of select="@name"/>
            </h4>
        </xsl:for-each>

    </xsl:for-each>
</xsl:template>
</xsl:stylesheet>

```

Listing 7.16 kursliste18.xslt

Das Stylesheet arbeitet mit zwei verschachtelten `<xsl:for-each>`-Schleifen. Die äußere Schleife arbeitet die Elemente `<referent>` ab. In der Variablen `ref` wird der Name des Referenten, der gerade bearbeitet wird, mit `select="."` gespeichert. Diesen Wert benutzt dann die innere Schleife, die die Knotenliste mit den Kursen bearbeitet, um jeweils die Kurse aufzusuchen, bei denen der Name des Referenten mit dem in der Variablen enthaltenen Namen übereinstimmt.

Die Kurse der Referenten:

Bodo Klare - Tel: 766684

XSL-Praxis

Hanna Horn - Tel: 676689

XSLT-Einstieg

XML-Schnupperkurs

Hans Fromm - Tel: 476688

XML-Grundlagen

Abbildung 7.16 Die Ausgabe, die die Doppelschleife liefert

Temporäre Knotenbäume

Schließlich können Variablen auch benutzt werden, um temporäre Knotenbäume verfügbar zu halten. Dieser spezielle Datentyp, den das Element `<xsl:variable>` zusätzlich zu den bei XPath-Ausdrücken sonst möglichen Datentypen einführt, wird auch *Result Tree Fragment* genannt. Dieser Datentyp repräsentiert einen Teil des Ergebnisbaums und wird wie eine Knotenliste behandelt, die nur aus einem Knoten besteht. Die Operationen, die auf einen solchen temporären Knotenbaum angewendet werden dürfen, sind eingeschränkt auf solche, die auch auf String-Werte angewendet werden können.

In der folgenden Template-Regel für den Wurzelknoten wird eine Variable `liste` definiert, der das Ergebnis eines mit `<xsl:call-template>` aufgerufenen Templates zugewiesen wird. An dieses Template wird als Parameter ein XPath-Ausdruck übergeben, der die Namen der Referenten liefern soll.

Das aufgerufene Template benutzt diesen Parameter-Ausdruck im `select`-Attribut einer `<xsl:for-each>`-Schleife, die die Namen zusammensucht und daraus den temporären Knotenbaum aufbaut. Über zwei `<xsl:if>`-Elemente wird anhand der Position des aktuellen Elements in der Knotenliste hinter den Namen noch ein trennendes Komma oder ein »und« gesetzt. Beachten Sie hier, dass der logische Operator `<` in dem ersten Testausdruck maskiert werden muss.

Das gesamte Ergebnisbaumfragment wird an das aufrufende Template zurückgegeben und mit Hilfe eines `<xsl:value-of>`-Elements in den Ausgabebaum geschrieben. Um zu demonstrieren, dass auf diesen besonderen Datentyp tatsächlich String-Funktionen angewendet werden können, wird die `normalize-space()`-Funktion verwendet, die überflüssige Leerzeichen zwischen den Namen entfernt.

```
<xsl:template match="/">
  <xsl:variable name="liste">
    <xsl:call-template name="referentenliste">
      <xsl:with-param name="namen"
        select="/kursprogramm/referent/@name"/>
    </xsl:call-template>
  </xsl:variable>
  <h3>Unsere Referenten:</h3>
  <p>
    <xsl:value-of select="$liste"/>
  </p>
</xsl:template>

<xsl:template name="referentenliste">
  <xsl:param name="namen"/>
  <xsl:for-each select="$namen">
```

```

    <xsl:value-of select="."/>
    <xsl:if test="position()&lt;last()-1">, </xsl:if>
    <xsl:if test="position()=last()-1"> und </xsl:if>
  </xsl:for-each>
</xsl:template>

```

Listing 7.17 kursliste19.xslt

Das Stylesheet liefert als Ergebnis die Zeilen aus [Abbildung 7.17](#).

Unsere Referenten:

Bodo Klare, Hanna Horn und Hans Fromm

Abbildung 7.17 Ausgabe, generiert aus einem temporären Knotenbaum

7.11.6 Rekursive Templates

Obwohl weder die Werte von Variablen noch die von Parametern geändert werden können, wenn sie einmal zugewiesen sind, gibt es doch Möglichkeiten, Effekte zu erreichen, wie sie in der prozeduralen Programmierung mit »richtigen« Variablen üblich sind. Dabei kann ausgenutzt werden, dass in Form der Iteration Parameter an Templates übergeben werden können, deren Definition zwar fixiert ist, deren Wert sich aber bei jedem Schleifendurchlauf ändert. Das erlaubt rekursive Templates, die sich selbst aufrufen. Um dies zu demonstrieren, wollen wir ein Verfahren, das Michael Kay in seinem Buch »XSLT Programmer's Reference« (erschienen bei Wrox, 2001) vorgestellt hat, auf unser Kursbeispiel anwenden. Die Quelldaten sind diesmal ein Protokoll der Kursbesuche:

```

<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="kursbesuch.xslt"?>
<kursbesuch>
  <kurs name="XML-Grundlagen" dauer="2 Tage">
    <teilnehmer>20</teilnehmer>
    <gebuehr>400</gebuehr>
  </kurs>

  <kurs name="XSL-Praxis" dauer="5 Tage">
    <teilnehmer>10</teilnehmer>
    <gebuehr>800</gebuehr>
  </kurs>

  <kurs name="XSLT-Einstieg" dauer="2 Tage">
    <teilnehmer>30</teilnehmer>
  </kurs>

```

```

    <gebuehr>400</gebuehr>
  </kurs>
</kursbesuch>

```

Listing 7.18 kursbesuch.xml

Das Stylesheet soll für jeden Kurs die Gebühr mit der Zahl der Teilnehmer multiplizieren und die Gesamtsumme der Kursgebühren errechnen und ausgeben. Das Stylesheet kann so aussehen:

```

<?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:variable name="gesamtsumme">
      <xsl:call-template name="gebuehrensumme">
        <xsl:with-param name="liste" select="//kurs"/>
      </xsl:call-template>
    </xsl:variable>
    Summe der Kursgebühren: <xsl:value-of select="formatnumber
      ($gesamtsumme, 'i #.00')"/>
  </xsl:template>
  <xsl:template name="gebuehrensumme">
    <xsl:param name="liste"/>
    <xsl:choose>
      <xsl:when test="$liste">
        <xsl:variable name="erster_kurs" select="$liste[1]"/>
        <xsl:variable name="summe_rest">
          <xsl:call-template name="gebuehrensumme">
            <xsl:with-param name="liste"
              select="$liste[position()>1]"/>
          </xsl:call-template>
        </xsl:variable>
        <xsl:value-of select="$erster_kurs/teilnehmer *
          $erster_kurs/gebuehr + $summe_rest"/>
      </xsl:when>

      <xsl:otherwise>0</xsl:otherwise>
    </xsl:choose>
  </xsl:template>
</xsl:stylesheet>

```

Listing 7.19 kursbesuch.xslt

Dieses Stylesheet liefert die Ausgabe:

Summe der Kursgebühren: € 28000.00

Die Template-Regel für den Wurzelknoten definiert eine Variable `gesamtsumme`, deren Wert durch das Template `gebuehrensumme` geliefert werden soll. Als Parameter wird die Liste der Kurselemente übergeben. Dieses Template soll die Liste der Kurselemente abarbeiten, bis sie leer ist. Das Template multipliziert für das erste Kurselement die Zahl der Teilnehmer mit der Kursgebühr und ruft sich dann mit der jedes Mal um ein Element verkürzten Liste so lange selbst auf, bis der letzte Knoten verarbeitet ist.

Das aufrufende Template gibt den errechneten Gesamtbetrag schließlich mit `<xsl:value-of>` aus, nachdem der gelieferte String mit der Funktion `format-number()` in eine formatierte Zahl umgewandelt worden ist.

7.12 Hinzufügen von Elementen und Attributen

Über Stylesheets lassen sich nicht nur Elemente und Attribute aus dem Quelldokument auswerten, sondern bei Bedarf auch neue Elemente und Attribute hinzufügen, falls in der Quelldatei bestimmte Informationen fehlen. XSLT bietet dafür folgende Elemente:

- ▶ `<xsl:element>`
- ▶ `<xsl:attribut>`
- ▶ `<xsl:comment>`
- ▶ `<xsl:processing-instruction>`
- ▶ `<xsl:text>`

In der Regel ist es nicht notwendig, `<xsl:element>` zu verwenden, um Elemente in das Ergebnisdokument zu schreiben. Sie können einfach mit literalen Ergebniselementen arbeiten, wie es in den bisherigen Beispielen bereits geschehen ist. Die Anweisung ist aber nützlich, wenn es darum geht, Informationen, die im Quelldokument in Form eines Attributs vorliegen, in der Ausgabe in ein eigenes Element zu übernehmen. Auch neue Attribute können nachträglich in Elemente für die Ausgabe eingefügt werden.

7.12.1 Elemente aus vorhandenen Informationen erzeugen

Mit dem folgenden Stylesheet wird aus unserem Kursprogramm eine neue XML-Datei erzeugt, die die beiden Attribute des Elements `<kurs>` in Kindelemente umwandelt, wobei der Elementname vom Attributnamen übernommen und der Elementinhalt vom Wert des jeweiligen Attributs genommen wird.


```

<?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"/>

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

  <xsl:template match="kurs">
    <kurs>

      <xsl:for-each select="@*">
        <xsl:element name="{name()}">
          <xsl:value-of select="."/>
        </xsl:element>
      </xsl:for-each>
    </kurs>
  </xsl:template>

</xsl:stylesheet>

```

Listing 7.20 kursliste20.xslt

Die `<xsl:for-each>`-Schleife findet in unserem Beispiel die beiden Attribute des Elements `<kurs>` – `name` und `dauer` – und übernimmt jeweils den Attributnamen als neuen Elementnamen und den Attributwert als Inhalt der neuen Elemente. Das Stylesheet liefert ein neues XML-Dokument:

```

<?xml version="1.0" encoding="UTF-16"?>
<kursprogramm>
  <kurs>
    <name>XML-Grundlagen</name>
    <dauer>2 Tage</dauer>
  </kurs>
  <kurs>
    <name>XSL-Praxis</name>
    <dauer>5 Tage</dauer>
  </kurs>
  <kurs>
    <name>XSLT-Einstieg</name>

```

```

    <dauer>2 Tage</dauer>
  </kurs>
</kursprogramm>

```

7.12.2 Attributlisten

Das Element `<xsl:attribut>` kann auch in dem Top-Level-Element `<xsl:attribute-set>` auftauchen, das meistens verwendet wird, um mehrfach benötigte Attribute zu definieren. Eine typische Verwendung sind Attributblöcke mit Angaben zur Formatierung von Tabellen, `<div>`-Elementen etc.

```

<xsl:attribute-set name="Tabellenformat">
  <xsl:attribut name="rand">2</xsl:attribut>
  <xsl:attribut name="fuellung">3</xsl:attribut>
  <xsl:attribut name="breite">400</xsl:attribut>
</xsl:attribute-set>

```

Ein solcher Satz kann in einem Template in folgender Form übernommen werden:

```

<table xsl:use-attribute-sets="Tabellenformat">

```

7.12.3 Texte und Leerräume

Anstatt Texte direkt in ein Template zu schreiben, wie es in einigen Beispielen bereits vorgekommen ist, kann auch das Element `<xsl:text>` verwendet werden. Es ist insbesondere nützlich, um Leerräume in das Ergebnisdokument einzufügen, die sonst »gestrippt« würden. Vergleichen Sie folgende Zeilen. Das Leerzeichen in der ersten Zeile würde in der Ausgabe nicht mehr auftauchen, das in der zweiten Zeile bliebe dagegen erhalten.

```

<p><value-of select="name"> <value-of select="vorname"></p>
<p><value-of select="name"><xsl:text> </xsl:text>
  <value-of select="vorname"></p>

```

7.12.4 Kontrolle der Ausgabe

Wenn Sie dem Prozessor die Art der Ausgabe nicht einfach überlassen, können Sie – wie schon angesprochen – das Element `<xsl:output>` für entsprechende Anweisungen verwenden, wie der bei der Abarbeitung des Stylesheets aufgebaute Ergebnisbaum schließlich in das Ergebnisdokument übertragen werden soll. (Der Ergebnisbaum kann auch direkt an eine andere Anwendung weitergereicht werden, etwa vermittelt über die DOM-Schnittstelle, dann darf die `<xsl:output>`-Anweisung auch ignoriert werden.)

Wird `xml` als Methode angegeben oder vom Prozessor als Vorgabe benutzt, beginnt das Ergebnisdokument auf jeden Fall mit der XML-Deklaration. Entitätsreferenzen werden im Ergebnisdokument nicht durch Zeichen wie `<` oder `&` ersetzt.

Die Methode `html` erzeugt Standard-HTML, die XML-Deklaration des Quelldokuments wird deshalb auch nicht in die Ausgabe übernommen. Das Fehlen von End-Tags für Elemente, bei denen dies in HTML möglich ist, wird zugelassen.

Bei der Methode `text` wird der vom Prozessor aufgebaute Ausgabebaum ohne weitere Änderung als einfaches Textdokument ausgegeben.

Abhängig von der Ausgabemethode, die Sie in dem Element `<xsl:output>` angeben, stehen weitere Attribute zur Verfügung.

So lässt sich über `encoding` der Zeichensatz festlegen, der für die Ausgabe verwendet werden soll. Der gewählte Wert wird bei einem XML-Dokument in die XML-Deklaration übernommen, bei HTML in ein entsprechendes `<META>`-Element, etwa `<META http-equiv="Content-Type" content="text/html; charset=UTF-8">`.

Über das Attribut `media-type` kann der MIME-Typ des Ergebnisdokuments vorgegeben werden, etwa:

```
<xsl:output media-type="text/xml"/>
```

Mit Hilfe der Attribute `doctype-system` und `doctype-public` lassen sich auch DTDs zuordnen, so dass in der Ausgabe eine entsprechende Dokumenttyp-Deklaration mit einem lokalen oder öffentlichen Identifier eingefügt wird.

Mit `indent` erreichen Sie, dass die Elemente zur besseren Lesbarkeit eingerückt werden.

7.13 Zusätzliche XSLT-Funktionen

In den XPath-Ausdrücken, die XSLT an verschiedenen Stellen verwendet, können die in [Kapitel 5](#), »Navigation und Verknüpfung«, zu XPath beschriebenen Funktionen benutzt werden. XSLT stellt aber noch einige zusätzliche Funktionen zur Verfügung, auf die an dieser Stelle kurz eingegangen werden soll.

7.13.1 Zugriff auf mehrere Quelldokumente

Einige der häufig eingesetzten Funktionen wollen wir an einem Beispiel vorstellen, das Ihnen zeigt, wie mit XSLT auf mehrere Dokumente gleichzeitig zugegriffen werden kann. Wir bleiben bei unseren Kursen und nehmen uns jetzt aber vor, die Anmeldungen zu den Kursen in Form kleiner XML-Dokumente zu erfassen und dann über ein zusammenfassendes Dokument mit Hilfe eines Stylesheets, das diesem zugeordnet wird, auszuwerten.

Die Lösung ist hier einfach gehalten, um den Vorgang übersichtlich darstellen zu können. Es ist aber ohne Weiteres möglich, noch wesentlich komplexere Elemente einzubauen, etwa Berechnungen, wie sie bereits in dem oben genannten Beispiel gezeigt wurden. Für jede Kursanmeldung soll eine eigene Datei verwendet werden, die in einer Instanz so aussieht:

```
<?xml version="1.0" encoding="UTF-8"?>
<kursanmeldung>
  <kurs name="XML-Grundlagen"/>
  <teilnehmer>
    <name>Gerd Wehn</name>
    <adresse>50678 Köln, Rolandstr. 10</adresse>
  </teilnehmer>
  <gebuehr>400</gebuehr>
</kursanmeldung>
```

Listing 7.21 kursanmeldung001.xml

Stellen Sie sich eine ganze Serie solcher Dateien vor, die mit unterschiedlichen Endziffern im Dateinamen arbeiten. Ein XML-Dokument, das diese kleinen Dokumente zusammenfasst, kann dann so aussehen:

```
<?xml version="1.0" encoding="UTF-8"?>
<kursanmeldungen>
  <beleg dateiname="kursanmeldung_001.xml"/>
  <beleg dateiname="kursanmeldung_002.xml"/>
  <beleg dateiname="kursanmeldung_003.xml"/>
</kursanmeldungen>
```

Listing 7.22 kursanmeldunggesamt.xml

Dieses Hauptdokument lässt sich nun mit einem Stylesheet verarbeiten, das sich so darstellt:

```
<?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" />
  <xsl:decimal-format name="euro" decimal-separator=","
    grouping-separator="."/>

  <xsl:template match="/">
    <html>
      <head>
        <title>Kursanmeldungen</title>
      </head>
```

```

<body>
  <h2>Kursanmeldungen:</h2>
  <xsl:for-each select="/kursanmeldungen/beleg">
    <xsl:apply-templates select="document(@dateiname)/kursanmeldung"/>
  </xsl:for-each>
</body>
</html>
</xsl:template>
<xsl:template match="kursanmeldung">
  <h4><xsl:value-of select="//kurs/@name"/></h4>
  <p>Name: <xsl:value-of select="//teilnehmer/name"/> </p>
  <p>Adresse: <xsl:value-of select="//teilnehmer/adresse"/> </p>
  <p>Betrag: <xsl:value-of select="format-number(//gebuehr,
    '##.###,00 &#8364;', 'euro')"/> </p>
</xsl:template>
</xsl:stylesheet>

```

Listing 7.23 kursanmeldung.xslt

Das Stylesheet wertet das Hauptdokument aus, indem es in einer `<xsl:for-each>`-Schleife jeden Beleg abarbeitet, der im Hauptdokument aufgeführt wird. Dabei wird jedes Mal mit Hilfe der `document()`-Funktion in die Dokumente mit den einzelnen Anmeldungen hineingesehen. Das erste Argument der Funktion liefert den Namen der Datei, das zweite liefert gleich eine Knotenliste innerhalb dieser Datei.

Die Ausgabe der so gewonnenen Daten wird über die zweite Template-Regel bestimmt. Die XPath-Ausdrücke für die `select`-Attribute werden nun schon im Kontext der Dateien mit den Einzelbelegen ausgewertet.

7.13.2 Zahlenformatierung

Um die Euro-Beträge korrekt darzustellen, ist am Anfang des Stylesheets mit `<xsl:decimal-format>` ein Zahlenformat definiert worden, das das passende Dezimaltrennzeichen und das Zeichen für die Zifferngruppierung gewährleistet. Dieses Format wird innerhalb des Templates mit Hilfe der Funktion `format-number()` aufgerufen, wobei das erste Argument den Betrag angibt, das zweite Argument ein Formatierungsmuster und das dritte Argument den Namen des oben definierten Formats. Im Formatierungsmuster wird das Doppelkreuz verwendet, um führende Nullen zu kennzeichnen, die nicht angezeigt werden sollen. Das Euro-Zeichen wird mit Hilfe einer dezimalen Zeichenreferenz angegeben, da das Stylesheet mit `encoding="UTF-8"` arbeitet. [Abbildung 7.18](#) zeigt das Ergebnis in Mozilla Firefox.

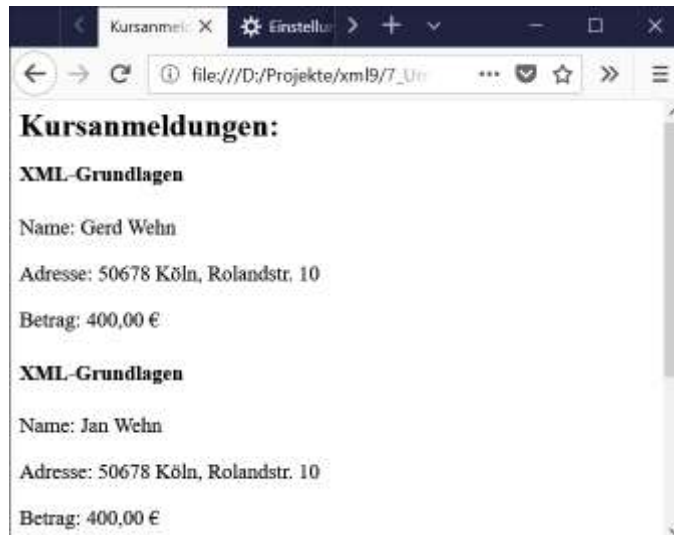


Abbildung 7.18 Ausgabe der zusammengefassten Dokumente

7.13.3 Liste der zusätzlichen Funktionen in XSLT

Funktion	Bedeutung
<code>current()</code>	Liefert den aktuellen Knoten. Entspricht dem XPath-Ausdruck ".", außer innerhalb von Prädikaten, wenn der aktuelle Knoten nicht mit dem Kontextknoten identisch ist. Kann nicht in Mustern verwendet werden.
<code>document()</code>	Erlaubt es, von einem Stylesheet aus auf externe XML-Quelldateien zuzugreifen. Wird typischerweise bei <code>href</code> -Attributen verwendet, um über einen URI Dateien auszuwerten.
<code>element-available()</code>	Prüft, ob ein Element für den verwendeten XSLT-Prozessor verfügbar ist. Damit können insbesondere Fehler vermieden werden, wenn Erweiterungselemente benutzt werden, die dem Prozessor nicht bekannt sind.
<code>format-number()</code>	Wandelt einen String in eine Zahl mit dem angegebenen Format um.
<code>function-available()</code>	Prüft, ob eine Funktion für den verwendeten XSLT-Prozessor verfügbar ist. Das gilt insbesondere für Erweiterungsfunktionen.

Tabelle 7.5 Zusatzfunktionen in XSLT 1.0

Funktion	Bedeutung
<code>generate-id()</code>	Erzeugt einen String in Form eines XML-Namens, der Knoten eindeutig identifiziert. Wie der String aussieht, hängt allerdings vom verwendeten Prozessor ab.
<code>id()</code>	Liefert eine Knotenliste mit einem bestimmten ID-Attribut.
<code>key()</code>	Wird verwendet, um einen Knoten zu finden, dem mit <code><xsl:key></code> ein bestimmter Schlüssel zugewiesen worden ist.
<code>lang()</code>	Prüft, ob die Sprache des Kontextknotens, die durch das Attribut <code>xml:lang</code> gesetzt werden kann, der als Argument angegebenen Sprache entspricht.
<code>system-property()</code>	Liefert Informationen über die Systemumgebung, in der das Stylesheet ausgeführt wird, insbesondere den XSLT-Prozessor.
<code>unparsed-entity-uri()</code>	Erlaubt den Zugriff auf die Deklarationen von ungeparsten Entitäten in der DTD des Quelldokuments.

Tabelle 7.5 Zusatzfunktionen in XSLT 1.0 (Forts.)

7.14 Mehrfache Verwendung von Stylesheets

Ein Stylesheet kann andere Stylesheets in sich einschließen und sie einen entsprechenden Teil der Transformationsarbeit erledigen lassen. Es kann also aus mehrfach verwendbaren Bausteinen zusammengefügt werden.

Stylesheets werden häufig für bestimmte Fragmente von Dokumenten entwickelt, die immer wieder benötigt werden. Werden XML-Daten in HTML-Seiten exportiert, soll vielleicht ein bestimmter Seitenkopf verwendet werden mit allgemeinen Angaben über eine Firma oder Organisation. Dieselbe Ausgabeform wird auch für solche Daten benötigt, die tabellarisch ausgegeben werden sollen. Ein Stylesheet kann zum Beispiel eingesetzt werden, um die Fonts zusammenzustellen, die innerhalb einer Website vorkommen, oder auch für die Farben, die für bestimmte Seitenbereiche vorgesehen sind.

Ähnlich wie es für XML-Schemas beschrieben wurde, gibt es hier zwei Verfahren, Stylesheet-Fragmente von außen zu integrieren: `<xsl:import>` und `<xsl:include>`. Beides sind Top-Level-Elemente, dürfen also nur als Kindelemente von `<xsl:stylesheet>` verwendet werden, wobei `<xsl:import>`-Elemente sogar vor allen anderen Top-Level-Elementen angeführt werden müssen. In beiden Fällen wird der Prozessor angewiesen, die Anweisung selbst durch den Inhalt des Stylesheets zu ersetzen, das im `href`-Attribut angegeben wird. Dabei werden alle Top-Level-Elemente des externen Style-

sheets übernommen, nur das `<xsl:stylesheet>`-Element wird vernachlässigt, weil das aufnehmende Stylesheet ja bereits über dieses Element verfügt.

7.14.1 Stylesheets einfügen

Das einfachere Verfahren ist das Inkudieren. Dabei wird nur der Text des externen Stylesheets eingefügt, und die eingefügten Elemente und Templates werden so behandelt, als ob sie schon immer zum Stylesheet gehört hätten. Wenn ein Stylesheet beispielsweise drei Fonts zur allgemeinen Verwendung definiert, kann dies so aussehen:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:variable name="Textfont" select="'Times'"/>
  <xsl:variable name="Titelfont" select="'Arial'"/>
  <xsl:variable name="Tabfont" select="'Geneva'"/>
</xsl:stylesheet>
```

Ein Stylesheet kann die Variablendefinitionen mit folgender Anweisung inkludieren:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:include href="Standardfonts.xslt"/>
  ...
```

7.14.2 Stylesheets importieren

Beim Importieren dagegen unterscheidet der Prozessor sowohl bei den Templates als auch bei eventuellen globalen Variablen zwischen solchen, die ursprünglich zu dem importierenden Stylesheet gehören, und solchen, die importiert worden sind. Den importierten Objekten wird ein geringerer Rang – *Importpräzedenz* ist der Ausdruck dafür – zugewiesen. Daraus ergibt sich, dass der Import hauptsächlich dann sinnvoll ist, wenn allgemeine Vorgaben importiert werden, die im speziellen Fall aber überschrieben oder außer Kraft gesetzt werden sollen. In dem folgenden Auszug eines importierenden Stylesheets werden die Font-Variablen zwar ebenfalls übernommen, aber eine dieser Variablen wird durch eine globale Variablendefinition überschrieben, die denselben Namen verwendet.

```
<?xml version="1.0" encoding=" UTF-8"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:import href="Standardfonts.xslt"/>
```



```
<xsl:variable name="Tabfont" select="'Helvetica'"/>
...
</xsl:stylesheet>
```

Innerhalb von inkludierten oder importierten Stylesheets können selbst wieder Inklusionen oder Importe vorkommen, die dann unter Beachtung der Importpräferenz behandelt werden. Wenn A das Stylesheet B importiert, das wiederum das Stylesheet C importiert, haben die aus C übernommenen Templates und Variablen den geringsten Rang.

7.15 Übersetzungen zwischen XML-Vokabularen

Die Freiheiten, die XML bei der Modellierung von Daten- und Dokumentstrukturen gewährt, haben dazu geführt, dass ständig neue Vokabulare für ganz unterschiedliche Anwendungsbereiche entstehen. Gleichzeitig werden große Anstrengungen unternommen, diesen Vokabularen Gewicht und Verbindlichkeit zu verschaffen, um beispielsweise wenigstens den Datenaustausch zwischen Firmen einer Branche zu vereinfachen, etwa nach dem Muster: Alle Apotheker sprechen ApoXML.

Es wäre aber ganz unrealistisch, zu glauben, dass sich etwa alle Teilnehmer an E-Commerce-Aktivitäten über kurz oder lang auf eine einzige Sprache einigen werden, um ihre Geschäftsprozesse abzuwickeln. Es ist auch sehr die Frage, ob das so wünschenswert wäre. Daraus folgt zwangsläufig, dass es, selbst wenn XML als Metasprache weltweit akzeptiert ist, immer nebeneinanderlaufende Vokabulare für sich überschneidende Anwendungsbereiche geben wird. Daraus folgt die Notwendigkeit der Übersetzung, wie sie sich ja auch aus der Vielfalt der natürlichen Sprachen ergibt und zur Bereicherung des »Projekts Mensch« beiträgt.

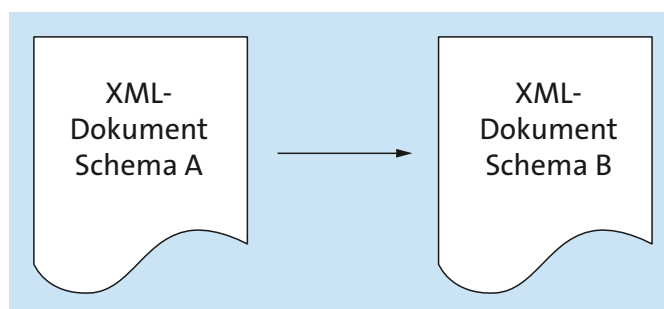


Abbildung 7.19 Übersetzung zwischen XML-Dokumenten

Erfreulicherweise ist mit XSLT bereits ein Spezialist für solche Übersetzungsleistungen bei der Arbeit. Als Übersetzer fungiert in diesem Fall ein XSLT-Stylesheet. (Die Bezeichnung »Stylesheet« ist allerdings hier etwas ungenau, da XSLT ja tatsächlich

nur dazu benutzt wird, ein XML-Vokabular in ein anderes zu übersetzen, womit noch nichts über die Formatierung der Daten gesagt sein muss.)

7.15.1 Diverse Schemas für gleiche Informationen

Zunächst ein kleines Beispiel. Ein Kursangebot könnte so aussehen:

```
<?xml version="1.0" encoding="UTF-8"?>
<kursprogramm>
  <kurs>
    <bezeichnung>XML-Grundlagen</bezeichnung>
    <dozent>Hans Fromm</dozent>
    <termin>12.11.2019</termin>
  </kurs>
  <kurs>
    <bezeichnung>XSL-Praxis</bezeichnung>
    <dozent>Bodo Klare</dozent>
    <termin>10.10.2019</termin>
  </kurs>
  <kurs>
    <bezeichnung>XSLT-Einstieg</bezeichnung>
    <dozent>Hanna Horn</dozent>
    <termin>12.03.2019</termin>
  </kurs>
</kursprogramm>
```

Listing 7.24 kurs_element_version.xml

Das XML-Dokument verwendet eine implizite Struktur, die ausschließlich mit verschachtelten Elementen arbeitet. Was ist nun aber, wenn versucht wird, ein Kursprogramm zusammenzustellen, das Kurse unterschiedlicher Anbieter auflisten soll? Es ist leicht vorstellbar, dass ein anderes Schulungsinstitut die gleichen Informationen in einer anderen Struktur aufbereitet, zum Beispiel unter Verwendung von Attributen anstelle von Elementen. Hier eine mögliche Variante:

```
<?xml version="1.0" encoding="UTF-8"?>
<kursprogramm>
  <kurs name="HTML-Grundlagen" referent="Klaus Hartung" termin="12.04.2019"/>
  <kurs name="HTML-Praxis" referent="Bernd Ette" termin="12.06.2019"/>
  <kurs name="DHTML-Einstieg" referent="Hans Horn" termin="20.10.2019"/>
</kursprogramm>
```

Listing 7.25 kurs_attribut_version.xml

Für einen menschlichen Betrachter ist ziemlich leicht zu erkennen, dass die beiden Varianten für jeden Kurs dieselbe Informationsmenge bereitstellen, obwohl teilweise andere Namen für die einzelnen Informationspartikel verwendet werden. Das Element `<dozent>` in Variante A entspricht dem Attribut `referent` in Variante B, das Element `<bezeichnung>` entspricht dem Attribut `name`. Ein Programm dagegen kann diese Zuordnungen nicht erkennen, es sei denn, es erhält entsprechende Instruktionen.

7.15.2 Angleichung durch Transformation

Soll die zweite Variante als Vorbereitung für eine einheitliche Darstellung in einem bestimmten Medium in die Struktur der ersten Variante umgewandelt werden, kann ein XSLT-Stylesheet entworfen werden, das die dafür erforderlichen Transformationen beschreibt und mit dessen Hilfe ein XSLT-Prozessor ein Ergebnisdokument erzeugt, das der Variante entspricht, die nur mit Elementen arbeitet.

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">
  <xsl:output method="xml" encoding="UTF-8" indent="yes"/>
  <xsl:template match="/">
    <kursprogramm>
      <xsl:for-each select="kursprogramm/kurs">
        <kurs>
          <bezeichnung>
            <xsl:value-of select="@name"/>
          </bezeichnung>

          <dozent>
            <xsl:value-of select="@referent"/>
          </dozent>

          <termin>
            <xsl:value-of select="@termin"/>
          </termin>
        </kurs>
      </xsl:for-each>
    </kursprogramm>
  </xsl:template>
</xsl:stylesheet>
```

Listing 7.26 kurs_transformation.xslt

Das Stylesheet verwendet eine einzige Template-Regel, die auf den Wurzelknoten des Dokuments angewendet wird. Dieses Element wird mit `match="/"` aufgespürt. Der Ergebnisbaum wird dann aus den einzelnen Elementen des Kursprogramms aufgebaut. Für jedes Kurselement wird mit der `xsl:for-each`-Anweisung die gewünschte Umwandlung von Attributen in Elemente und die teilweise damit verbundene Umbenennung vorgenommen. Dies geschieht einfach dadurch, dass dem jeweiligen Element der Wert des entsprechenden Attributs zugewiesen wird. Das Element `<dozent>` erhält beispielsweise den Wert, der mit der Anweisung `xsl:value-of` von dem Attribut `referent` abgefragt wird:

```
<dozent><xsl:value-of select="@referent"/></dozent>
```

Um zu erreichen, dass die Attributvariante in die Elementvariante umgewandelt wird, muss in der Instanz mit der Attributvariante eine Verknüpfung mit dem XSLT-Stylesheet eingefügt werden, und zwar in etwa so:

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

7.16 Umwandlung von XML in HTML und XHTML

Solange HTML das Web dominiert, wird die Umwandlung von XML-Daten in HTML sicher zu den vorrangigen Aufgaben gehören, die sich in diesem Bereich stellen. In den meisten Beispielen dieser Einführung ist das Ergebnisdokument eine HTML-Seite gewesen.

Grundsätzlich sind dabei zwei Verfahren möglich: Die Transformation kann entweder schon auf dem Server stattfinden oder auf dem Client. Für die Bearbeitung auf dem Server werden beispielsweise Java-Servlets wie *Apache Cocoon* eingesetzt. Mehr Details dazu finden Sie unter cocoon.apache.org.

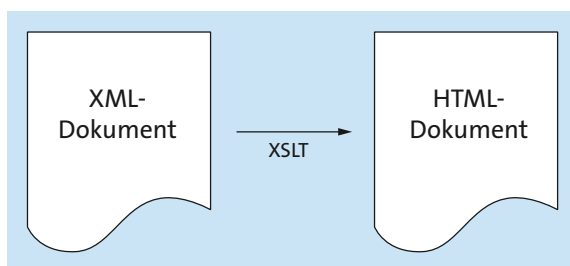


Abbildung 7.20 XSLT kann aus XML wieder HTML erzeugen.

Für die Transformation auf dem Client wird ein Browser benötigt, der XSLT beherrscht. Das ist bei den aktuellen Browserversionen in der Regel der Fall.

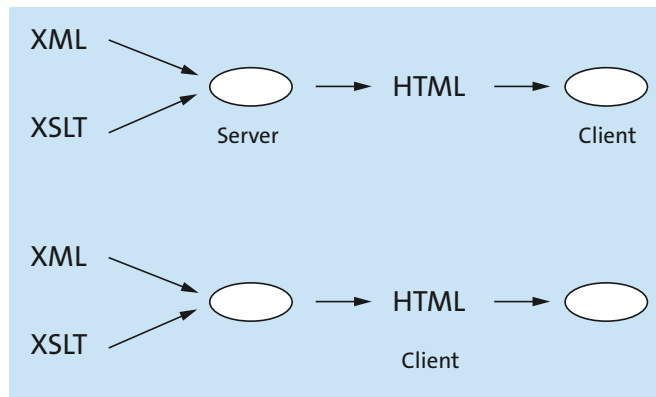


Abbildung 7.21 Die Alternativen bei der Transformation in HTML

7.16.1 Datenübernahme mit Ergänzungen

Wir kommen hier noch einmal auf das kleine Lagerbeispiel aus dem [Kapitel 3](#), »Dokumenttypen und Validierung« zurück. Um eine Liste des Lagerbestands beispielsweise für eine Intranet-Anwendung zu erstellen, kann folgendes Stylesheet verwendet werden. Das Stylesheet wertet nicht nur die Quelldaten aus, sondern fügt auch passende Beschriftungen für die Liste hinzu:

```

<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="html" encoding="UTF-8"/>
  <xsl:template match="/">
    <html>
      <head>
        <title>Lagerliste</title>
        <style type="text/css" media="screen">

          table {
            background-color: silver;
            margin: 1em;
            padding: 1em;
            border: 1px black }
          caption {
            font-size: large;
            background-color: silver;
            border-bottom: 3px black }

        </style>
      </head>
    </html>
  </template>
</xsl:stylesheet>
  
```

```

</head>
<body>
  <table width="180" border="1" cellspacing="15"
        cellpadding="10">
    <caption>Aktueller Lagerbestand:</caption>
    <xsl:call-template name="tabellenkopf"/>
    <xsl:call-template name="tabellenkoerper"/>
  </table>
</body>
</html>
</xsl:template>

<xsl:template name="tabellenkopf">
  <tr>
    <xsl:for-each select="//Artikel[1]/*">
      <td>
        <xsl:value-of select="name(current())"/>
      </td>
    </xsl:for-each>
  </tr>
</xsl:template>

<xsl:template name="tabellenkoerper">
  <xsl:for-each select="/Lager/Artikel">
    <tr>
      <xsl:for-each select=".*">
        <td>
          <xsl:value-of select="current()"/>
        </td>
      </xsl:for-each>
    </tr>
  </xsl:for-each>
</xsl:template>

</xsl:stylesheet>

```

Listing 7.27 lagertab.xslt

Als output-Methode wird "html" verwendet. Das Stylesheet baut die für die HTML-Seite notwendigen Elemente <html>, <head> und <body> in der ersten Template-Regel ein, indem es die entsprechenden Anweisungen in das Ergebnisdokument schreibt.

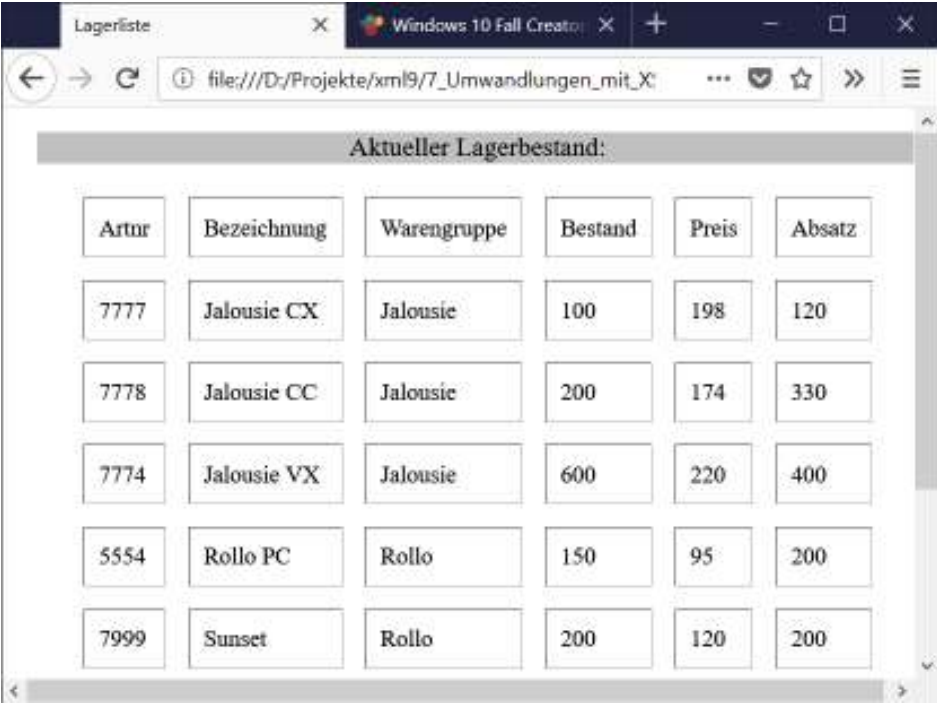
7.16.2 Tabellenkopf mit CSS

Um Ihnen zu zeigen, dass in XSLT-Stylesheets für HTML-Ergebnisdokumente ohne Weiteres auch CSS eingebaut werden kann, haben wir über ein `<style>`-Element zwei CSS-Definitionen für die Elemente `<table>` und `<caption>` eingefügt.

Von der Template-Regel, die dem Wurzelknoten zugeordnet ist, werden nacheinander zwei benannte Templates aufgerufen. Das erste sorgt dafür, dass die Tabelle eine Spaltenbeschriftung erhält, die hier automatisch aus den Elementnamen der Kinder des Elements `<Artikel>` erzeugt wird. Dazu wird der XPath-Ausdruck `"//Artikel[1]/*"` verwendet. Das heißt, es wird der erste Artikel verwendet und dann eine Knotenliste mit allen Kindelementen dieses Artikels abgearbeitet. Von dem jeweils aktuellen Knoten wird der Name in den Kopf der Tabellenspalte übernommen.

7.16.3 Aufbau einer Tabelle

Das zweite Template sorgt dann für die Verteilung der Daten auf die einzelnen Tabellenzellen. Dazu wird mit einer doppelten `<xsl:for-each>`-Schleife gearbeitet. Die erste sorgt dafür, dass alle Artikel bearbeitet werden, die zweite übernimmt bei dem gerade bearbeiteten Artikel die String-Werte der Kindelemente. [Abbildung 7.22](#) zeigt das Ergebnis im Firefox-Browser.



Artnr	Bezeichnung	Warengruppe	Bestand	Preis	Absatz
7777	Jalousie CX	Jalousie	100	198	120
7778	Jalousie CC	Jalousie	200	174	330
7774	Jalousie VX	Jalousie	600	220	400
5554	Rollo PC	Rollo	150	95	200
7999	Sunset	Rollo	200	120	200

Abbildung 7.22 Die Lagerliste im Firefox-Browser mit den hinzugefügten Beschriftungen

7.16.4 Transformation in XHTML

Eine der wichtigen aktuellen XML-Anwendungen ist *XHTML*, eine Sprache, die zunächst als Umformulierung von HTML 4.01 entstanden ist. Sie war gedacht als Brücke, die schließlich von HTML zu XML führen sollte. Auf die aktuelle Rolle von XHTML als alternative Syntax für HTML5 werden wir in Kapitel 15 noch einmal eingehen.

Das W3C hatte die Empfehlung für XHTML 1.0 als Ausgangspunkt für eine ganze Familie von XHTML-Dokumenttypen und -Modulen im Januar 2000 verabschiedet (siehe www.w3.org/TR/xhtml1/).

XHTML 1.0 ist in drei Varianten verfügbar, die über entsprechende DTDs definiert sind. Diese DTDs bilden die entsprechenden HTML 4.01-DTDs nach. Die DTDs sollten immer angegeben werden, weil sonst der Browser in der Regel in den sogenannten Quirks-Modus wechselt, bei dem insbesondere die Umsetzung von CSS häufig fehlerhaft ist.

XHTML Strict

Diese Version ist ganz auf die Auszeichnung von Inhalten konzentriert. Um die Daten im Browser darzustellen, sind deshalb in der Regel Stylesheets erforderlich. Die notwendige Dokumenttyp-Deklaration ist:

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

Als MIME-Typ wird "application/xhtml+xml" erwartet, "text/html" wird aber auch hier toleriert.

XHTML Transitional

Diese weniger strenge Variante ist für ältere Browser gedacht, die noch keine Stylesheets unterstützen, und erlaubt noch Elemente wie `` und `<center>`.

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

Die meisten Seiten, die XHTML verwenden, arbeiten mit dieser DTD. In der Regel wird diese Dokumenttyp-Deklaration mit der Angabe kombiniert, dass der MIME-Typ "text/html" verwendet werden soll, was diese DTD erlaubt. Das führt am Ende dazu, dass der Browser die Seite wie eine HTML-Seite behandeln kann.

XHTML Frameset

Dies ist eine spezielle Variante, bei der XHTML Strict um die Unterstützung von Frames erweitert ist. Dabei wird das Element `<body>` durch `<frameset>` ersetzt.


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

7.16.5 XHTML-Module

Die Idee, XHTML zu modularisieren, also in kombinierbare Bausteine zu zerlegen, ergab sich aus der Tatsache, dass inzwischen von ganz unterschiedlichen Gerätetypen und Plattformen aus auf Webinhalte zugegriffen wird. Browser, die auf ein Handy zugeschnitten werden, können so beispielsweise einen anderen Satz von XHTML-Modulen unterstützen als ein TV-Gerät, das für den Zugriff aufs Internet verwendet wird, oder ein normaler PC.

Das W3C hatte 2001 eine Empfehlung für die Modularisierung von XHTML veröffentlicht – *XHTMLMOD* – und auf dieser Basis eine Empfehlung für *XHTML 1.1* als modulbasiertes XHTML. Diese unterscheidet sich von *XHTML 1.0 Strict* zunächst aber nur minimal, hauptsächlich durch die Möglichkeit, `<ruby>`-Elemente zuzulassen. Bei diesem Element wird eine Verknüpfung zwischen einem Basistext und einem Kürzel für diesen Basistext hergestellt, wie etwa:

```
<ruby>
  <rb>WWW</rb>
  <rt>World Wide Web</rt>
</ruby>
```

Außerdem wurde das bei allen Elementen verfügbare Element `lang` durch `xml:lang` ersetzt, bei den Elementen `<a>` und `<map>` wurde `name-Attribut` durch das `id-Attribut` ersetzt. Die Dokumenttyp-Deklaration sieht so aus:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1 //EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
```

XHTML1.1 verhindert, dass fehlerhafter Code vom Browser einfach als HTML-Code behandelt wird.

Ein neueres Beispiel für einen solchen modularisierten Dokumenttyp ist *XHTML Basic 1.1*, festgelegt in einer Empfehlung vom Juli 2008, die einen eingeschränkten Satz von XHTML-Features enthält und für PDAs, Settop-Boxen, Pager und Handys gedacht ist.

Die Arbeit an der lange erwarteten Neufassung XHTML 2.0 wurde vom W3C 2007 zugunsten von HTML5 gestoppt und XHTML5 in der Folge nur noch als zweite Syntax von dem, was mit HTML als erster Syntax spezifiziert ist, behandelt. Darauf gehen wir in [Kapitel 15](#), »HTML5 und XHTML«, noch näher ein.

7.16.6 Allgemeine Merkmale von XHTML

Der entscheidende Unterschied zwischen XHTML und HTML liegt darin, dass es sich bei jedem XHTML-Dokument bereits um ein gültiges XML-Dokument handelt. Die Werkzeuge, mit denen auf XML-Daten zugegriffen werden kann, etwa um die Gültigkeit zu prüfen, stehen damit auch für XHTML zur Verfügung. Gleichzeitig kann ein Browser jedoch in der Regel ein XHTML- wie ein HTML-Dokument wiedergeben.

Das Skelett eines XHTML-Dokuments mit der Transitional-DTD sieht so aus:

```
<?xml version="1.0" encoding="UTF-8"?>
<!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> ... </title>
  </head>
  <body>
    <p> ... </p>
  </body>
</html>
```

7.16.7 Aufbau eines XHTML-Dokuments

Dass es sich um ein XML-Dokument handelt, wird schon aus der XML-Deklaration am Anfang sichtbar. Diese Zeile kann allerdings weggelassen werden, wenn mit der Zeichencodierung UTF-8 oder UTF-16 gearbeitet wird. Bei anderen Codierungen wie ISO-8859-1 muss die Deklaration verwendet werden.

Dahinter muss eine DOCTYPE-Deklaration folgen, die das Dokument einer der möglichen DTDs zuordnet. Anstatt die öffentliche DTD des W3Cs über den URI aufzurufen, kann auch eine lokale Kopie der DTD auf dem Webserver abgelegt werden. Dies kann das Laden der Seite etwas beschleunigen.

Das oberste Element ist immer `<html>`. Darin muss der für XHTML reservierte Namensraum unter `http://www.w3.org/1999/xhtml` über das `xmlns`-Attribut angegeben werden. Alle Elemente im Dokument müssen zu diesem Namensraum gehören.

Die wichtigsten Merkmale im Vergleich zu HTML sind folgende:

- ▶ XHTML-Dokumente müssen als XML-Anwendungen wohlgeformt sein. Elemente müssen also korrekt geschachtelt sein und dürfen sich nicht überlappen.
- ▶ Alle Tag- und Attributnamen werden kleingeschrieben (einzige Ausnahme ist DOCTYPE).
- ▶ Es sind nur vollständige Tags erlaubt, das heißt, für jedes Start-Tag muss ein entsprechendes End-Tag vorhanden sein. Nur bei leeren Elementen ist die für XML erlaubte Kurzform möglich, zum Beispiel `<p/>` für einen leeren Absatz. (Bei einigen

Browsern sollte noch ein Leerzeichen vor den Schrägstrich gesetzt werden, damit sie besser zwischen leeren Tags und Container-Tags unterscheiden können.)

- ▶ Attributwerte müssen immer in Anführungszeichen eingeschlossen werden. Es ist nicht erlaubt, ein Attribut ohne Wertangabe zu verwenden. Notfalls sollten also Dummy-Werte angegeben werden. Für `<hr noshade>` wird in XHTML `<hr noshade="noshade" />` verwendet, der Attributname also als Attributwert angegeben.
- ▶ Script- und Style-Elemente sollten als CDATA-Blöcke eingefügt werden.
- ▶ Für einige Elemente gelten Einschränkungen in Bezug auf Elemente, die sie enthalten dürfen:
 - `<a>` darf kein `<a>` enthalten.
 - `<pre>` darf kein ``, `<object>`, `<big>`, `<small>`, `<sub>` oder `<sup>` enthalten.
 - `<button>` darf kein `<input>`, `<select>`, `<textarea>`, `<label>`, `<button>`, `<form>`, `<fieldset>`, `<iframe>` oder `<isindex>` enthalten.
 - `<label>` darf kein `<label>` enthalten.
 - `<form>` darf kein `<form>` enthalten.
- ▶ Statt des `name`-Attributs muss das `id`-Attribut verwendet werden.

7.16.8 Automatische Übersetzung

Aktuelle Webeditoren wie Microsoft Expression Web erlauben eine automatische Übersetzung von HTML in XHTML. [Abbildung 7.23](#) zeigt ein Beispiel einer HTML-Seite, bei der eine ganze Reihe von End-Tags weggelassen worden ist.

```

1 <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
2 <html>
3   <head>
4     <meta content="text/html; charset=utf-8" http-equiv="Content-Type">
5     <title>Eine Liste</title>
6   </head>
7
8   <body>
9
10    <!-- kommentar -->
11    <ul>
12      <li>Eine Liste
13      <li>ist eine Liste
14      <li>ist eine Liste.
15    </ul>
16    <ol start="42">
17      <li>Eine Aufzählung
18      <li>muss nicht bei 1 beginnen.
19    </ol>
20
21  </body>
22
23 </html>
```

Abbildung 7.23 HTML-Code mit zahlreichen unvollständigen Markups

Nach der Übersetzung in XHTML sieht das Markup für die Listenelemente aus wie in [Abbildung 7.24](#).

```

<ul>
  <li>Eine Liste</li>
  <li>ist eine Liste</li>
  <li>ist eine Liste.</li>
</ul>
<ol start="42">
  <li>Eine Aufzählung</li>
  <li>muss nicht bei 1 beginnen.</li>
</ol>

```

Abbildung 7.24 Der XHTML-Code mit den vervollständigten Markups

Sie können sich auch das ursprünglich von Dave Raggett entwickelte *Tidy*-Tool für alle möglichen Plattformen und auch als Source aus dem Web herunterladen, dessen aktuelle Version unter <http://www.html-tidy.org> angeboten wird.

Wenn XML-Daten in einem Webbrowser ausgegeben werden sollen, liegt es nahe, mit XSLT statt einer Umwandlung in HTML eine Umwandlung in XHTML vorzunehmen. Das Verfahren entspricht weitgehend demjenigen, welches für HTML beschrieben worden ist, nur müssen die aufgeführten Unterschiede beachtet werden.

7.17 XSLT-Editoren

Die Entwicklung von XSLT per Hand in einem einfachen Texteditor ist zweifellos ein etwas mühsames Unterfangen. Deshalb gibt es Werkzeuge wie den XSLT-Designer *StyleVision* aus dem XMLSpy-Paket, die einen visuellen Entwurf eines Stylesheets erlauben. Der entsprechende Code wird dabei automatisch generiert und kann bei Bedarf nachbearbeitet werden.



Abbildung 7.25 Fenster des XSLT-Designers mit einem ersten Designentwurf

Die Vorgehensweise sieht im Prinzip so aus, dass zunächst eine Datei geöffnet wird, die das zugrunde liegende Schema oder eine DTD enthält. Für das Design des Stylesheets wird eine der Struktur entsprechende XML-Dokumentinstanz zugeordnet.

Um das Stylesheet zu gestalten, werden in dem Elementbaum des Schemas die gewünschten Komponenten ausgewählt und in die Designansicht, und zwar an die gewünschte Stelle, herübergezogen, die durch den Einfüge-Cursor bestimmt wird.

Ist ein Element in der Designansicht ausgewählt, können über die übrigen Register in dem Fenster alle Eigenschaften zugeordnet werden, die sich für die Textgestaltung, die Positionierung und das Layout zuweisen lassen.

Zur Kontrolle dient das Vorschauregister für HTML, das die Ansicht im Browser simuliert.



Abbildung 7.26 Vorschau auf dem HTML-Register

Der generierte Code kann über den Befehl DATEI • GENERIERTE DATEIEN SPEICHERN als XSLT-Datei gespeichert und in der XMLSpy-Umgebung nachbearbeitet werden, wenn es nötig ist.

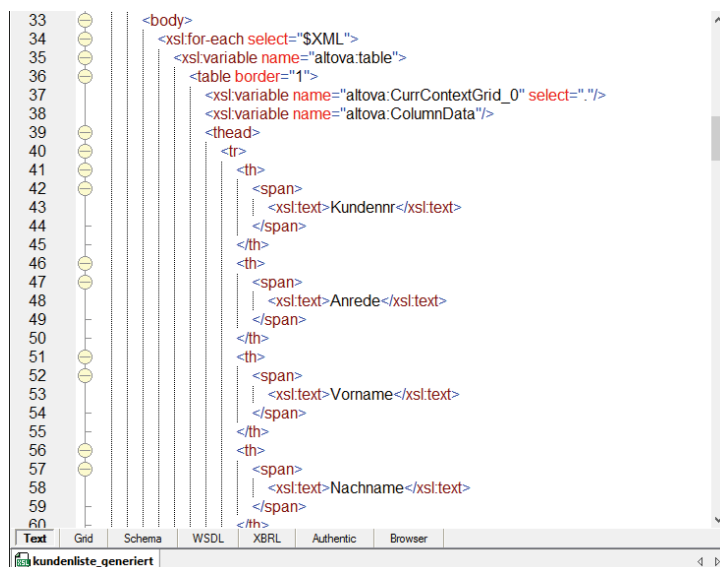


Abbildung 7.27 Das Register »Text« in XMLSpy zeigt den generierten XSLT-Code.

7.18 Kurzreferenz zu XSLT 1.0

<xsl:apply-imports/>

Diese Anweisung sorgt dafür, dass auf den aktuellen Knoten nur die in das Stylesheet importierten Template-Regeln angewendet werden. Es gibt keine Inhalte oder Attribute.

<xsl:apply-templates>

Diese Anweisung wendet auf die von `select` bestimmte Knotenmenge die jeweils passenden Template-Regeln an.

```
<xsl:apply-templates
  select = Knotenmengenausdruck
  mode = QName>
  <!-- Inhalt: (xsl:sort | xsl:with-param)* -->
</xsl:apply-templates>
```

<xsl:attribute>

Die Anweisung liefert ein XML-Attribut zu dem aktuellen Element.

```
<xsl:attribute
  name = { QName }
  namespace = { URI-Referenz }>
  <!-- Inhalt: Template -->
</xsl:attribute>
```

<xsl:attribute-set>

Die Anweisung definiert einen Satz von Attributen, der innerhalb eines Elements mit der Anweisung `<xsl:use-attribute-set>` anstelle einzelner Attribute verwendet werden kann.

```
<!-- Top-Level-Element -->
<xsl:attribute-set
  name = QName
  use-attribute-sets = QNamen>
  <!-- Inhalt: xsl:attribute* -->
</xsl:attribute-set>
```

<xsl:call-template>

Ruft die Template-Regel mit dem angegebenen Namen auf.

```
<xsl:call-template
  name = QName>
  <!-- Inhalt: xsl:with-param* -->
</xsl:call-template>
```

<xsl:choose>

Wertet das Template der ersten `when`-Klausel aus, wenn die angegebene Bedingung erfüllt ist. Im anderen Fall wird das Template der `otherwise`-Klausel verwendet.

```
<xsl:choose>
  <!-- Inhalt: (xsl:when+, xsl:otherwise?) -->
</xsl:choose>
```

<xsl:comment>

Liefert einen XML-Kommentar, der das angegebene Template als Text enthält.

```
<xsl:comment>
  <!-- Inhalt: Template -->
</xsl:comment>
```

<xsl:copy>

Kopiert den aktuellen Kontextknoten in das resultierende Baumfragment, zusammen mit einem eventuellen Namensraumknoten.

```
<xsl:copy
  use-attribute-sets = QNamen>
  <!-- Inhalt: Template -->
</xsl:copy>
```

<xsl:copy-of>

Gibt die dem Ausdruck entsprechende Knotenmenge aus.

```
<xsl:copy-of
  select = Ausdruck />
```

<xsl:decimal-format>

Legt ein Zahlenformat fest, das benutzt werden soll, wenn Zahlen in das Ergebnisdokument geschrieben werden sollen. Wird ein Name zugewiesen, kann das Format über die Funktion `format-number()` gezielt verwendet werden, ansonsten wird das angegebene Format als Standardformat behandelt. Mehrere benannte Formate können definiert werden.

```

<!-- Top-Level-Element -->
<xsl:decimal-format
  name = QName
  decimal-separator = char
  grouping-separator = char
  infinity = string
  minus-sign = char
  NaN = string
  percent = char
  per-mille = char
  zero-digit = char
  digit = char
  pattern-separator = char />

```

<xsl:element>

Gibt ein XML-Element aus mit dem lokalen Teil des angegebenen Namens, das zum angegebenen Namensraum gehört und dessen Kinder auf dem angegebenen Template beruhen.

```

<xsl:element
  name = { QName }
  namespace = { URI-Referenz }
  use-attribute-sets = QNamen>
  <!-- Inhalt: Template -->
</xsl:element>

```

<xsl:fallback>

Wertet das angegebene Template aus, wenn die Eltern-Instruktion vom Prozessor nicht verarbeitet wird. Wird hauptsächlich verwendet, wenn Erweiterungselemente eingesetzt werden, bei denen nicht sicher ist, ob sie vom XSLT-Prozessor verarbeitet werden können.

```

<xsl:fallback>
  <!-- Inhalt: Template -->
</xsl:fallback>

```

<xsl:for-each>

Erzeugt eine Schleife, um alle Knoten der ausgewählten Knotenmenge mit dem Template auszuwerten, wobei die Reihenfolge durch eine Sortierung mit `xsl:sort` beeinflusst werden kann.


```
<xsl:for-each
  select = Knotenmengenausdruck>
  <!-- Inhalt: (xsl:sort*, template) -->
</xsl:for-each>
```

<xsl:if>

Wertet das Template nur dann aus, wenn der Testausdruck den Wert *wahr* ergibt.

```
<xsl:if
  test = logischer Ausdruck>
  <!-- Inhalt: Template -->
</xsl:if>
```

<xsl:import>

Importiert das angegebene Stylesheet. Variablen und Templates, die importiert werden, haben einen geringeren Rang als die Variablen und Templates des importierenden Stylesheets selbst.

```
<xsl:import
  href = URI-Referenz />
<!-- Top-Level-Element -->
```

<xsl:include>

Übernimmt das angegebene Stylesheet. Die darin enthaltenen Templates und Variablen sind mit denen des inkludierenden Stylesheets gleichrangig.

```
<xsl:include
  href = URI-Referenz />
<!-- Top-Level-Element -->
```

<xsl:key>

Wird verwendet, um Knoten im aktuellen Dokument zu indizieren. Die benannten Schlüssel können mit der Funktion `key()` in XPath-Ausdrücken und -Mustern benutzt werden.

```
<xsl:key
  name = QName
  match = pattern
  use = Ausdruck />
<!-- Top-Level-Element -->
```

<xsl:message>

Gibt eine Nachricht aus; außerdem kann damit die Ausführung des Stylesheets gestoppt werden.

```
<xsl:message
  terminate = "yes" | "no">
  <!-- Inhalt: Template -->
</xsl:message>
```

<xsl:namespace-alias>

Definiert ein Alias für einen Namensraum. Wird hauptsächlich benötigt, wenn ein Stylesheet ein anderes Stylesheet als Ergebnisdokument erzeugen soll und die direkte Verwendung des Namensraums zu Schwierigkeiten führen würde.

```
<!-- Top-Level-Element -->
<xsl:namespace-alias
  stylesheet-prefix = prefix | "#default"
  result-prefix = prefix | "#default" />
```

<xsl:number>

Erzeugt eine Zahl aus dem in value angegebenen Ausdruck.

```
<xsl:number
  level = "single" | "multiple" | "any"
  count = pattern
  from = pattern
  value = numerischer Ausdruck
  format = { string }
  lang = { nmtoken }
  letter-value = { "alphabetic" | "traditional" }
  grouping-separator = { char }
  grouping-size = { number } />
```

<xsl:otherwise>

Definiert innerhalb eines <xsl:choose>-Elements den Fall, dass keiner der when-Tests bestanden wird.

```
<xsl:otherwise>
  <!-- Inhalt: Template -->
</xsl:otherwise>
```

<xsl:output>

Definiert die wesentlichen Merkmale des Ergebnisdokuments, die der XSLT-Prozessor gewährleisten soll.

```
<!-- Top-Level-Element -->
<xsl:output
  method = "xml" | "html" | "text" | QName-aber-nicht-NCName
  version = nmtoken
  encoding = string
  omit-xml-declaration = "yes" | "no"
  standalone = "yes" | "no"
  doctype-public = string
  doctype-system = string
  cdata-section-elements = QNamen
  indent = "yes" | "no"
  media-type = string />
```

<xsl:param>

Definiert einen Parameter, der vom Stylesheet insgesamt oder von einer Vorlage benutzt werden soll.

```
<!-- Top-Level-Element --> oder Teil eines Templates
<xsl:param
  name = QName
  select = Ausdruck>
  <!-- Inhalt: Template -->
</xsl:param>
```

<xsl:preserve-space>

Mit Hilfe dieses Elements können die Elemente im Quelldokument bestimmt werden, bei denen Leerzeichen, Leerzeilen und Tabulatorsprünge erhalten bleiben sollen.

```
<!-- Top-Level-Element -->
<xsl:preserve-space
  elements = tokens />
```

<xsl:processing-instruction>

Liefert eine Processing Instruction mit dem angegebenen Ziel.

```
<xsl:processing-instruction
  name = { ncname }>
  <!-- Inhalt: Template -->
</xsl:processing-instruction>
```

<xsl:sort>

Sortiert die aktuelle Knotenliste, die durch das jeweilige Elternelement – `<xsl:apply-templates>` oder `<xsl:for-each>` – gegeben ist; kann mehrfach verwendet werden, um Mehrfach-Schlüssel zu definieren.

```
<xsl:sort
  select = string-Ausdruck
  lang = { nmtoken }
  data-type = { "text" | "number" | QName-aber-nicht-NCName }
  order = { "ascending" | "descending" }
  case-order = { "upper-first" | "lower-first" } />
```

<xsl:strip-space>

Mit Hilfe dieses Elements können Elemente im Quelldokument bestimmt werden, bei denen Leerzeichen, Leerzeilen und Tabulatorsprünge entfernt werden sollen.

```
<!-- Top-Level-Element -->
<xsl:strip-space
  elements = tokens />
```

<xsl:stylesheet>

Dieses Element ist das Wurzelement eines XSLT-Stylesheets. Als Synonym kann auch `<xsl:transform>` verwendet werden.

```
<xsl:stylesheet
  id = id
  extension-element-prefixes = tokens
  exclude-result-prefixes = tokens
  version = number>
  <!-- Inhalt: (xsl:import*, Top-Level-Elemente) -->
</xsl:stylesheet>
```

<xsl:template>

Definiert ein Template für das Ergebnisdokument. Template-Regeln, die mit dem `match`-Attribut arbeiten, werden ausgeführt, wenn eine Knotenmenge gefunden wird, die dem XPath-Muster entspricht. Templates, die benannt sind, werden dagegen über den Namen mit `<xsl:call-template>` aufgerufen.

```
<!-- Top-Level-Element -->
<xsl:template
  match = pattern
  name = QName
```

```

    priority = number
    mode = QName>
    <!-- Inhalt: (xsl:param*, template) -->
</xsl:template>

```

<xsl:text>

Liefert den in #PCDATA gefundenen Text, wobei sich der Schutz der eingebauten Entitäten kontrollieren lässt.

```

<xsl:text
  disable-output-escaping = "yes" | "no">
  <!-- Inhalt: #PCDATA -->
</xsl:text>

```

<xsl:transform>

Kann anstelle des Elements <xsl:stylesheet> verwendet werden, etwa um deutlich zu machen, dass nur Transformationen und keine Formatierungen vorgenommen werden sollen.

```

<xsl:transform
  id = id
  extension-element-prefixes = tokens
  exclude-result-prefixes = tokens
  version = number>
  <!-- Inhalt: (xsl:import*, Top-Level-Elements) -->
</xsl:transform>

```

<xsl:value-of>

Liefert die Zeichenkette, die dem angegebenen XPath-Ausdruck entspricht, und überträgt sie in das Ergebnisdokument.

```

<xsl:value-of
  select = Ausdruck
  disable-output-escaping = "yes" | "no" />

```

<xsl:variable>

Deklariert eine Variable mit dem angegebenen Namen und initialisiert sie entweder mit dem select-Wert oder durch das Template.

```

<!-- Top-Level-Element --> oder innerhalb eines Templates
<xsl:variable
  name = QName

```

```

    select = Ausdruck>
    <!-- Inhalt: Template -->
</xsl:variable>

```

<xsl:when>

Ist eine der Wahlmöglichkeiten in einem <xsl:choose>-Element.

```

<xsl:when
  test = boolean-Ausdruck>
  <!-- Inhalt: Template -->
</xsl:when>

```

<xsl:with-param>

Wird verwendet, um die Werte von Parametern zu bestimmen, die beim Aufruf eines Templates eingesetzt werden sollen.

```

<xsl:with-param
  name = QName
  select = Ausdruck>
  <!-- Inhalt: Template -->
</xsl:with-param>

```

7.19 XSLT 2.0

Wie schon angesprochen, hat das W3C im Januar 2007 eine erweiterte Empfehlung für XSL Transformations (XSLT) herausgebracht. *XSLT 2.0* ist entwickelt worden, um die Gestaltungsmöglichkeiten durch Stylesheets zu erweitern und um einige Schwächen von XSLT 1.0 zu beseitigen.

7.19.1 Die wichtigsten Neuerungen

Durch die gleichzeitige Verabschiedung von XPath 2.0 wird insbesondere der Zugriff auf die von einem Stylesheet zu verarbeitenden Daten wesentlich verfeinert und erweitert. XSLT 2.0 profitiert hier in erster Linie von dem in [Kapitel 5](#), »Navigation und Verknüpfung«, bereits beschriebenen Ausbau des Datenmodells und der Einführung zahlreicher neuer Funktionen und Operatoren in XPath 2.0. Insbesondere die Unterstützung von XML Schema schafft hier viele neue Möglichkeiten. Gleichzeitig bleibt die Unterstützung von XPath 1.0 weitgehend erhalten.

Ablösung der temporären Knotenbäume

In [Abschnitt 7.11.5](#), »Typische Anwendungen von Variablen in XSLT«, sind die temporären Knotenbäume – *Result Tree Fragment* – in XSLT 1.0 als eine Möglichkeit beschrieben worden, Teile des Ergebnisbaumes innerhalb eines `<xsl:variable>`-Elements festzuhalten. Die Handhabung solcher Fragmente unterliegt gewissen Einschränkungen, die sich in der Praxis als eher hinderlich erwiesen haben. Aus diesem Grund wurden diese Fragmente in XSLT 2.0 abgeschafft.

Stattdessen kann nun bei einem `<xsl:variable>`-Element eine Sequenz von Knoten verwendet werden. Diese Sequenz stellt alle Bearbeitungsmöglichkeiten zur Verfügung, die für Sequenzen in XSLT 2.0 gegeben sind. Der Wert der Variablen wird dabei jeweils als Wurzelement behandelt. Die Navigation innerhalb des temporären Baumes kann dann mit den üblichen XPath-Ausdrücken erfolgen, wie das folgende Beispiel zeigt:

```
<xsl:template match="/">
  <xsl:variable name="temp">
    <xsl:element name="anrede">
      <xsl:element name="a">Frau</xsl:element>
      <xsl:element name="a">Herr</xsl:element>
      <xsl:element name="a">Firma</xsl:element>
    </xsl:element>
  </xsl:variable>
  <result>
    <xsl:value-of select="$temp/anrede/a[3]" />
  </result>
</xsl:template>
```

Das Ergebnis dieser Template-Regel ist ein Knoten mit dem Wert *Firma*.

Die Auswahl aus einer Sequenz lässt sich mit dem neuen `<xsl:sequence>`-Element allerdings noch einfacher erreichen.

```
<xsl:template match="/">
  <xsl:variable name="anrede" as="xs:string*">
    <xsl:sequence select="'Frau', 'Herr', 'Firma'" />
  </xsl:variable>
  <result>
    <xsl:value-of select="$temp/[3]" />
  </result>
</xsl:template>
```

Dabei wird für die Variable über das neue `as`-Attribut mit `xs:string*` ein Datentyp angegeben, der eine Sequenz von 0 bis beliebig viele Zeichenketten erlaubt.

Gruppierungen

Eine besonders praktische Anweisung in XSLT 2.0 ist das Element `<xsl:for-each-group>`, das die Gruppierung von Sequenzen wesentlich vereinfacht, ganz gleich, ob es sich um Sequenzen von Knoten oder von atomaren Werten handelt. Das Element unterstützt vier Attribute für die Steuerung der gewünschten Gruppierung:

- ▶ `group-by`
Arbeitet mit einem Gruppierungsschlüssel und fasst alle Datenelemente zusammen, die in Bezug auf diesen Schlüssel denselben Wert liefern. Diese Methode beachtet übrigens nicht die Abfolge der Knoten. Duplikate in einer Gruppe werden zugelassen. Derselbe Knoten kann unter Umständen auch Mitglied verschiedener Gruppen sein.
- ▶ `group-adjacent`
Arbeitet mit einem Gruppierungsschlüssel und fasst alle Datenelemente zusammen, die in Bezug auf diesen Schlüssel denselben Wert liefern, allerdings mit der Einschränkung, dass die Datenelemente gleichzeitig innerhalb der Eingabesequenz benachbart sind.
- ▶ `group-starting-with`
In diesem Fall wird immer dann eine neue Gruppe begonnen, wenn eines der Datenelemente mit einem bestimmten Muster übereinstimmt. Diese Methode lässt sich anwenden, wenn die Elemente in einer bestimmten Anordnung vorliegen (dies gilt auch für die direkt im Folgenden beschriebene vierte Methode).
- ▶ `group-ending-with`
Es wird immer dann eine Gruppe beendet, wenn eines der Datenelemente mit einem bestimmten Muster übereinstimmt.

Um auf die aktuelle Gruppe zuzugreifen, kann die Funktion `current-group()` verwendet werden. Die neue Funktion `current-grouping-key()` liefert jeweils den aktuell verwendeten Schlüssel. Als Kindelement lässt sich `xsl:sort` benutzen, um die Reihenfolge zu bestimmen, in der die Gruppen verarbeitet werden sollen. Ein einfaches Beispiel für das erste Verfahren verwendet die folgende Lagertabelle:

```
<?xml version="1.0" encoding="UTF-8"?>
<Lager>
  <Artikel>
    <Artnr>7777</Artnr>
    <Bezeichnung>Jalousie CX</Bezeichnung>
    <Warengruppe>Jalousie</Warengruppe>
    <Preis>198</Preis>
  </Artikel>
  <Artikel>
    <Artnr>7778</Artnr>
```



```

    <Bezeichnung>Jalousie CC</Bezeichnung>
    <Warengruppe>Jalousie</Warengruppe>
    <Preis>174</Preis>
  </Artikel>
  <Artikel>
    <Artnr>7774</Artnr>
    <Bezeichnung>Jalousie VX</Bezeichnung>
    <Warengruppe>Jalousie</Warengruppe>
    <Preis>220</Preis>
  </Artikel>
  <Artikel>
    <Artnr>5554</Artnr>
    <Bezeichnung>Rollo PC</Bezeichnung>
    <Warengruppe>Rollo</Warengruppe>
    <Preis>95</Preis>
  </Artikel>
  <Artikel>
    <Artnr>7999</Artnr>
    <Bezeichnung>Sunset</Bezeichnung>
    <Warengruppe>Rollo</Warengruppe>
    <Preis>120</Preis>
  </Artikel>
  <Artikel>
    <Artnr>8444</Artnr>
    <Bezeichnung>Markise SK</Bezeichnung>
    <Warengruppe>Markise</Warengruppe>
    <Preis>280</Preis>
  </Artikel>
</Lager>

```

Listing 7.28 lagertab2.xml

Um daraus ein XML-Dokument zu erstellen, bei dem die Elemente nach dem Element `<Warengruppe>` gruppiert werden, kann folgende Template-Regel verwendet werden:

```

<xsl:template match="Lager">
  <Lager>
    <xsl:for-each-group select="/Lager/Artikel" group-by="Warengruppe">
      <Warengruppe>
        <Name><xsl:value-of select="Warengruppe"/></Name>
        <xsl:for-each select="current-group()">
          <Artikel>
            <Bezeichnung><xsl:value-of select="Bezeichnung"/>

```

```

        </Bezeichnung>
        <Preis><xsl:value-of select="Preis"/></Preis>
    </Artikel>
</xsl:for-each>
</Warengruppe>
</xsl:for-each-group>
</Lager>
</xsl:template>

```

Listing 7.29 lagertab2.xslt

Um die zu einer Gruppe gehörenden Daten zusammenzustellen, wird innerhalb der Schleife mit `<xsl:for-each-group>` eine zweite Schleife mit `<xsl:for-each select="current-group()">` eingebaut. Das Ergebnis sieht dann so aus:

```

<Lager>
  <Warengruppe>
    <Name>Jalousie</Name>
    <Artikel>
      <Bezeichnung>Jalousie CX</Bezeichnung>
      <Preis>198</Preis>
    </Artikel>
    <Artikel>
      <Bezeichnung>Jalousie CC</Bezeichnung>
      <Preis>174</Preis>
    </Artikel>
    ...
  </Warengruppe>
</Lager>

```

Ein Beispiel für das Attribut `group-starting-with` verwendet die folgende Titelliste:

```

<?xml version="1.0" encoding="UTF-8"?>
<titelliste>
  <titel samegroup="yes">Stromboli</titel>
  <titel samegroup="yes">Capri now</titel>
  <titel samegroup="no">No Rome</titel>
  <titel samegroup="yes">Go to Paris</titel>
  <titel samegroup="yes">Out of Bielefeld</titel>
  <titel samegroup="no">Londonderry</titel>
  <titel samegroup="yes">Blue Sky</titel>
  <titel samegroup="yes">Boomtown</titel>
</titelliste>

```

Listing 7.30 titelliste.xml

Die Template-Regel dafür:

```
<xsl:template match="titelliste">
  <titelliste>
    <xsl:for-each-group select="*"
      group-starting-with="titel[not(@samegroup='yes')]">
      <gruppe>
        <xsl:for-each select="current-group()">
          <titel><xsl:value-of select="."/></titel>
        </xsl:for-each>
      </gruppe>
    </xsl:for-each-group>
  </titelliste>
</xsl:template>
```

Listing 7.31 titelgruppe.xslt

Dieses Stylesheet erzeugt die folgende Ausgabe:

```
<?xml version="1.0" encoding="UTF-8"?>
<titelliste ...>
  <gruppe>
    <titel>Stromboli</titel>
    <titel>Capri now</titel>
  </gruppe>
  <gruppe>
    <titel>No Rome</titel>
    <titel>Go to Paris</titel>
    <titel>Out of Bielefeld</titel>
  </gruppe>
  <gruppe>
    <titel>Londonderry</titel>
    <titel>Blue Sky</titel>
    <titel>Boomtown</titel>
  </gruppe>
</titelliste>
```

Benutzerdefinierte Funktionen für XPath-Ausdrücke

Mit Hilfe der Anweisung `<xsl:function>` können in XSLT 2.0 benutzerdefinierte XSLT-Funktionen erzeugt werden, die sich dann in XPath 2.0-Ausdrücken verwenden lassen. Die Anweisung `xsl:function` legt dabei den Namen der neuen Funktion fest, bestimmt die benötigten Parameter und implementiert die gewünschte Funktionalität. Ist die Funktion definiert, kann sie im Stylesheet über den jeweiligen Namen aufgerufen werden.

Mehrere Ausgabedokumente

Mit XSLT 2.0 wird es auch möglich, über eine Transformation direkt mehrere Ergebnisbäume zu produzieren und auf diese Weise gleich mehrere Ausgabedokumente zu erzeugen. So können beispielsweise in einem Zug ein Gesamtdokument und eine Dokumentübersicht ausgegeben werden. Dazu werden einfach `<xsl:result-document>`-Elemente hintereinandergeschaltet, die jeweils die unterschiedlichen Ausgabebebaume beschreiben:

```
<xsl:template match="/">
  <xsl:result-document href="output1.xml">
    ... Beschreibung des ersten Ausgabebaums
  </xsl:result-document>
  <xsl:result-document href="output2.xml">
    ... Beschreibung des zweiten Ausgabebaums
  </xsl:result-document>
</xsl:template>
```

XHTML-Ausgabe

Während XSLT 1.0 nur die Ausgabe in XML, HTML oder Text anbot, unterstützt die Version 2.0 nun auch direkt die Ausgabe in XHTML über eine entsprechende Methode der `xsl:output-` oder der `xsl:result-document-`Anweisung.

7.19.2 Neue Funktionen in XSLT 2.0

Neben den zahlreichen neuen Funktionen in XPath 2.0, die in XSLT 2.0 über entsprechende Pfadausdrücke genutzt werden können, stehen einige eigene neue Funktionen zur Verfügung. Ein größerer Teil betrifft die Formatierung von Datums- und Zeitwerten. Die umständlichen Verfahren, solche Werte als Zeichenfolgen auszuwerten und in die verschiedenen Bestandteile zu zerlegen, werden auf diese Weise überflüssig. Die verwendeten Datentypen entsprechen dabei denen der XML Schema-Spezifikation.

Funktion	Beschreibung
<code>current-group</code>	Liefert die Inhalte der aktuellen Gruppe in <code>xsl:for-each-group</code> .
<code>current-grouping-key</code>	Liefert den aktuellen Gruppenschlüssel in <code>xsl:for-each-group</code> .
<code>format-date</code>	Formatiert ein Datum.

Tabelle 7.6 Liste zusätzlicher Funktionen in XSLT 2.0

Funktion	Beschreibung
format-dateTime	Formatiert einen datetime-Wert.
format-time	Formatiert einen Zeitwert.
regex-group	Liefert einen Satz von Teilstrings, die durch die <code>xsl:matching-substring</code> -Anweisung verfügbar sind.
type-available	Prüft die Verfügbarkeit eines bestimmten Datentyps.
unparsed-entity-public-id	Liefert den <i>Public Identifier</i> einer Ressource.
unparsed-text	Liest eine externe Ressource ein und liefert einen String.
unparsed-text-available	Prüft, ob sich eine externe Ressource mit <code>unparsed-text</code> einlesen lässt.

Tabelle 7.6 Liste zusätzlicher Funktionen in XSLT 2.0 (Forts.)

7.19.3 Neue Elemente in XSLT 2.0

Neben der schon angesprochenen Gruppenbearbeitung stellt XSLT 2.0 vor allem neue Anweisungen für die Behandlung von atomaren Werten und von Sequenzen zur Verfügung. Im Folgenden sind die neuen Elemente in XSLT 2.0 in einer Kurzreferenz zusammengestellt:

<xsl:analyze-string>

Wendet auf den String, den der `select`-Ausdruck liefert, den regulären Ausdruck an, der mit `regex` angegeben wird. Der String wird in Teilstrings zerlegt, die entweder mit dem regulären Ausdruck übereinstimmen oder nicht übereinstimmen. Auf diese Weise können in dem Eingabestring beispielsweise bestimmte Zeichen durch andere ersetzt werden.

```
<xsl:analyze-string
  select = Ausdruck
  regex = { string }
  flags? = { string }>
  <!-- Inhalt: (xsl:matching-substring?,
               xsl:non-matching-substring?, xsl:fallback*) -->
</xsl:analyze-string>
```

<xsl:character-map>

Wird verwendet, um während der Serialisierung gezielt bestimmte Zeichen durch andere zu ersetzen.

```
<xsl:character-map
  name = QName
  use-character-maps? = QNamen>
  <!-- Inhalt: (xsl:output-character*) -->
</xsl:character-map>
```

<xsl:document>

Wird verwendet, um einen neuen Dokumentknoten zu erzeugen.

```
<xsl:document
  validation? = "strict" | "lax" | "preserve" | "strip"
  type? = QName>
  <!-- Inhalt: Sequenzkonstruktor -->
</xsl:document>
```

<xsl:for-each-group>

Wird verwendet, um aus einer Sequenz von Knoten oder Einzelwerten Gruppen zu bilden.

```
<xsl:for-each-group
  select = Ausdruck
  group-by? = Ausdruck
  group-adjacent? = Ausdruck
  group-starting-with? = pattern
  group-ending-with? = pattern
  collation? = { uri }>
  <!-- Inhalt: (xsl:sort*, Sequenzkonstruktor) -->
</xsl:for-each-group>
```

<xsl:function>

Wird verwendet, um benutzerdefinierte XSLT-Funktionen zu erzeugen, die von XPath 2.0-Ausdrücken aufgerufen werden können.

```
<xsl:function
  name = QName
  as? = Sequenztyp
  override? = "yes" | "no">
  <!-- Inhalt: (xsl:param*, Sequenzkonstruktor) -->
</xsl:function>
```

<xsl:import-schema>

Wird verwendet, um XML Schema-Komponenten zu identifizieren, die zur Verfügung stehen müssen, bevor Quelldokumente verfügbar sind.

```
<xsl:import-schema
  namespace? = URI-Referenz
  schema-location? = URI-Referenz>
  <!-- Inhalt: xs:schema? -->
</xsl:import-schema>
```

<xsl:matching-substring>

Wird innerhalb von `xsl:analyze-string` verwendet, um die vorgegebene Aktion zu bestimmen, die bei den Teilstrings stattfinden soll, die dem regulären Ausdruck entsprechen.

```
<xsl:matching-substring>
  <!-- Inhalt: Sequenzkonstruktor -->
</xsl:matching-substring>
```

<xsl:namespace>

Wird verwendet, um einen Namensraumknoten zu erzeugen.

```
<xsl:namespace
  name = { NCName }
  select? = Ausdruck>
  <!-- Inhalt: Sequenzkonstruktor -->
</xsl:namespace>
```

<xsl:next-match>

Wird verwendet, um die als Nächstes auszuführende Template-Regel zu bestimmen.

```
<xsl:next-match>
  <!-- Inhalt: (xsl:with-param | xsl:fallback)* -->
</xsl:next-match>
```

<xsl:non-matching-substring>

Wird innerhalb von `xsl:analyze-string` verwendet, um die vorgegebene Aktion zu bestimmen, die bei den Teilstrings stattfinden soll, die dem regulären Ausdruck nicht entsprechen.

```
<xsl:non-matching-substring>
  <!-- Inhalt: Sequenzkonstruktor -->
</xsl:non-matching-substring>
```

<xsl:output-character>

Wird verwendet, um einen einzelnen Eintrag innerhalb von `xsl:character-map` festzulegen.

```
<xsl:output-character
  character = char
  string = string />
```

<xsl:perform-sort>

Wird verwendet, um eine Sequenz zu sortieren.

```
<xsl:perform-sort
  select? = Ausdruck>
  <!-- Inhalt: (xsl:sort+, Sequenzkonstruktor) -->
</xsl:perform-sort>
```

<xsl:result-document>

Erweitert die Möglichkeiten der in XSLT 1.0 verwendeten `xsl:output`-Anweisung. Sie erlaubt es, die Eigenschaften des endgültigen Ausgabebaums detailliert festzulegen.

```
<xsl:result-document
  format? = { QName }
  href? = { URI-Referenz }
  validation? = "strict" | "lax" | "preserve" | "strip"
  type? = QName
  method? = { "xml" | "html" | "xhtml" | "text" |
    QName-but-not-ncname }
  byte-order-mark? = { "yes" | "no" }
  cdata-section-elements? = { QNamen }
  doctype-public? = { string }
  doctype-system? = { string }
  encoding? = { string }
  escape-uri-attributes? = { "yes" | "no" }
  include-content-type? = { "yes" | "no" }
  indent? = { "yes" | "no" }
  media-type? = { string }
  normalization-form? = { "NFC" | "NFD" | "NFKC" | "NFKD" |
    "fully-normalized" | "none" | nmtoken }
  omit-xml-declaration? = { "yes" | "no" }
  standalone? = { "yes" | "no" | "omit" }
  undeclare-prefixes? = { "yes" | "no" }
  use-character-maps? = QNamen
```



```

    output-version? = { nmtoken }>
    <!-- Inhalt: Sequenzkonstruktor -->
</xsl:result-document>

```

<xsl:sequence>

Wird innerhalb eines Sequenzkonstruktors verwendet, um eine Sequenz von Knoten oder Einzelwerten zu bilden.

```

<xsl:sequence
  select = Ausdruck>
  <!-- Inhalt: xsl:fallback* -->
</xsl:sequence>

```

7.20 XSLT 3.0

Im Juni 2017 erschien die Empfehlung für XSLT 3.0, editiert von Michael Kay, dessen Firma Saxonica auch für die Saxon-Prozessoren für XSLT, XQuery und XML Schema verantwortlich zeichnet. (Eine 30-tägige Testversion von Saxon-EE kann über www.saxonica.com heruntergeladen werden.) XSLT 3.0 unterstützt die XPath-Versionen 3.0 und 3.1, die dafür definierten Funktionen können also auch in den Stylesheets verwendet werden. Das Element `<xsl:stylesheet>` erhält das Attribut `version = "3.0"`. Generell wird von einem XSLT-Prozessor jetzt erwartet, dass er mit allen eingebauten Datentypen zurechtkommt, die sich mit XML Schema verwenden lassen.

7.20.1 Streams

Die wesentliche Neuerung in XSLT 3.0 ist die Unterstützung von Transformationen in Form von Streams. Eines der bisher bestehenden Hemmnisse für die Anwendung von XSLT bei sehr großen Datenmengen ist ja, dass die Knotenbäume der Quelldaten und die der Ausgabedaten immer komplett im Speicher aufgebaut werden müssen. Um den Umgang mit Streams zu ermöglichen und zu erleichtern, wurden neue Elemente wie `<xsl:source-dokument>` oder `<xsl:merge>` für das Einlesen und Mischen von Daten und Funktionen wie `fn:copy-of` und `fn:snapshot` entwickelt, die es erlauben, Input-Dokumente in Sequenzen von kleineren Teilbäumen zu zerlegen, die sich nacheinander abarbeiten lassen. Die neue `<xsl:mode>`-Deklaration bestimmt, ob die Verarbeitung als Stream möglich ist, und legt unter Umständen für Templates bestimmte Einschränkungen fest, die sicherstellen, dass das auch funktioniert.

Die Stream-Verarbeitung umgeht nicht nur eventuelle Speichergrenzen, sie erlaubt es auch, mit der Ausgabe von Daten bereits zu beginnen, bevor der komplette Input eingelesen ist. Ein einfaches Beispiel, wie eine Stream-Verarbeitung angestoßen wer-

den kann, ist das folgende Beispiel. Stellen Sie sich dazu eine Lagerdatei mit mehreren Millionen Artikeln vor, die nach Warengruppen aufgeteilt sind. Dann ist es möglich, das Gesamtlager in mehrere XML-Dateien aufzuteilen, die nacheinander verarbeitet werden können, so dass es nicht mehr notwendig ist, die gesamte Datenmasse auf einmal in den Speicher zu laden. Dabei hilft das Zusammenspiel der Elemente `<xsl:source-document>`, `<xsl:result-document>` und `<xsl:copy-of>`. Das Attribut `streamable="yes"` sorgt dafür, dass die Verarbeitung als Stream gestartet wird.

```
<xsl:source-document streamable="yes" href="gesamtlager.xml">
  <xsl:for-each select="lager">
    <xsl:for-each select="warengruppe">
      <xsl:result-document href="warengruppe{position()}.xml">
        <xsl:copy-of select="."/>
      </xsl:result-document>
    </xsl:for-each>
  </xsl:for-each>
</xsl:source-document>
```

Ob allerdings ein Stylesheet in der Lage ist, große Mengen an XML-Daten in Form von Streams zu verarbeiten, ist an zahlreiche Bedingungen geknüpft, die bei der Entwicklung des Stylesheets zu berücksichtigen sind. Generell lässt sich davon ausgehen, dass sich große Datenmengen dann problemlos als Streams verarbeiten lassen, wenn sie sich ohne Umstände in Teilbäume zerlegen lassen, die dann nacheinander verarbeitet werden können.

7.20.2 Packages

Ein zweiter Schwerpunkt der Neuerungen betrifft die Möglichkeiten, Stylesheets modular zusammenzufügen. Dazu wurde ein neues Konzept für Pakete entwickelt, die als Schnittstellen konzipiert sind. Die Pakete werden in XML-Dateien angelegt, deren Wurzelement `<xsl:packages>` ist. Sie werden auch als *Package-Manifest* angesprochen. Über das Attribut `name` muss der Name und über `package-version` die Versionsnummer angegeben werden.

Inhalt des Pakets sind Komponenten wie Funktionen, globale Variablen und Parameter, benannte Templates, Attributsätze etc. Optionale Kindelemente von `<xsl:packages>` sind `<xsl:use-package>` und `<xsl:expose>`. Ersteres erlaubt den Bezug auf andere, externe Pakete. Letzteres erlaubt die Festlegung, welche der Komponenten des Pakets in welchem Umfang nach außen sichtbar sind und also von dem Stylesheet, das das Paket benutzt, verwendet werden können. Auf diese Weise wird die Wiederverwendbarkeit des Codes vereinfacht. Ein Stylesheet kann ohne Weiteres aus mehreren Paketen zusammengefügt werden.

Die XSLT 3.0-Empfehlung enthält einen Link auf ein Muster-Stylesheet zur Umwandlung von XML in JSON, das als Paket angeboten wird. Hier einige kurze Auszüge daraus:

```
<?xml version="1.0" encoding="UTF-8"?>
...
<xsl:package
  name="http://www.w3.org/2013/XSLT/xml-to-json"
  package-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:j="http://www.w3.org/2013/XSLT/xml-to-json"
  exclude-result-prefixes="xs fn j" default-mode="j:xml-to-json"
  version="3.0">

  <xsl:variable name="quot" visibility="private">"</xsl:variable>
  <xsl:param name="indent-spaces" select="2"/>
  ...
  <xsl:mode name="indent" _streamable="{ $STREAMABLE}" visibility="public"/>
  <xsl:mode name="no-indent" _streamable="{ $STREAMABLE}"
    visibility="public"/>
  <xsl:mode name="key-attribute" streamable="false" on-no-match="fail"
    visibility="public"/>
  ...
  <xsl:function name="j:xml-to-json" as="xs:string" visibility="public">
    <xsl:param name="input" as="node()"/>
    <xsl:sequence select="j:xml-to-json($input, map{})"/>
  </xsl:function>
  ...
  <xsl:template match="/" mode="indent no-indent">
    <xsl:apply-templates mode="#current"/>
  </xsl:template>
  ...
```

7.20.3 Umgang mit Maps

Die Map als neuer Datentyp ist dafür gedacht, komplexere Datenstrukturen einfacher handhaben zu können. Dazu steht eine Reihe von neuen Elementen und Funktionen zur Verfügung.

Das folgende Listing zeigt ein einfaches Beispiel für den Aufbau und die Abfrage einer Map.

```

<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet 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:math="http://www.w3.org/2005/xpath-functions/math"
  xmlns:array="http://www.w3.org/2005/xpath-functions/array"
  xmlns:map="http://www.w3.org/2005/xpath-functions/map"
  xmlns:xhtml="http://www.w3.org/1999/xhtml"
  xmlns:err="http://www.w3.org/2005/xqt-errors"
  exclude-result-prefixes="array fn map math xhtml xs err" version="3.0">
  <xsl:output method="xml" version="1.0" encoding="UTF-8" indent="yes"/>
  <xsl:variable name="bookmap" as="map(xs:integer, xs:string)">
    <xsl:map>
      <xsl:map-entry key="9783842101715" select="'Excel-Ratgeber'"/>
      <xsl:map-entry key="9783842101739" select="'Excel-Handbuch'"/>
      <xsl:map-entry key="9783842101722" select="'Excel-Funktionen'"/>
    </xsl:map>
  </xsl:variable>
  <xsl:template match="/" name="xsl:initial-template">
    <xsl:for-each select="map:keys($bookmap)">
      <isbn key="{.}">
        <xsl:value-of select="map:get($bookmap, .)"/>
      </isbn>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>

```

Listing 7.32 bookman.xslt

7.20.4 XML und JSON

Die Empfehlung für XSLT 3.0 hat auch auf die Bedeutung reagiert, die das JavaScript-basierte Datenformat JSON – *JavaScript Object Notation* – in den letzten Jahren gewonnen hat. Insbesondere bei Webanwendungen und -diensten wird JSON inzwischen häufig anstelle von XML für den Datenaustausch zwischen Server und Client verwendet, etwa bei Anwendungen, die mit REST, einem Programmierparadigma für verteilte Systeme, arbeiten. Das JSON-Format ist zwar auch textorientiert wie XML, aber wesentlich einfacher, ein Vorteil, den sich die Programmierer nicht haben entgehen lassen. Allerdings ist JSON eben nur ein Datenformat, also keine Metasprache wie XML und auch nicht Teil einer ganzen Sprachfamilie, die Komponenten für die Definition von Datenmodellen, für die Transformation von Daten und für ihre Abfrage enthält. Auch das Element `<xsl:output>` wurde entsprechend um die neue

Methode "json" erweitert. Die bisherigen Methoden "html" und "xhtml" unterstützen jetzt HTML5 und XHTML5.

Das folgende Template zeigt, wie ein Schnipsel JSON-Code mit Hilfe der Funktion `fn:json-to-xml` in XML-Code umgewandelt werden kann.

```
<xsl:template match="/">
  <xsl:variable name="t" as="xs:string">
    {
      "Kennzeichen": {
        "GB": "Great Britain",
        "DE": "Deutschland"
      }
    }
  </xsl:variable>
  <xsl:sequence select="json-to-xml($t)" />
</xsl:template>
```

Abbildung 7.28 zeigt die generierte XML-Datei.



Abbildung 7.28 JSON, umgewandelt in XML

7.20.5 Iterationen

In Abschnitt 7.11.6, »Rekursive Templates«, ist ein Verfahren beschrieben, wie kumulierte Werte errechnet werden können, obwohl ja in XSLT die Einschränkung besteht und auch in XSLT 3.0 bestehen bleibt, dass Variablen nur einmal ein Wert zugewiesen werden kann. Das neue Element `<xsl:iterate>` vereinfacht solche rekursiven Aufrufe.

Das folgende kleine Beispiel zeigt, wie sich Werte aus einem Quelldokument auf einfache Weise aufaddieren lassen. Die XML-Datei enthält die Beträge von datierten Transaktionen.

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="file:///D:/Projekte/xml9/7_
Umwandlungen_mit_XSLT/transaktionen.xslt"?>
<transaktionen>
  <transaktion datum="2018-09-01" wert="12.00"/>
  <transaktion datum="2018-09-02" wert="8.00"/>
```

```

    <transaktion datum="2018-09-03" wert="-2.00"/>
    <transaktion datum="2018-09-04" wert="5.00"/>
</transaktionen>

```

Listing 7.33 transaktionen.xml

Im Stylesheet wählt das Element `<xsl:iterate>` zunächst die Knotenmenge der einzelnen Transaktionen aus der Quelldatei aus. Mit Hilfe des Parameters `kontostand` wird jeweils das erreichte Zwischenergebnis festgehalten, die Variable `neuerkontostand` erhält ihren Wert durch das Hinzuaddieren. Mit dem Element `<xsl:next-iteration>` wird die Wiederholung des Vorgangs angestoßen. Die Werte werden jeweils an die beiden Attribute des neuen Elements `<kontostand>` übergeben.


```

<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  version="3.0">
  <xsl:template match="/">
    <konto>
      <xsl:iterate select="transaktionen/transaktion">
        <xsl:param name="kontostand" select="0.00" as="xs:decimal"/>
        <xsl:variable name="neuerkontostand"
          select="$kontostand + xs:decimal(@wert)"/>
        <kontostand datum="{@datum}"
          wert="{format-number($neuerkontostand, '0.00')}"/>
        <xsl:next-iteration>
          <xsl:with-param name="kontostand"
            select="$neuerkontostand"/>
        </xsl:next-iteration>
      </xsl:iterate>
    </konto>
  </xsl:template>
</xsl:stylesheet>

```

Listing 7.34 transaktionen.xslt

Abbildung 7.29 zeigt die kleine XML-Datei, die das Stylesheet als Ausgabe liefert.



```

<?xml version="1.0" encoding="UTF-8"?>
<konto xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <kontostand datum="2018-09-01" wert="12.00"/>
  <kontostand datum="2018-09-02" wert="20.00"/>
  <kontostand datum="2018-09-03" wert="18.00"/>
  <kontostand datum="2018-09-04" wert="23.00"/>
</konto>

```

Abbildung 7.29 Ergebnis der Iteration

7.20.6 Akkumulatoren

Eine weitere Möglichkeit, kumulierte Werte zu gewinnen und damit umzugehen, wird mit dem neuen Element `<xsl:accumulator>` und seinem Kindelement `<xsl:accumulator-rule>` sowie den beiden Funktionen `fn:accumulator-before` und `fn:accumulator-after` bereitgestellt. Diese Elemente stellen im Stylesheet globale Zähler zur Verfügung. Solche Zähler sind insbesondere bei einer Verarbeitung als Stream nützlich, aber ebenso, wenn nicht mit Streams gearbeitet wird. Der mit diesem Element eingerichtete Zähler ist mit den Knoten in einem Knotenbaum verknüpft. Über die beiden Funktionen `fn:accumulator-before` und `fn:accumulator-after` kann jeweils der Zählerstand vor und nach Erreichen eines Kontextknotens festgehalten werden. Wie gezählt wird, kann über das Kindelement `<xsl:accumulator-rule>` festgelegt werden.

Eine `<xsl:accumulator>`-Deklaration kann allerdings nur als Top-Level-Element in einem Stylesheet-Modul erscheinen, also als direktes Kind von `<xsl:stylesheet>`.

Zur Demonstration der Arbeitsweise des globalen Zählers greifen wir noch einmal auf die Lagerdatei zurück, die bereits bei der Besprechung von XSLT 2.0 verwendet wurde. Hier nur ein kurzer Auszug:

```
<Lager>
  <Artikel>
    <Artnr>7777</Artnr>
    <Bezeichnung>Jalousie CX</Bezeichnung>
    <Warengruppe>Jalousie</Warengruppe>
    <Bestand>100</Bestand>
    <Preis>198</Preis>
  </Artikel>
...
```

Es soll jetzt darum gehen, eine neue XML-Datei zu erzeugen, die jedem Artikel eine fortlaufende Lagerposition zuweist und am Ende angibt, wie viele unterschiedliche Artikel vorhanden sind. Der Code kann so aussehen:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet 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:math="http://www.w3.org/2005/xpath-functions/
  math" xmlns:array="http://www.w3.org/2005/xpath-functions/array" xmlns:map=
  "http://www.w3.org/2005/xpath-functions/map" xmlns:xhtml="http://www.w3.org/
  1999/xhtml" xmlns:err="http://www.w3.org/2005/xqt-errors" exclude-result-
  prefixes="array fn map math xhtml xs err" version="3.0">
  <xsl:output method="xml" version="1.0" encoding="UTF-8" indent="yes"/>
  <xsl:accumulator name="Artikelanzahl" as="xs:integer"
    initial-value="0" streamable="no">
```

```

        <xsl:accumulator-rule match="Artikel"
            select="$value + 1">
        </xsl:accumulator-rule>
    </xsl:accumulator>
    <xsl:template match="Lager">
        <xsl:for-each select="//Artikel">
            <Lagerposition Artikelnr="{Artnr}" Bezeichnung="{Bezeichnung}">
                <xsl:value-of select="fn:accumulator-after('Artikelanzahl')" />
            </Lagerposition>
        </xsl:for-each>
        <Artikelanzahl>
            <xsl:value-of select="fn:accumulator-after('Artikelanzahl')" />
        </Artikelanzahl>
    </xsl:template>
</xsl:stylesheet>

```

Listing 7.35 lagertab3.xslt

Der Zähler erhält zunächst einen Namen, damit er angesprochen werden kann. Anschließend wird die Form der Berechnung, hier eine fortlaufende Zählung, festgelegt. Im Template wird innerhalb der Schleife eine fortlaufende Nummer für jeden Artikel erzeugt, und nach der Schleife wird der zuletzt erreichte Wert als Gesamtanzahl der Artikel ausgegeben. [Abbildung 7.30](#) zeigt das Ergebnis.

```

<?xml version="1.0" encoding="UTF-8"?>
<Lagerposition Artikelnr="7777" Bezeichnung="Jalousie CX">1</Lagerposition>
<Lagerposition Artikelnr="7778" Bezeichnung="Jalousie CC">2</Lagerposition>
<Lagerposition Artikelnr="7774" Bezeichnung="Jalousie VX">3</Lagerposition>
<Lagerposition Artikelnr="5554" Bezeichnung="Rollo PC">4</Lagerposition>
<Lagerposition Artikelnr="7999" Bezeichnung="Sunset">5</Lagerposition>
<Lagerposition Artikelnr="8444" Bezeichnung="Markise SK">6</Lagerposition>
<Artikelanzahl>6</Artikelanzahl>

```

Abbildung 7.30 Berechnungen mit einem Zähler

7.20.7 Unterstützung für XSLT 3.0?

Der neue Standard wird von den bekannten XML-Tools wie jenen von Altova oder Oxygen in unterschiedlichem Umfang unterstützt. Altova bietet dafür einen eigenen Prozessor an, der die neuen Streaming-Funktionen noch nicht unterstützt, aber andererseits auch das Einbinden anderer Prozessoren erlaubt. Oxygen bindet den Saxon-EE-Prozessor ein, bei dem die Streaming-Unterstützung über den Einstellungsdialog aktiviert werden kann.

Bei den Browsern sieht es bisher bescheiden aus. Chrome unterstützt die direkte Ausgabe von lokalen Dateien mit XSLT-Transformationen sowieso gar nicht. Bei den anderen Browsern reicht die Unterstützung in der Regel nur bis XSLT 2.0. Es wird sich

zeigen, ob die sehr komplexen Erweiterungen des XSLT-Standards sich in Zukunft auszahlen. Der Erfolg von JSON zeigt zumindest, dass viele Entwickler eher dazu neigen, einfachere Lösungen vorzuziehen.

7.20.8 Liste der zusätzlichen Funktionen in XSLT 3.0

Die folgende Tabelle listet die neuen Funktionen in alphabetischer Reihenfolge für die Präfixe `fn` und `map` auf.

Funktion	Bedeutung
<code>fn:accumulator-after</code>	Liefert den Wert des mit <code><xsl:accumulator></code> deklarierten Zählers nach Erreichen des Kontextknotens.
<code>fn:accumulator-before</code>	Liefert den Wert des mit <code><xsl:accumulator></code> deklarierten Zählers vor Erreichen des Kontextknotens.
<code>fn:available-system-properties</code>	Liefert eine Liste der Namen der Systemeigenschaften.
<code>fn:collation-key</code>	Generiert aus einer Zeichenfolge einen Schlüssel für eine Kollation.
<code>fn:copy-of</code>	Erzeugt eine tiefe Kopie der angegebenen Sequenz.
<code>fn:current-merge-group</code>	Liefert die Gruppe der Datenelemente, die aktuell von einer <code><xsl:merge></code> -Anweisung bearbeitet wird.
<code>fn:current-merge-key</code>	Liefert den Merge-Schlüssel aus der Gruppe der Datenelemente, die aktuell von einer <code><xsl:merge></code> -Anweisung bearbeitet wird.
<code>fn:current-output-uri</code>	Liefert den Wert des aktuellen Output-URI.
<code>fn:json-to-xml</code>	Wandelt einen String im JSON-Format in das XML-Format um.
<code>fn:snapshot</code>	Liefert eine Kopie einer Sequenz mit allen Elementen, einschließlich der Attribute und Namensräume.
<code>fn:stream-available</code>	Bestimmt, ob ein Dokument als Stream verarbeitet werden kann.

Funktion	Bedeutung
<code>fn:xml-to-json</code>	Wandelt einen String im XML-Format in das JSON-Format um.
<code>map:contains</code>	Prüft, ob es für den angegebenen Schlüssel in der Map einen Eintrag gibt.
<code>map:entry</code>	Liefert die Map, die den angegebenen Eintrag enthält.
<code>map:find</code>	Sucht in Map-Einträgen nach dem Wert, der zu dem angegebenen Schlüssel gehört.
<code>map:for-each</code>	Wendet die angegebenen Anweisungen auf alle Einträge einer Map an und liefert eine Zeichenverknüpfung aller Ergebnisse.
<code>map:get</code>	Liefert zu einem Schlüssel in der angegebenen Map den entsprechenden Wert.
<code>map:keys</code>	Liefert eine Sequenz aller Schlüssel in einer Map.
<code>map:merge</code>	Liefert eine Map aus den Einträgen der angegebenen Maps.
<code>map:put</code>	Fügt einen neuen Eintrag an eine Map an oder überschreibt in einem vorhandenen Eintrag mit dem angegebenen Schlüssel den bisherigen Wert mit dem angegebenen Wert.
<code>map:remove</code>	Entfernt aus einer Map die Einträge mit den angegebenen Schlüsseln.
<code>map:size</code>	Liefert die Zahl der Einträge in einer Map.

7.20.9 Neue Elemente in XSLT 3.0

Im Folgenden finden Sie eine alphabetisch geordnete Liste der in XSLT 3.0 neu eingeführten Elemente mit den jeweils möglichen Attributen.

<xsl:accept>

Die Anweisung erlaubt es, die Sichtbarkeit einer Komponente aus einem verwendeten Paket zu bestimmen.

```
<xsl:accept
  component = "template" | "function" | "attribute-set" |
  "variable" | "mode" | "*"
  names = tokens
  visibility = "public" | "private" | "final" | "abstract" | "hidden" />
```

<xsl:accumulator>

Liefert einen Zähler, der beim Durchlauf einer Knotenmenge verwendet werden kann.

```
<xsl:accumulator
  name = eqname
  initial-value = expression
  as? = sequence-type
  streamable? = boolean >
  <!-- Inhalt: xsl:accumulator-rule+ -->
</xsl:accumulator>
```

<xsl:accumulator-rule>

Das Element ist das Kindelement von <xsl:accumulator> und legt dessen Zählweise fest.

```
<xsl:accumulator-rule
  match = pattern
  phase? = "start" | "end"
  select? = expression >
  <!-- Inhalt: Sequenzkonstruktor -->
</xsl:accumulator-rule>
```

<xsl:assert>

Das Element wird verwendet, um zu prüfen, ob der Wert des angegebenen Ausdrucks wahr ist.

```
<xsl:assert
  test = expression
  select? = expression
  error-code? = { eqname } >
  <!-- Inhalt: Sequenzkonstruktor -->
</xsl:assert>
```

<xsl:break>

Wird verwendet, um eine Schleife zu stoppen.

```
<xsl:break
  select? = expression >
  <!-- Inhalt: Sequenzkonstruktor -->
</xsl:break>
```

<xsl:catch>

Wird innerhalb eines <xsl:try>-Elements verwendet, um anzugeben, was im Falle eines Fehlers geschehen soll.

```
<xsl:catch
  errors? = tokens
  select? = expression >
  <!-- Inhalt: Sequenzkonstruktor -->
</xsl:catch>
```

<xsl:context-item>

Wird als Kind von <xsl:template> verwendet, um zu erklären, ob das betreffende Datenelement erforderlich ist oder nicht.

```
<xsl:context-item
  as? = item-type
  use? = "required" | "optional" | "absent" />
```

<xsl:evaluate>

Erlaubt die Auswertung eines als Zeichenfolge generierten XPath-Ausdrucks.

```
<xsl:evaluate
  xpath = expression
  as? = sequence-type
  base-uri? = { uri }
  with-params? = expression
  context-item? = expression
  namespace-context? = expression
  schema-aware? = { boolean } >
  <!-- Inhalt: (xsl:with-param | xsl:fallback)* -->
</xsl:evaluate>
```

<xsl:expose>

Dieses Element erlaubt es, die Sichtbarkeit der ausgewählten Komponenten in einem Paket festzulegen.

```
<xsl:expose
  component = "template" | "function" | "attribute-set" |
  "variable" | "mode" | "*"
  names = tokens
  visibility = "public" | "private" | "final" | "abstract" />
```

<xsl:fork>

Das Element erlaubt es, die Ergebnisse mehrerer Auswertungen in einer Sequenz zusammenzuführen.

```
<xsl:fork>
  <!-- Inhalt: (xsl:fallback*, ((xsl:sequence, xsl:fallback*)* |
  (xsl:for-each-group, xsl:fallback*))) -->
</xsl:fork>
```

<xsl:global-context-item>

Wird verwendet, um festzulegen, ob ein globales Kontextdatenelement erforderlich ist und, falls ja, was der benötigte Typ ist.

```
<xsl:global-context-item>
  as? = item-type
  use? = "required" | "optional" | "absent" />
</xsl:global-context-item>
```

<xsl:iterate>

Das Element erlaubt es, eine Sequenz mehrfach zu durchlaufen.

```
<xsl:iterate>
  as? = item-type
  use? = "required" | "optional" | "absent" />
</xsl:iterate>
```

<xsl:map>

Das Element liefert eine neue Map.

```
<xsl:map>
  <!-- Inhalt: Sequenzkonstruktor -->
</xsl:map>
```

<xsl:map-entry>

Das Element liefert einen Map-Eintrag mit einem Schlüssel und einem Wert dafür.

```
<xsl:map-entry
  key = expression
  select? = expression >
  <!-- Inhalt: Sequenzkonstruktor -->
</xsl:map-entry>
```

<xsl:merge>

Das Element erlaubt es, mehrere Input-Sequenzen zu einer Output-Sequenz zusammenzuführen.

```
<xsl:merge>
  <!-- Inhalt: (xsl:merge-source+, xsl:merge-action, xsl:fallback*) -->
</xsl:merge>
```

<xsl:merge-action>

Das Element ist Kind von <xsl:merge> und legt fest, was mit einem Schlüsselwert der Input-Sequenzen geschehen soll.

```
<xsl:merge-action>
  <!-- Inhalt: Sequenzkonstruktor -->
</xsl:merge-action>
```

<xsl:merge-key>

Legt den Merge-Schlüssel für ein Element innerhalb von <xsl:merge-source> fest.

```
<xsl:merge-key
  select? = expression
  lang? = { language }
  order? = { "ascending" | "descending" }
  collation? = { uri }
  case-order? = { "upper-first" | "lower-first" }
  data-type? = { "text" | "number" | eqname } >
  <!-- Inhalt: Sequenzkonstruktor -->
</xsl:merge-key>
```

<xsl:merge-source>

Das Element definiert eine oder mehrere Input-Sequenzen.

```
<xsl:merge-source
  name? = ncname
  for-each-item? = expression
  for-each-source? = expression
  select = expression
```

```

    streamable? = boolean
    use-accumulators? = tokens
    sort-before-merge? = boolean
    validation? = "strict" | "lax" | "preserve" | "strip"
    type? = eqname >
    <!-- Inhalt: xsl:merge-key+ -->
</xsl:merge-source>

```

<xsl:mode>

Das Element bestimmt die Aktion, die vorzunehmen ist, wenn es keine Matching-Template-Regel oder wenn es mehrere konkurrierende Regeln gibt.

```

<xsl:mode
  name? = eqname
  streamable? = boolean
  use-accumulators? = tokens
  on-no-match? = "deep-copy" | "shallow-copy" | "deep-skip" | "shallow-skip" |
    "text-only-copy" | "fail"
  on-multiple-match? = "use-last" | "fail"
  warning-on-no-match? = boolean
  warning-on-multiple-match? = boolean
  typed? = boolean | "strict" | "lax" | "unspecified"
  visibility? = "public" | "private" | "final" />

```

<xsl:next-iteration>

Setzt den Durchlauf einer Input-Sequenz mit dem nächsten Datenelement fort.

```

<xsl:next-iteration>
  <!-- Inhalt: (xsl:with-param*) -->
</xsl:next-iteration>

```

<xsl:on-completion>

Gibt an, was beim Durchlauf einer Input-Sequenz geschehen soll, wenn kein Datenelement mehr gefunden wird.

```

<xsl:on-completion
  select? = expression >
  <!-- Inhalt: Sequenzkonstruktor -->
</xsl:on-completion>

```

<xsl:on-empty>

Legt fest, was geschehen soll, wenn eine Input-Sequenz leer ist.

```
<xsl:on-empty
  select? = expression >
  <!-- Inhalt: Sequenzkonstruktor -->
</xsl:on-empty>
```

<xsl:on-non-empty>

Legt fest, was geschehen soll, wenn eine Input-Sequenz nicht leer ist.

```
<xsl:on-non-empty
  select? = expression >
  <!-- Inhalt: Sequenzkonstruktor -->
</xsl:on-non-empty>
```

<xsl:override>

Wird als Kind von <xsl:use-package> eingesetzt, um eine Komponente des verwendeten Pakets zu überschreiben.

```
<xsl:override>
  <!-- Inhalt: (xsl:template | xsl:function | xsl:variable | xsl:param |
  xsl:attribute-set)* -->
</xsl:override>
```

<xsl:package>

Das Element wird eingesetzt, um ein Bündel von Komponenten für die Verwendung in Stylesheets zur Verfügung zu stellen.

```
<xsl:package
  id? = id
  name? = uri
  package-version? = string
  version = decimal
  input-type-annotations? = "preserve" | "strip" | "unspecified"
  declared-modes? = boolean
  default-mode? = eqname | "#unnamed"
  default-validation? = "preserve" | "strip"
  default-collation? = uris
  extension-element-prefixes? = prefixes
  exclude-result-prefixes? = prefixes
  expand-text? = boolean
  use-when? = expression
  xpath-default-namespace? = uri >
  <!-- Inhalt: ((xsl:expose | declarations)*) -->
</xsl:package>
```


<xsl:source-document>

Das Element wird verwendet, um ein Input-Dokument per Streaming zu lesen und zu verarbeiten.

```
<xsl:source-document
  href = { uri }
  streamable? = boolean
  use-accumulators? = tokens
  validation? = "strict" | "lax" | "preserve" | "strip"
  type? = eqname >
  <!-- Inhalt: Sequenzkonstruktor -->
</xsl:source-document>
```

<xsl:try>

Das Element erlaubt das Abfangen von dynamischen Fehlern.

```
<xsl:try
  select? = expression
  rollback-output? = boolean >
  <!-- Inhalt: (Sequenzkonstruktor, xsl:catch, (xsl:catch | xsl:fallback)*) -->
</xsl:try>
```

<xsl:use-package>

Bestimmt über die Nutzung von Komponenten aus dem angegebenen Paket.

```
<xsl:use-package
  name = uri
  package-version? = string >
  <!-- Inhalt: (xsl:accept | xsl:override)* -->
</xsl:use-package>
```

<xsl:where-populated>

Legt die Schritte fest, die erfolgen sollen, wenn eine Input-Sequenz nicht leer ist.

```
<xsl:where-populated>
  <!-- Inhalt: Sequenzkonstruktor -->
</xsl:where-populated>
```