9 XML

9.1 | Anwendung und Grundbegriffe

XML steht für **eXtensible Markup Language** und bezeichnet ein Verfahren, um **Texte auszuzeichnen** und **Informationen zu kodieren**. Es ist entwickelt worden, um die Strukturen eines Dokuments kenntlich und damit für den Computer verarbeitbar zu machen, indem Kodierungen (›Auszeichnungen‹) in einen laufenden Text eingefügt werden. Mit XML können aber auch andere, multimediale Datenbestände organisiert werden, indem textliche oder numerische Informationen in eine Struktur eingefügt werden.

XML verwendet für die Kodierungen die gleichen Zeichen wie für den Text oder die zu beschreibenden Daten. Diese **Zeichendaten** stammen aus einem bestimmten Zeichenvorrat (s. Kap. 5), heutzutage meistens Unicode in der Form UTF-8. Das sorgt dafür, dass XML mit allen gängigen Editoren geschrieben, auf allen Plattformen und Betriebssystemen gelesen werden kann und deshalb als sehr zukunftssicher gilt. Die Kodierung von Information mit den gleichen Zeichen, die für den Text verwendet werden, und mit sprechenden Codes hat den Vorteil, dass die Textauszeichnung für Menschen ebenso wie für Maschinen lesbar ist.

Um zwischen Kodierungen und Text zu unterscheiden, legt XML einige Regeln fest. Typisch für XML sind die spitzen Klammern (< und >): Text innerhalb der spitzen Klammern ist Markup, außerhalb der eigentliche Text. Die mit den spitzen Klammern angezeigten Kodierungen heißen ***tag*** (engl. ›Etikett‹, ›Kennzeichner‹). Die *tags* bilden normalerweise korrespondierende Paare aus einem Anfangstag bzw. öffnenden *tag* (`<gedicht>`) und einem Endtag bzw. schließenden *tag*, das den gleichen Namen hat wie das Anfangstag, aber von einem Schrägstrich eingeleitet wird (`</gedicht>`). So kann man mit den Kodierungen Textbereiche markieren und ihnen Bedeutungen zuweisen. Im Beispiel 1 werden z. B. ›Krethi & Plethi‹ als Titel und ›Kurt Tucholsky‹ als Autor markiert oder **ausgezeichnet**.

Beispiel 1: Ein in XML kodiertes Gedicht (Auszug)

```
<?xml version="1.0" encoding="UTF-8"?>
<gedichtsammlung xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xmlns="http://gedichtforschung.org/gedichtsammlungen"
         xsi:SchemaLocation="http://gedichtforschung.org/gedichtsammlung.
xsd">
  <gedicht>
    <bibliografisch>
      <titel>Krethi &amp; Plethi</titel>
      <autor typ="normiert" identifikation="gnd_11862444X">
        <name><vorname>Kurt</vorname> <nachname>Tucholsky</nachname></name>
        <name typ="alias">Theobald Tiger</name>
        <!-- unter diesem Pseudonym veröffentlicht -->
      </autor>
```


F. Jannidis et al. (Hrsg.), *Digital Humanities*

```
      <quelle>Berliner Tageblatt, Nr. 487
        <datum normiert="1918-09-23">23. September 1918</datum>
      </quelle>
    </bibliografisch>
    <text sprache="deutsch">
      <strophe>
        <vers><name typ="person" identifikation="gnd_116996595">Vater
          <nachname>Liebert</nachname></name> hat eine Rede vom Stapel
          gelassen,</vers>
        <vers>in der er sagte, der <name typ="institution" identifikation=
        "gnd_35236-6"> Reichstag</name> täte ihm nicht mehr passen.</vers>
      </strophe>
      <strophe>
        <vers>Denn in diesen durchaus traurigen Verein</vers>
        <vers>kämen ja sogar <redewendung>Krethi und Plethi</redewendung>
        hinein.</vers>
      </strophe>
      <strophe>
        <vers>Ich weiß nun nicht genau, wer Krethi und Plethi sind;</vers>
        <vers>vielleicht meint er damit meinen Vater oder dein Enkelkind.
        </vers>
      </strophe>
      <strophe>
        <vers>Aber das weiß ich: die <name typ="ereignis" identifikation=
        "wikidata_Q28205"> Schlacht bei Warschau</name> und <name typ=
        "ereignis" identifikation="wikidata_Q1926157">in den Argonnen
        </name>,</vers>
        <vers>die haben Deutschlands Krethi und Plethi gewonnen.</vers>
      </strophe>
      <luecke grund="auswahl"/> <!-- weiterer Text fehlt hier -->
    </text>
  </gedicht>
  <gedicht>
    <bibliografisch><!--...--></bibliografisch>
    <text><!--...--></text>
  </gedicht>
</gedichtsammlung>
```

Da Textinhalte und XML-Codes wie z. B. *tags* mit dem gleichen Zeichenvorrat geschrieben werden, ist es notwendig, bestimmte Zeichen zu reservieren. Sie müssen dann durch eine sogenannte Zeichenentität (***character entity***) ersetzt werden, die die allgemeine Form `&Entitätsname;` hat. Damit ist neben der spitzen Klammer auch das `&` ein Signalzeichen mit einer besonderen Bedeutung. Für den normalen Gebrauch im Text und zur Vermeidung von Konflikten in bestimmten Kontexten sind sie deshalb ggf. zu ersetzen:

Zeichen	Entity	Bedeutung
<	<	less than
>	>	greater than
&	&	ampersand
"	"	quotation mark
'	'	apostrophe

Tab. 4 Character Entities

Die Einheit aus *tags* und allem, was zwischen den *tags* steht, heißt **Element**. Elemente können Textinhalt oder andere Elemente oder beides enthalten. Im Beispiel gibt es ein Element `vers`, das Text enthält, der teilweise wieder mit anderen tags (z. B. ›name‹) ausgezeichnet ist. Elementnamen berücksichtigen Groß- und Kleinschreibung (sind *case sensitive*): `titel` und `Titel` sind zwei verschiedene Elementtypen. Elementnamen müssen mit einem Buchstaben oder einem Unterstrich beginnen; sie dürfen Buchstaben, Zahlen, Bindestriche, Unterstriche und Punkte, aber keine Leerzeichen enthalten.

Wenn Elemente keinen Inhalt, also weder Textdaten noch andere Elemente enthalten, nennt man sie **leere Elemente**. Statt `<luecke></luecke>` wird im Beispiel deshalb die abgekürzte Schreibweise `<luecke/>` verwendet.

Elemente können mit **Attributen** um weitere Informationen ergänzt werden: `<text sprache="deutsch">`. Attribute sind Teil des Anfangstags und bestehen aus einem Attributnamen und einem in Anführungszeichen stehenden Attributwert, die mit einem Gleichheitszeichen verbunden sind. Ein Element kann viele verschiedene Attribute enthalten, die durch ein Leerzeichen getrennt werden. Zwei Attribute gleichen Namens darf es innerhalb eines Elements nicht geben. Attributnamen beginnen mit einem Buchstaben oder dem Unterstrich und unterscheiden zwischen Groß- und Kleinschreibung.

XML-Dokumente können **Kommentare** enthalten, die nur für den menschlichen Leser gedacht sind und von der Software ignoriert werden, die die Dokumente dem Regelwerk von XML entsprechend einlesen und vorverarbeiten (parsen). Kommentare haben eine spezielle Kodierungsform für Anfang und Ende: `<!-- Kommentar -->`

9.2 | Grundstrukturen

Schachtelung: XML-Elemente können andere Elemente enthalten. Umgekehrt wird daraus die wichtigste Regel von XML: Alle Elemente müssen vollständig in einem anderen Element enthalten sein. Diese Schachtelung führt dazu, dass:

- die Struktur der Kodierungen strikt hierarchisch ist.
- alles in einem obersten Element enthalten ist, das man Wurzelelement (*root*) nennt.

Bei dieser Hierarchie der sich auffächernden Elemente spricht man von einer **Baumstruktur**.

Die Elemente eines XML-Dokuments werden **Knoten** (*nodes*) genannt. In der hierarchischen Ordnung spricht man von über- und untergeordneten Elementen, die in einer sogenannten **Eltern-Kind-Relation** stehen. Man sagt, dass `gedichtsamm-`

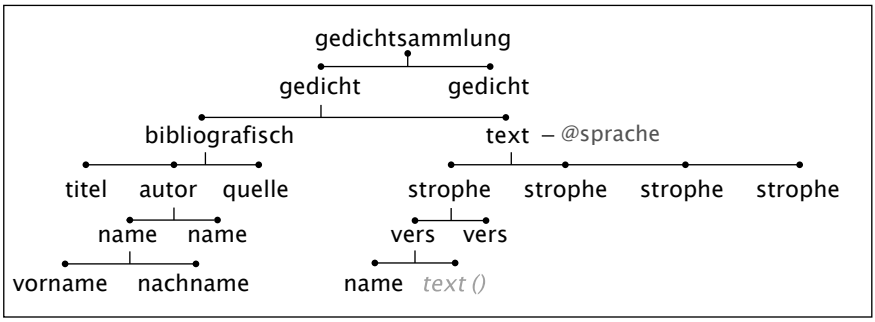

Abb. 27 Schematische Darstellung der (Baum-)Struktur der XML-Kodierung in Beispiel 1

`lung` ein Elter (*parent*) von `gedicht` ist und dass `strophe` ein Kind (*child*) von `text` ist. Auch sagt man, dass sich `text` und `vers` wie Vorfahre (*ancestor*) und Nachkomme (descendant) verhalten, wobei diese Relation beliebig viele Ebenen überspringen kann. Zugleich haben XML-Dokumente eine **sequentielle Ordnung**: Elemente stehen vor (*preceding*) und hinter (*following*) anderen Elementen. So sind z. B. alle `strophe`-Elemente nachfolgend (*following*) zum `bibliografisch`-Element. Haben Elemente das gleiche Elternelement, dann nennt man sie **Geschwister** (*siblings*). Alle Strophen eines Textes sind Geschwister. Die Strophen verschiedener Texte sind aber nur die Nachkommen des gemeinsamen Vorfahren `gedichtsammlung`.

In XML dürfen sich Elemente **nicht überlappen**: Jedes Element, das in einem anderen enthalten ist, muss geschlossen werden, bevor das Elternelement endet. Eine Struktur `<a> ... <b> ... </a> ... </b>` überlappt (overlap) und ist nicht erlaubt, weil hier nicht mehr klar ist, was eigentlich in was enthalten ist und damit keine klare Hierarchie mehr besteht.

Die Struktur eines XML-Dokuments besteht im Wesentlichen aus einem **Baum**. Es hat ein Wurzelelement, das alle weiteren Elemente und Inhalte einschließt. Zusätzlich gibt es aber noch den sogenannten **Prolog**, der vor dem Starttag des Wurzelelements steht. Diese enthält mindestens die XML-Deklaration (mit Angaben zur XML-Version und zur Zeichencodierung, Zeile 1 in Beispiel 1) und weitere optionale Angaben, wie z. B. *processing instructions* dazu, welchem Regelwerk (Schema) das Dokument entsprechen oder mit welchen Verarbeitungsanweisungen es transformiert werden soll.

Programme, die XML-Daten einlesen und prüfen, heißen ›Parser‹. Werden die grundlegenden Regeln eingehalten, dann kann der Parser aus dem Dokument die Baumstruktur der Daten konstruieren und für die weitere Arbeit damit zur Verfügung stellen. Dokumente, die die **grundlegenden Regeln einhalten**, nennt man **wohlgeformt** (*well formed*). Kann ein Parser außerdem feststellen, dass ein Dokument die in einem **Schema** (s. u.) festgelegten Regeln der Verwendung von Elementen und Attributen einhält, dann gilt es außerdem als **gültig** bzw. valide (*valid*). Die kontinuierliche Validierung stellt sicher, dass auch über viele Dokumente oder Bearbeiter hinweg die gleichen Regeln zur Beschreibung und Strukturierung von Textdaten eingehalten werden.

9.3 | Konzepte und Datenmodell

Die Idee, Text mit Markierungen zu versehen, die zusätzliche Informationen für die weitere Verarbeitung und Darstellung geben, stammt aus der Welt des Drucksatzes: Strukturell bestimmte Textpassagen (z. B. Überschriften) wurden im Manuskript gekennzeichnet, damit im Druck dann besondere Schrifteigenschaften wie fett gesetzte Buchstaben oder Unterstreichungen verwendet wurden. Diese Konzepte wurden schon in den 1960er Jahren auf die Textverarbeitung mit dem Computer übertragen. Textauszeichnungssprachen beruhen wie XML meist darauf, dass Kodierungen mit demselben Zeichensatz wie der eigentliche Text in diesen eingefügt werden, verwenden jedoch eine unterschiedliche Syntax und verfolgen unterschiedliche Ziele. Textauszeichnungssprachen wie LaTeX, MarkDown oder das Wiki-Markup stellen die Verarbeitung und die Darstellung des Textes in den Vordergrund. Textauszeichnung für Verarbeitung des Textes wird ›**prozedurale Auszeichnung**‹ (*procedural markup*), Textauszeichnung für die Darstellung ›präsentationale Auszeichnung‹ (*presentational markup*) genannt. Einen anderen Ansatz verfolgt die ›**deskriptive Auszeichnung**‹ (*descriptive markup*). Sie geht davon aus, dass sich die Bedeutung von Textdaten von ihrer Darstellung trennen lässt und dass die Auszeichnung dazu dient, die Bedeutung von Textabschnitten zu beschreiben, die unabhängig von ihrer Darstellung kodiert werden kann.

Das Konzept deskriptiver Auszeichnungssprachen ist in den 1970er Jahren u. a. von Charles M. Goldfarb eingeführt worden. Es führte zur Entwicklung der ›Standardized General Markup Language‹ (SGML), die 1986 als ISO 8879 normiert wurde. XML ist eine Weiterentwicklung von SGML, die das komplexe Regelwerk so reduziert, dass nach XML kodierte Daten leichter von Programmen verarbeitet werden können. Dies liegt vor allem am Verzicht auf Mechanismen zur Beschreibung von impliziten (schließende Tags sind nicht notwendig) und von überlappenden Strukturen (CONCUR Funktion).

Lesbarkeit für Mensch und Maschine: Zu den Grundprinzipien von SGML und damit auch von XML gehört, dass der in den Text eingefügte Code nicht nur von Maschinen sondern auch von Menschen lesbar sein sollte. Dieses Prinzip schließt binäre Kodierungen aus, da sie nicht unmittelbar lesbar wären. Zur guten Praxis von XML gehört auch, dass die Namen von Elementen und Attributen sprechend sind. Von Programmierern wird diese ›Weitschweifigkeit‹ als Problem wahrgenommen, weil sie z. B. die zu übertragende Datenmenge erhöht. Aus Sicht der digitalen Geisteswissenschaften ist diese Weitschweifigkeit eine vorteilhafte Eigenschaft, denn sie erlaubt es, mit dem in XML kodierten Text in einer Art und Weise umzugehen, die dem Gebrauch natürlicher Sprache sehr ähnlich ist.

XML in der Version 1.0 ist 1998 vom World Wide Web Consortium (W3C) als Standard (*recommendation*) verabschiedet worden und im Grunde heute noch gültig. Selbst die fünfte Auflage von 2008 hat nur Unwesentliches geändert. Das gleiche gilt für die XML-Version 1.1 von 2004, die nur den Umgang mit Unicode betrifft, und das 2001 neu eingeführte ›XML Infoset‹, das gemeinsame konzeptionelle und definitorische Grundlage für die gesamte XML-Sprachfamilie festlegt.

XML ist eine Auszeichnungssprache, die sonst z. B. durch typografische oder Layout-Strukturen oder durch menschliches Kontextwissen implizit gegebene Informationen explizit macht. Was in einem Dokument zentriert und fett gedruckt ist, kann als Element vom Typ `Überschrift` identifiziert und markiert werden;

›Vater Liebert‹ (aus unserem Beispiel 1) kann als Bezeichnung für eine bestimmte `Person` erkannt und ausgezeichnet werden, um zusätzliches Wissen an den Text anzulagern und die Stelle z. B. als Registerbegriff zu verwenden; die bibliographische Angabe ›23. September 1918‹ kann als `Datum` beschrieben und mit einer formalisierten Zusatzangabe ›1918–09–23‹ versehen werden, um sie mit dem Computer leichter berechenbar zu machen. Ebenso können Festlegungen getroffen werden, wie dass `Strophen Verse` enthalten oder `Personennamen` in `Vor-` und `Nachnamen` gegliedert sind. XML erlaubt es so, genau festgelegte Strukturen für die Beschreibung von Texten je nach Dokumentart oder Wissensbereich zu schaffen. Man kann mit XML Gedichte ebenso systematisch kodieren wie Banküberweisungen oder das Wissen zu den Beständen eines Ersatzteillagers. XML ist deshalb eine Technologie, die es ermöglicht, Sprachen zu erschaffen, mit denen wir höchst strukturiert und sehr explizit mit Informationen umgehen können. Gleichzeitig können wir mit XML nah an unserer natürlichen Verwendung von Sprache bleiben, und mit eingeführten Begriffen über die Objekte sprechen, die vom Computer verarbeitet werden sollen.

XML selbst ist strenggenommen gar keine Auszeichnungssprache, sondern eine **Metasprache**, weil sie nur ein allgemeines Regelwerk zur Textauszeichnung liefert, die Festlegung des eigentlichen Vokabulars und seiner Verwendung aber nicht vorwegnimmt. Das ›eXtensible‹ in XML steht dafür, dass man seine eigenen Beschreibungsmodelle und Konstrukte kreieren kann. XML sagt ›erfinde Deine eigenen Elemente‹. XML ist die Grundlage vieler konkreter Sprachen bzw. Standards zur Beschreibung von Informationsdomänen, die **Anwendungen** von XML sind. Dies sind z. B. wichtige Sprachen in den digitalen Geisteswissenschaften, wie die Regeln der Text Encoding Initiative (TEI), Metadatenstandards wie Dublin Core (DC), METS (Metadata Encoding and Transmission Standard) oder LIDO (Lightweight Information Describing Objects). Aber auch in vielen anderen Domänen wird XML zur Beschreibung und zum Austausch von Daten verwendet. Dies reicht von Überweisungen zwischen Banken bis hin zu Büro-Anwendungen wie Microsoft Office-Programmen oder OpenOffice. Wenn man einmal hinter die Kulissen schaut, entdeckt man heute in fast allen Bereichen der Datenhaltung und des Datenaustauschs XML.

Universalität von XML: Die Benutzung einfacher Zeichendaten, die Plattformunabhängigkeit, der Charakter als offener und voll dokumentierter W3C-Standard, die Einfachheit, die Menschenlesbarkeit, der breite Einsatzbereich und die weite Verbreitung sind Argumente für die Verwendung von XML. In den Geisteswissenschaften spielen aber noch weitere Eigenschaften eine Rolle, die XML auch von anderen Technologien zur Organisation von Daten unterscheidet. Hier ist zunächst der merkwürdige Doppelcharakter von XML als hierarchischem Baum *und* als sequentiellem Dokument zu bedenken. Während Auszeichnungssprachen zunächst für die Kodierung von Texten und Dokumenten geschaffen worden sind, haben sie sich inzwischen zu einem allgemeinen Austauschformat für alle möglichen Daten und Datenstrukturen entwickelt. Damit gehen zwei grundverschiedene Verwendungsweisen einher.

- **Dokumentorientiertes Markup:** Auf der einen Seite spricht man von dokumentorientiertem Markup, wenn man bestehende Textdaten allmählich auszeichnet, auf ihrer Grundlage ein Modell entwickelt und damit zu komplexen, tief gestaffelten Strukturen kommt, die häufig Elemente mit **mixed content** (Elemente *und* Textdaten) aufweisen.

- **Datenzentriertes Markup** hingegen speichert in eher klaren und strikten Strukturen und weniger tief hierarchisierten Bäumen kontrollierte Werte. Hier wird XML oft als Austauschformat für Anwendungen verwendet, die selbst z. B. ein relationales Datenbankmodell implementieren.

Beide Sichtweisen gehen von unterschiedlichen Startpunkten aus. Die eigentliche Intention bei XML war ursprünglich die Repräsentation, Explikation und Annotation bestehender Daten. Das Verhältnis des XML-Modells zur Welt ist hier deshalb eher **deskriptiv** (beschreibend). XML kann aber auch verwendet werden, um Daten in ein bestimmtes Modell der Welt einzufüllen – es wird dann eher **präskriptiv** (vorschreibend) verwendet.

In der Praxis gehen beide Prinzipien Hand in Hand: Modelle und Sprachen werden als Beschreibung der Welt und bestehender Objekte entwickelt, dienen dann aber auch als kodifizierter Konsens des Sprechens über Phänomene, als Vorgabe und als Kontrollmechanismus für erhobene Daten. Seine Stärke hat XML aber immer dort, wo es um die Verwaltung komplexer, lose strukturierter Wissensbestände mit dem Bedarf an flexibler Modellierung und der Verwaltung von *mixed content*-Phänomenen geht. Gerade das letztere ist ein Alleinstellungsmerkmal im Vergleich mit anderen Datenmodellen wie dem relationalen Datenbankmodell oder JSON (JavaScript Object Notation) (s. u.).

Das **relationale Datenbankmodell** (s. Kap 8) basiert auf der relationalen Algebra und verfügt damit über ein abgesichertes mathematisches Modell. Mit den Modellierungsprinzipien der Normalisierung führt es zu sehr klaren, rationalen, redundanzarmen Strukturen. Gute Kontrollmöglichkeiten bei den Datentypen und die Ausgereiftheit, die einfache Benutzbarkeit und die gute Performanz von Datenbankmanagementsystemen haben diesen Ansatz so sehr zum Normalfall gemacht, dass in der Alltagssprache fast immer ›relationale Datenbanken‹ gemeint sind, wenn ›Datenbank‹ gesagt wird. Das Konzept stößt aber an seine Grenzen, wenn es um dokumentorientierte oder Textdaten geht, die mit einem komplexen oder semistrukturierten Modell beschrieben werden sollen. Die Verwendung von XML ist vorzuziehen, wenn die Sequentialität des Textes, komplexe Hierarchien, eher lockere Beschreibungsmodelle und *mixed content*-Phänomene wichtige Modellaspekte sind, die als strukturierte Datenbasis verwaltet und algorithmisch verarbeitet werden sollen.

In letzter Zeit erfreut sich die ›JavaScript Object Notation‹ (JSON) zunehmender Beliebtheit. Dieses Format ist ebenfalls textbasiert, menschenlesbar und prinzipiell hierarchisch organisiert. Im Gegensatz zu XML gilt das Format als leichtgewichtiger und weniger weitschweifig, und es lässt sich einfach mit den aktuellen Webtechnologien verarbeiten, weil die Strukturen von JSON direkt in JavaScript-Objekte übersetzt werden. Die Kontrollmöglichkeiten durch Datentypen und festgelegte Schemata stehen derzeit noch am Anfang der Entwicklung. Von der Ausdrucksmächtigkeit her kann man zugespitzt sagen: ›JSON ist wie XML ohne *mixed content*‹.

XML selbst ist also prinzipiell ausdrucksmächtiger als das relationale Datenmodell oder JSON, dadurch aber unter Umständen auch komplexer. XML findet seine Grenze der Ausdrucksmächtigkeit vor allem an Phänomenen überlappender Hierarchien, für die es zwar mit leeren Elementen und dem sogenannten stand-off-markup Lösungsansätze gibt; diese führen aber zu einer noch höheren Komplexität. In der Praxis versucht man deshalb, überlappende Hierarchien zu vermeiden. Ist das nicht möglich, muss man die Komplexität in Kauf nehmen oder auf andere Auszeichnungssprachen ausweichen, in denen *overlap* möglich ist.

9.4 | Modelle und Schemata

XML legt nicht fest, welche Elemente und Elementnamen verwendet werden. Mit XML kann man deshalb frei gewählte Strukturen von Texten und Daten kodieren. Es gibt aber ein paar praktische Hinweise, wie man seine eigenen Elementnamen und Strukturen bildet: Elemente dienen der Abstraktion von Textphänomenen oder der Beschreibung von Strukturen. Man definiert deshalb Elemente für Phänomene, die voraussichtlich häufiger vorkommen bzw. auf mehrere Texte oder Dokumente anwendbar sind. Elementnamen sollten sprechend und verständlich aber kurz sein. Bindestriche und Punkte sind in Elementnamen unüblich. Es gibt Stile für Elementnamen wie z. B. ›CamelCase‹ (wenn sich ein Elementname aus mehreren Wörtern zusammensetzt, werden diese mit Großbuchstaben voneinander getrennt). Man ist hier frei, sollte aber im Vorgehen konsistent sein.

Man kann viele Textphänomene oder Strukturen mit Elementen *oder* mit Attributen ausdrücken, z. B. `<text><autor>Kurt Tucholsky</autor>...</text>` und `<text autor="Kurt Tucholsky">...</text>`. Man bevorzugt **Elemente**, wenn man komplexe Inhalte beschreiben will, die vielleicht später weiter ausgezeichnet oder annotiert werden sollen. Man verwendet **Attribute** für eher technische Informationen über ein Element oder für einfache Inhalte, insbesondere wenn sie kontrolliert und systematisch gebraucht werden, z. B. wenn man auf eine Identifikationsnummer für eine Person verweisen will: `<autor gnd="11862444X">Panter, Peter</autor>` oder `<zahl wert="141">CXLI</zahl>`. Mehrere Inhalte in einer Attributkategorie sollten keinesfalls mit mehreren Attributen (`autor="Kafka, Franz" autor1="Brod, Max"`), sondern als wiederholte Unterelemente (`<autor>Kafka, Franz</autor><autor>Brod, Max</autor>`) ausgedrückt werden.

Strukturlogik: Mit Hilfe von **Schemata** kann man Elemente und Attribute definieren sowie ihre Verschachtelung und ihre Inhaltstypen festlegen. Man kann mit einem Schema also eine bestimmte Logik der Text- und Datenstrukturen festlegen und so ein Modell kodifizieren (s. Kap. 2.1 zu »Datenmodellierung«). Ein Schema erlaubt es damit einerseits, bei der Auszeichnung von Text bzw. der Erfassung von Daten gemeinsame Regeln zu verwenden, und damit sicherzustellen, dass bestimmte Informationen auch erfasst werden, oder bestimmte Strukturen ausgeschlossen sind. Andererseits unterstützt ein Schema die Weiterverarbeitung und den Austausch von Daten, indem sie die Funktion und das Verhalten von Elementen beschreiben (im Beispiel: `titel` und `autor` sind Teil der `bibliografischen` Beschreibung eines `gedichts`).

Wenn ein XML-Dokument den Regeln eines Schemas entspricht, wird es, wie gesagt, als ›valide‹ gegen das Schema bezeichnet. Die Validität gegen ein Schema kann von Software (den oben erwähnten XML-Parsern) überprüft werden. Mit einer solchen Prüfung wird sichergestellt, dass Dokumente einem gemeinsamen Modell entsprechen. Es ist aber damit noch nichts darüber ausgesagt, ob die Kodierung auch inhaltlich richtig ist oder was die Kodierung eigentlich meint.

Um Modelle in XML zu beschreiben, gibt es verschiedene Schemasprachen, die sich vor allem in ihrer Syntax, teilweise aber auch in ihrer Ausdrucksmächtigkeit unterscheiden: Mit manchen Schemasprachen kann man z. B. genauer festlegen, welche Datentypen ein Element enthält. Für die Beschreibung von Schemata für XML-Daten sind die folgenden vier besonders erwähnenswert:

- **DTD** (**D**ocument **T**ype **D**efinition): die älteste Schemasprache, die zwar leicht erlernbar und gut lesbar ist, aber nur noch selten verwendet wird, weil sie nicht der XML-Syntax folgt und etwas weniger ausdrucksmächtig als andere Sprachen ist.
- **XSD** (**X**ML **S**chema **D**efinition): die vom W3C entwickelte Schemasprache, die selbst in XML formuliert ist. Sie ist sehr ausdrucksmächtig, für den menschlichen Leser aber relativ unübersichtlich.
- **RNG** (**R**elax **NG** = **Re**gular **La**nguage Description for **X**ML **N**ew **G**eneration): Eine Schemasprache die zwei Arten von Syntax anbietet: eine nach den Regeln von XML wohlgeformte und eine kompaktere Syntax; etwas weniger ausdrucksmächtig als XSD.
- **Schematron:** Eine in XML ausgedrückte Sprache zur Validierung von XML-Dokumenten, die nicht eine Struktur definiert, sondern nur wann ein XML-Element einer Struktur widerspricht. Einzelne Elemente der Regeldefinitionen sind in XSD 1.1 übernommen worden; wenig verbreitet.

Ein Schema für unser Beispiel könnte in XSD ungefähr so aussehen:

Beispiel 2: Ausschnitt aus einem XML Schema

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDe-
fault="qualified" targetNamespace="http://gedichtforschung.org/gedicht-
sammlungen" xmlns="http://gedichtforschung.org/gedichtsammlungen">
<!-- ... -->
  <xs:element name="bibliografisch">
    <xs:annotation>
      <xs:documentation>Das Element enthält die bibliografischen Angaben
      des Gedichts.</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:sequence minOccurs="0">
        <xs:element ref="titel"/>
        <xs:element ref="autor"/>
        <xs:element ref="quelle"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="titel" type="xs:string"/>
  <xs:element name="autor">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" ref="name"/>
      </xs:sequence>
      <xs:attribute name="identifikation" use="required" type="xs:NCName"/>
      <xs:attribute name="typ" use="required" type="xs:NCName"/>
    </xs:complexType>
  </xs:element>
<!-- ... -->
<xs:schema>
```

Das Schema im Beispiel 2 enthält eine Liste von Elementdefinitionen (`<xs:element>`), zu denen jeweils der Name definiert ist (Attribut `name`). Der Inhalt jedes Elements ist über den Datentyp (Attribut `type`), oder ein komplexeres Inhaltsmodell `<xs:complexType>` für Kindelemente und *mixed content* definiert. Das Beispiel verwendet die einfachen Datentypen `xs:string` und `xs:date`. Diese Typen sind vom W3C definiert und stehen für einfachen Text (`xs:string`) bzw. eine Datumsangabe im Schema JJJJ-MM-TT (`xs:date`).

Weitere häufiger gebrauchte Datentypen könnten sein `xs:integer` für ganze Zahlen (positive wie negative) und `xs:double` für Kommazahlen. `<xs:complexType>` steht für eine Struktur, die mehr als nur einfachen Text oder Zahlen enthält. Mit dem Attribut `mixed` wird angegeben, ob das Element sowohl Text als auch Elemente enthalten kann. Bei der Auszeichnung von Texten ist das der Normalfall. Die Kindelemente können nun als Sequenz von Elementen (`<xs:sequence>`) oder als ein Element von vielen (`<xs:choice>`) definiert sein. In der Sequenz ist die Reihenfolge des Vorkommens der Elemente festgelegt. Die Attribute `minOccurs` und `maxOccurs` in der Definition des `<xs:complexType>` geben an, ob das Kindelement vorkommen muss und wie häufig es vorkommen kann. Die Schemadefinition `minOccurs="0"` sagt also, dass ein Element nicht vorkommen muss, aber kann, während `minOccurs="1"` sagt, dass ein Element vorkommen muss (nämlich mindestens einmal). Die Angabe `maxOccurs="unbounded"` sagt aus, dass ein Element beliebig häufig vorkommen kann, während `maxOccurs="1"` festlegt, dass ein Element nur genau einmal vorkommen darf.

Die Definition des `<xs:complexType>` enthält auch die Liste der möglichen Attribute für ein Element (`<xs:attribute>`). Ein Attribut, dass in jedem Fall vergeben werden muss, wird mit `use="required"` angegeben. Die Definition kann Elemente und Attribute explizit definieren oder nur auf sie verweisen (Attribut `ref`), was es erlaubt, Elemente und Attribute allgemein zu definieren und an verschiedenen Stellen zu verwenden. Im Bereich `<xs:annotation>` werden Anmerkungen zur Elementdefinition gegeben, wie z. B. in `<xs:documentation>` eine Beschreibung der Funktion des Elements.

Die Verwendung von Elementnamen hängt an dem Wissensbereich (der **Domäne**), um die es geht: Das obige Bespiel versteht unter ›Text‹ den Text eines Gedichts, der aus Versen besteht. In anderen Gattungen muß Text aber nicht aus Versen bestehen. Eine ›Mutter‹ ist in der Welt der Werkzeuge etwas anderes als in einer Datenbank zu Familienverhältnissen. Um diesen gebundenen Gebrauch von Elementnamen klar zu machen, gibt es das Konzept des sogenannten **Namensraums** (*namespace*). In unserem Beispiel 1 wird im Wurzelelement angegeben, dass das XML-Dokument den Regeln des XML-Schemas unter http://gedichtforschung.org/gedichtsammlung.xsd folgt (Attribut `xsi:SchemaLocation`) und die Elemente damit dem Namensraum `http://gedichtforschung.org/gedichtsammlungen` angehören (Attribut `xmlns`). In einem XML-Dokument können grundsätzlich Elemente aus mehreren Namensräumen vermischt werden, sofern sie entsprechend deklariert sind: `<familie:mutter>Conny</familie:mutter> gab ihm eine <werkzeug:mutter>12er</werkzeug:mutter>.`

Namensräume werden normalerweise im Wurzelelement eines XML-Dokuments deklariert und mit Hilfe von Attributen abgekürzt: `xmlns:tei="http://www.tei-c.org/ns/1.0"` erlaubt es so, im Dokument Elementnamen mit dem Präfix `tei:` zu versehen. Ein solches Präfix legt fest, dass das Element aus dem Namensraum der TEI stammt. Der Standardnamensraum eines XML-Dokuments wird mit

dem Attribut `xmlns` festgelegt. Elementnamen ohne Präfix stammen dann aus diesem Namensraum. Namensräume richtig zu bezeichnen, ist insbesondere bei der Umwandlung von XML zu HTML mit XSLT notwendig, denn dort werden sowohl Elemente aus dem Schema des Ausgangsdokuments als auch Elemente aus HTML und schließlich noch Elemente aus dem XML-Schema für die Transformation (XSLT) verwendet.

Auszeichnungssprachen: Konkrete Auszeichnungssprachen, die durch Schemata kodifiziert und dann zur Auszeichnung von Texten oder zur Beschreibung von Daten eingesetzt werden, sind Anwendungen von XML. Es gibt sehr viele auf XML basierende oder XML als Metagrammatik oder Syntax benutzende Sprachen. Zu den bekanntesten gehören die TEI (Text Encoding Initiative) für die Kodierung von Texten, XHTML als XML-konforme Variante *der* Sprache für Webseiten, DocBook für die Erstellung von Dokumentationen, MathML für die Beschreibung mathematischer Formeln oder SVG (Scalable Vector Graphics) für die Beschreibung von Vektorgrafiken.

9.5 | XPath

```
/gedichtsammlung/gedicht/bibliografisch/titel
```

Dies ist ein XPath-Ausdruck. Im Eingangsbeispiel werden damit alle Titel der Gedichtsammlung geliefert. Es ist ein formaler Ausdruck für die Anordnung »gib mir alle Elemente, die diese Bedingung erfüllen: es gibt ein Element `gedichtsammlung`, das ein Kindelement `gedicht` hat, das ein Kindelement `bibliografisch` hat, das ein Kindelement `titel` hat«.

XML beschreibt strukturierte Informationen. Um mit XML etwas zu tun, braucht man zunächst XPath, die Pfadsprache zur Navigation in XML-Dokumenten und zur Adressierung, Selektion, Vorverarbeitung und Rückgabe von verschiedenen Informationen. Weitere XML-Standards benutzen XPath, um in XML-Daten etwa bestimmte Stellen anzusteuern: XSLT als Transformationssprache oder XQuery als Abfragesprache im Zusammenspiel mit XML-Datenbanken. Man kann XPath aber auch für sich nutzen, um z. B. Daten abzufragen und zu analysieren.

XPath-Ausdrücke, manchmal auch Lokalisierungspfade genannt, können sehr komplex werden. Sie bestehen aber zunächst einfach aus **Lokalisierungsschritten**, die mit dem rechtsgeneigten Schrägstrich (*slash* ›/‹) verkettet werden: Lokalisierungsschritt/Lokalisierungsschritt/Lokalisierungsschritt. Dabei ist zu beachten, dass XPath-Ausdrücke von der Wurzel des Baums oder einer bestimmten Stelle ausgehen können. Die Stelle, an der man sich in einer komplexeren Anwendung gerade befindet, wird **Kontext** genannt. XPath beschreibt also **absolute** oder **relative Pfade**.

Die einzelnen Lokalisierungsschritte bestehen aus bis zu drei Elementen: einem Achsenbezeichner (*Achse::*), einem Knotentest und einem Prädikat. Die **Achse** bezeichnet die Richtung, in der nach einem Knoten gesucht werden soll. Sie wird vor einem doppelten Doppelpunkt angegeben (z. B. `ancestor::`). Es gibt zunächst zwei Grunddimensionen: vertikale Achsen, mit denen im Baum ab- oder aufwärts navigiert wird und horizontale Achsen, mit denen im sequentiell verstandenen Dokument vorwärts oder rückwärts navigiert wird. Häufig benutzte Achsen heißen:

Achse	Kurz	Beschreibung
self	.	Der aktuelle Kontextknoten
child	(Nichts)	Kinder des Kontextknotens
parent	..	Eltern des Kontextknotens
descendant	//	Nachkommen in beliebiger Tiefe des Baumes
ancestor		Vorfahren in beliebiger Höhe des Baumes
following		In der Dokumentreihenfolge nach dem Kontextknoten kommende Knoten
preceding		In der Dokumentreihenfolge vor dem Kontextknoten kommende Knoten
attribute	@	Attributknoten des Kontextknotens

Tab. 5 XPath-Achsen

Der Ausdruck

```
child::gedichtsammlung/descendant::autor/attribute::identifikation
```

ist ein XPath-Ausdruck, der zunächst zu allen `name`-Elementen in `gedichtsammlung` geht, um das Attribut `identifikation` im Elternelement auszulesen, egal wie das Elternelement heißt. Das Sternchen (*Asterisk*) ist ein Platzhalter (*Wildcard*) für einen beliebigen Elementnamen. In der Regel benutzt man aber eine abgekürzte Schreibweise. Der obige Ausdruck ist äquivalent zu:

```
./gedichtsammlung//autor/@identifikation
```

Prädikate in XPath-Ausdrücken enthalten Bedingungen. Diese werden in eckigen Klammern `[]` angegeben. Zusätzlich stehen etliche **Funktionen** zur Verfügung, die grundsätzlich die Form `funktionsname()` haben. In den runden Klammern kann der Funktion etwas übergeben werden: Zahlen, Zeichenketten oder auch das Ergebnis von XPath-Ausdrücken. Manche Funktionen erwarten allerdings keine Übergabe. Außerdem kann man mit XPath auch *rechnen*, also mathematische Operationen durchführen.

```
count(//gedicht[position()='1']//strophe)
```

oder kürzer

```
count(//gedicht[1]//strophe)
```

Das heißt auf Deutsch: »Zähle die Strophen im ersten Gedicht«. Oder etwas ausführlicher: Übergib der Funktion `count()` die Ergebnismenge der Navigation zu allen Nachkommen `strophe` des Elements `gedicht`, das die Bedingung erfüllt, dass dieses Elements in seiner Geschwisterschar die erste Position einnimmt. Die Rückgabe lautet für unser Beispiel »4« und ist vom Datentyp ›Zahl‹. XPath-Ausdrücke und Funktionen können grundsätzlich verschiedene Arten von Rückgaben haben: einzelne Elemente, Element-Sets, Sequenzen (geordnete Reihen von Dingen wie Strings oder Elementen etc.), Zahlen, Zeichenketten oder Wahrheitswerte (*Booleans*).

```
count(//gedicht[1]//strophe) > 3
not(//gedicht//luecke/text())
```

Das sind zwei Ausdrücke, die beide den Wahrheitswert ›true‹ zurückliefern. Gefragt war hier »hat das erste Gedicht mehr als drei Strophen?« und »es gibt doch wohl keine Gedichte, bei denen das Element `luecke` doch Textzeichen enthält?«

Der XPath-Standard 1.0 wurde 1999 vom W3C verabschiedet, mit Version 2.0 sind 2007 viele neue Funktionen und einige neue Konzepte hinzugekommen. Kleinere Änderungen sind bei Version 3.0 hinzugekommen, die 2014 eine W3C *recommendation* wurde. Es empfiehlt sich, in der Praxis mindestens XPath 2.0 zu verwenden, auch wenn manche Technologien oder Programmiersprachen(-Bibliotheken) auf dem Stand von 1.0 stehengeblieben sind.

Es gibt inzwischen über 130 Funktionen in XPath. Zu den häufig gebrauchten gehören z. B.

Funktionsname (Übergabe)	Rückgabe	Beschreibung
`position()`	Zahl	Position eines Elements in seinem Kontext; abgekürzte Syntax [n]
`not(XPath)`	Boolean	Kehrt den Wahrheitswert des XPath-Ausdrucks um
`count(XPath)`	Zahl	Anzahl der Knoten in einer Knotenmenge
`contains(string, string)`	Boolean	Enthält die erste Zeichenkette (die der Rückgabewert eines XPath-Ausdrucks sein kann) die zweite Zeichenkette?
`matches(string, string)`	Boolean	Testet, ob der reguläre Ausdruck in der zweiten Zeichenkette auf die erste Zeichenkette passt.
`substring(string, number, number?)`	String	Extrahiert aus einer Zeichenkette einen Teil ab der zuerst genannten Position, in der Länge der zweiten Zahl.
`name()` bzw. `local-name()`	String	Name des aktuell angesteuerten Elements oder Attributs
`distinct-values()`	Sequenz	Liefert aus einer gegebenen Menge nur die Menge unterschiedlicher Dinge zurück.
`string(object)`	String	Verwandelt etwas explizit in eine Zeichenkette.
`number(object)`	Zahl	Verwandelt etwas explizit in eine Zahl.

Tab. 6 XPath Funktionen

XPath ist eine formale Sprache. Man kann, wie wir oben gesehen haben, zwischen normaler (z. B. deutscher) Sprache und XPath hin und her übersetzen, wobei dies zugleich ein Prozess fortschreitender Formalisierung ist:

1. »Wie alt ist das erste Gedicht?« wird formalisiert zu …
2. »Bilde die Differenz zwischen 2016 und dem bei den bibliografischen Angaben zu findenden Jahr der Quelle«. Dies wiederum bedeutet auf XPath-Pseudocode …
3. `2016-`(Jahr der Quelle des ersten Gedichts), oder als etwas präziserer Pseudocode …
4. `2016-` (Jahr der Quelle von `//gedicht[1]`), oder als etwas präziserer Pseudocode …
5. `2016-(//gedicht[1]/bibliografisch/quelle/datum/@normiert)`, … noch das Jahr extrahieren …
6. `2016-(//gedicht[1]/bibliografisch/quelle/datum/substring(@normiert,1,4))`, und in eine Zahl verwandeln …
7. `2016-number(//gedicht[1]/bibliografisch/quelle/datum/substring(@normiert,1,4))`

9.6 | XSLT

Transformation von XML-Dokumenten: XSL (eXtensible Stylesheet Language) ist eine Gruppe von Sprachen zur Transformation von XML-Dokumenten. Häufig wird XSL synonym mit XSLT verwendet, obwohl letzteres formal nur einer von zwei Teilen ist. XSLT ist eine Programmiersprache, mit der ein XML-Dokument in ein anderes Dokument in einem beliebigen textbasierten Format umgewandelt werden kann. Ein **XSLT-Prozessor** verarbeitet ein oder mehrere XML-Dokumente gemäß der Angaben in einem XSLT-Dokument und erzeugt ein oder mehrere Ergebnisdokumente in einem Format, welches vom XSLT-Dokument bestimmt wird (s. Abb. 28). In XSL wird XPath zur Adressierung von Elementen und anderen Inhalten in XML-Dokumenten verwendet.

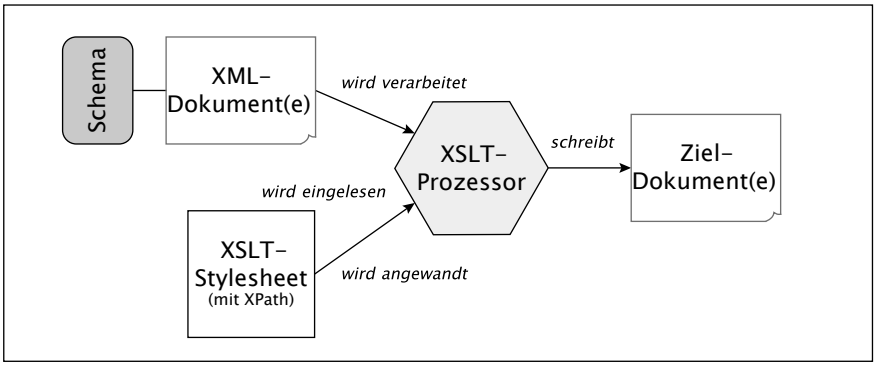

Abb. 28 Schematische Darstellung einer XSLT-Transformation

Oft sind das Ziel der Umwandlung HTML-Dokumente, also Webseiten. Dadurch können Inhalte eines projektspezifischen Datenmodells, das zur Auszeichnung von Texten oder zur Kodierung von Datenstrukturen verwendet wird, von Webbrowsern darstellt werden. Die Semantik der originalen XML-Elemente wird dabei durch die Seiten-Strukturbeschreibungen von HTML ersetzt. Man kann mit XSLT aber auch andere Dokumente im XML-Format, reinen Text, Vektorgrafiken (in SVG) oder JSON-Datenobjekte erzeugen. Mit den Komponenten von XSL können also aus einer einzigen XML-Datei unterschiedlichste Formen der Ausgabe erzeugt werden (»Single Source-Prinzip«).

Zu XSL gehört auch **XSL-FO** (eXtensible Styleheet Language, Formatting Objects), eine Sprache zur Erzeugung von abstrakten Beschreibungen von Druckdokumenten – die mit Hilfe von nachgeschalteten XSL-FO-Prozessoren als PDF ausgegeben werden können. Sie wird seit 2012 nicht mehr weiterentwickelt. Der XSLT-Standard befindet sich (Stand 2016) im Übergang von Version 2.0 zu 3.0. Während Version 2.0 eine Menge zusätzlicher Funktionen gebracht hat, ohne die ein professionelles Arbeiten kaum noch möglich ist, gibt es eine ganze Reihe von Technologien (Browser, Programmiersprachen wie PHP in seiner Standardinstallation), die nur die Befehle von XSLT 1.0 unterstützen.

Beispiel 3: Ein XSLT-Stylesheet, das die Strophen jedes Gedichts zählt und danach die Strophen als Absätze ausgibt.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="2.0">
  <xsl:template match="/">
    <h1>Meine Gedichtsammlung</h1>
    <xsl:apply-templates/>
  </xsl:template>
  <xsl:template match="gedicht">
    <xsl:value-of select="position()"/>. Gedicht: <xsl:value-of select=
    "bibliografisch/titel"/>
    <xsl:apply-templates/>
  </xsl:template>
  <xsl:template match="text">
    Das Gedicht hat <xsl:value-of select="count(strophe)"/> Strophen,
    nämlich:
    <xsl:apply-templates select="strophe"/>
  </xsl:template>
  <xsl:template match="strophe">
    <p><xsl:apply-templates select="vers"/></p>
  </xsl:template>
</xsl:stylesheet>
```

Wie wir sehen, werden die Befehle von XSL selbst in XML ausgedrückt. Nach der XML-Deklaration `<?xml version="1.0"?>` kommt das XSL-Wurzelelement `<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform">`.

Ein XSLT-Dokument besteht aus Verarbeitungsanweisungen, den sogenannten **Verarbeitungsmustern** oder **Schablonen** (**Templates,** `<xsl:template>`). In der Verarbeitung wird das XML-Dokument durchgegangen und Aktionen werden ausgeführt, sobald eine Schablone auf den aktuellen Inhalt ›passt‹. Die Schablonen können sich außerdem Informationen von anderen Stellen des XML-Dokuments holen und verarbeiten. Welches Template für welches Element verwendet werden soll, entscheidet sich anhand von XPath-Mustern im Attribut `match` und durch die Position, an der ein Template aufgerufen wird.

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

legt also die Regeln für die Verarbeitung aller Elemente mit dem Namen `strophe` fest. Der Parser verarbeitet das XML-Dokument immer von einer dem Wurzelknoten übergeordneten Struktur aus. Ein XSLT-Stylesheet besitzt deshalb in der Regel ein Template, das auf das gesamte Dokument passt, das als sein Wurzelelement mit einem einfachen Schrägstrich angesprochen wird:

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

Innerhalb einer Schablone können Elemente konstruiert, Texte geschrieben, andere XSL-Befehle ausgeführt oder weitere templates aufgerufen werden. Um Inhalte aus dem Ausgangsdokument auszulesen und z. B. im Zieldokument auszugeben, kann `<xsl:value-of>` mit einem XPath-Ausdruck im Attribut `select` verwendet werden (im Beispiel 3 wählt `<xsl:value-of select="bibliografisch/titel"/>` den Titel des Gedichts aus) oder man überlässt die weitere Verarbeitung einem anderen Template (z. B. mit `<xsl:apply-templates>`). Auch im letzteren Fall kann man das gewünschte Element des Ausgangsdokuments mit dem Attribut `select` ansprechen (das Beispiel `<xsl:apply-templates select="strophe"/>` lässt nur die Strophen des Gedichts verarbeiten). Wenn es für das aufgerufene Element kein passendes Template gibt, dann wenden die XSL-Prozessoren sogenannten ›built-in-Regeln‹ an und geben automatisch den Inhalt des Elements und aller Unterelemente aus.

Neben der Deklaration und dem Aufruf von templates (`template`, `apply-templates`) sowie dem Auslesen von Inhalten (`value-of`) werden die folgenden XSL-Elemente besonders häufig eingesetzt:

XSL-Befehl	Aufgabe
`<xsl:for-each select="XPath">`	tue etwas für jedes Element der mit dem XPath-Ausdruck ausgewählten Knotenmenge
`<xsl:sort select="XPath" data-type="text|number" order="ascending|descending"/>`	sortiert eine Knotenmenge. Leeres Element! Unmittelbar nach `<xsl:for-each>` oder `<xsl:apply-templates>`
`<xsl:if test="XPath">`	führe die folgenden Anweisungen nur aus, wenn der XPath-Ausdruck im Attribut `test` »true« zurückgibt

`<xsl:choose>`
`  <xsl:when test="XPath">`
`  <xsl:otherwise>`
	wählt unter beliebig vielen verschiedenen `when`-Bedingungen die erste, für die der XPath-Ausdruck im Attribut `test` »true« zurückgibt und führt sie aus. Sonst `otherwise`

`<xsl:variable name="string">`	eine Variable mit dem namen im Attribut `name`; der Variableninhalt kann innerhalb des Elements oder im Attribut `select="XPath"` aufgebaut werden. Variablen werden in XPath-Ausdrücken mit einem Dollar-Zeichen eingeleitet (`$string`) aufgerufen. Sie können auch komplette Bäume enthalten.

Tab. 7 XSL-Befehle

Es gibt zwei Stile, mit den Befehlen von XSL umzugehen: Der eine bevorzugt die Verwendung von Verarbeitungsmustern in Templates und verwendet zum Aufruf von Inhalten bevorzugt das XSL-Element `<xsl:apply-templates/>` (**Push-Paradigma** – Der Inhalt des Ausgangsdokuments wird beim Durchgehen einfach passenden Schablonen überlassen). Der andere rückt den Gesamtaufbau des Zieldokuments in den Vordergrund und integriert an den jeweils passenden Stellen das XSL-Element `<xsl:value-of>`, um die Inhalte des Ausgangsdokuments abzurufen (**Pull-Paradigma**). Im Push-Paradigma geschriebener Code ist schwieriger zu lesen, aber leichter zu pflegen und letztlich meistens eleganter.

9.7 | X-Technologien im Einsatz

Einfache Erstellung: XML-Dateien lassen sich mit jedem beliebigen Textverarbeitungsprogramm erstellen, das es erlaubt, Texte als Plain Text ohne Formatierungen zu speichern. Einige Plain-Text-Editoren (z. B. »jEdit«, »gEdit«, »notepad++«) bieten auch eine Unterstützung bei der Erfassung von XML durch farbige Hervorhebung des Codes, automatische Ergänzungen und Validierung. Daneben gibt es spezielle **XML-Editoren**, die ein großes Arsenal an zusätzlichen Funktionen bereitstellen, um XML- und verwandte Dokumenttypen (Schemadateien, XSL-Stylesheets, XQuery-Routinen) nicht nur auf verschiedene Weise (im Code, als visuelle Eingabeumgebung) zu schreiben, sondern auch in größeren Projekten zu verwalten, in Datenbanken einzubinden oder Verarbeitungsszenarien zu entwickeln. In den digitalen Geisteswissenschaften ist vor allem das Programm »oXygen« von der Firma SyncRo-Soft verbreitet. Es bietet zu allen Aspekten rund um XML die am weitest gehende Unterstützung. Unter den Open Source-Lösungen bietet »XMLcopy« einige nützliche Funktionalitäten, ist aber bei weitem nicht so mächtig wie oXygen oder andere kommerzielle XML-Editoren (z. B. »XMLSpy«, »XMLMind«, »Stylus Studio«).

XML steht als Technologie nicht für sich alleine, sondern ist Teil einer größeren **Familie von Standards**. Wir hatten gesehen, dass Schemata, XPath und XSLT wichtige zusätzliche Mittel sind, um Daten nicht nur zu strukturieren und zu beschreiben, sondern auch durch Regelwerke zu kontrollieren, in den Daten zu navigieren und sie durch Transformationen weiterzuverarbeiten. Eine Reihe von Standards ist in diesen Kernkomponenten bereits mitgedacht oder können sie erweitern:

- **XML-Infoset** definiert grundlegende Konzepte und Begriffe für XML.
- **XInclude** erlaubt die Einbindung externer XML-Dokumente.
- **XPointer** unterstützt die Referenzierung von bestimmten Teilen von XML-Dokumenten.

Andere Standards decken weitere wichtige oder nützliche Funktionsbereiche ab:

- **XQuery** ist eine Abfragesprache, mit der komplexe, datenbankbasierte Anwendungen entwickelt werden.
- **XForms** beschreibt Eingabeformulare und ihre Funktionalitäten zur Erfassung von Daten auf einer abstrakten Ebene.
- **XProc** beschreibt Verarbeitungsketten für XML-Dokumente (processing pipelines).

Alle Standards aus der X-Familie werden vom World Wide Web Consortium (W3C) betreut. Manche Komponenten werden aktiv weiterentwickelt und enthalten bei neuen Versionen erweiterte Möglichkeiten. Dies ist z. B. bei XPath und XSLT der Fall, deren Funktionsumfang sich im Übergang von 1.0 nach 2.0 und aktuell 3.0 stark erweitert hat. Andere gelten als stabil oder werden nicht weiterentwickelt, weil sie in der Praxis nur von geringerer Bedeutung sind.

Weiterverarbeitung: XML beschreibt Daten, Information und Wissen. Andere Standards aus der X-Familie helfen dabei, mit diesen Daten weiter zu arbeiten. Um aus XML schließlich Publikationen oder Anwendungen zu erstellen, nutzt man verschiedene Softwarekomponenten. Hierzu gehören vor allem XML-Parser bzw. XPath-, XSLT- und XQuery-Prozessoren, bei denen das Programm »Saxon« in der Regel den aktuellen Entwicklungsstand der Standards zuverlässig abbildet. »Saxon« ist

in den großen XML-Editoren per Voreinstellung eingebunden, kann aber auch von anderen Programmiersprachen genutzt werden. Vorsicht ist geboten, wenn man Programmiersprachen oder Softwareumgebungen zur Verarbeitung von XML nutzen will, die den aktuellen Stand der XML-Standards in ihren Programmbibliotheken nicht vollständig abdecken.

Für die Entwicklung von **Anwendungen auf der Basis von XML-Daten** gibt es viele verschiedene Wege und Szenarien, die sich grob in fünf Gruppen unterteilen lassen:

1. **Statistische Ergebnisse generieren:** Mit XSL(T) lassen sich leicht statische Ergebnisse generieren. Dies können Visualisierungen, vorverarbeitete Daten für andere Anwendungen, Druckvorlagen (z. B. mittels XSL-FO in PDF) aber auch komplette Webseiten sein. Für die Mächtigkeit bereits dieses simplen Ansatzes ist zu beachten, dass mit XSLT auch beliebig viele Ausgangsdokumente zusammen zu beliebig vielen Zieldokumenten verarbeitet werden können.
2. **XML-Darstellung im Browser:** Grundsätzlich kann XML auch direkt an Webbrowser ausgeliefert werden. Wenn dabei eine XSLT-Datei mit referenziert ist, können die Browser die Verarbeitung zu Darstellungsformen selbst übernehmen. Dieser Ansatz hat sich bisher allerdings nicht durchgesetzt.
3. **Verarbeitungsketten:** Die Verarbeitung von XML mit XSLT kann noch komplexer und mächtiger gestaltet werden, wenn (z. B. mit XProc) Verarbeitungsketten aufgebaut werden oder XML-Publishing-Systeme wie »Apache Cocoon« oder »Kiln« genutzt werden. Auf XSLT aufbauende Verarbeitungsketten können aber auch genutzt werden, um XML-Daten in Content Management Systeme (CMS) wie Wordpress, Typo3 oder Drupal einzuspeisen.
4. **XML-Datenbanken und Schnittstellen:** Der konsequente XML-Weg zu professionellen Anwendungen verläuft heute oft über native XML-Datenbanken wie eXist oder BaseX. Damit werden dann nicht nur die Daten verwaltet, sondern häufig auch über XQuery-Abfragen komplexe Applikationen aufgebaut. In diesem Szenario kann man ganz in der Welt der aufeinander abgestimmten X-Technologien bleiben. Es ist aber auch möglich, mit XML, XML-Datenbanken und XQuery die Daten vorverarbeitet an Schnittstellen (wie z. B. im REST-Paradigma) zu platzieren und darauf eine weitere Programmierschicht (z. B. in PHP oder JavaScript) aufzusetzen, die die eigentliche Präsentation steuert.
5. **XML-Verarbeitung mit anderen Programmiersprachen:** XML kann schließlich auch in anderen Umgebungen eingesetzt oder mit anderen Programmiersprachen verarbeitet werden. Dies hat seine Reize in der Mächtigkeit, Ausgereiftheit und weiten Verbreitung von gängigen Datenbanksystemen wie MySQL oder PostgreSQL, von modernen Datenmodellen wie JSON oder Programmiersprachen wie PHP, Python, Ruby oder Java. Es kann aber seine Tücken haben, wenn damit das Potential von XML als komplexer Datenstruktur oder die Möglichkeiten aktueller Verarbeitungsstandards (wie XPath 2.0/3.0, XSLT 2.0/3.0) nicht ausgeschöpft werden.

Literatur

Dubinko, Micah: *XForms essentials*. Sebastopol 2003.
Kay, Michael: *XSLT 2.0 and XPath 2.0 Programmer's Reference*. Indianapolis [4]2011. (*Die* Referenz.)
Krüger, Manfred: *XSL-FO verstehen und anwenden. XML-Verarbeitung für PDF und Druck*. Heidelberg 2006.
Lehner, Wolfgang/Schöning, Haralb: *XQuery: Grundlagen und fortgeschrittene Methoden*. Heidelberg 2004.
Siegel, Erik/Retter, Adam: *eXist: A NoSQL Document Database and Application Platform*. Sebastopol 2015.
Vonhoegen, Helmut: *Handbuch: Einstieg in XML: Grundlagen, Praxis, Referenz*. Bonn [8]2015. (Das beste deutschsprachige Einsteiger-Handbuch, auch wenn es dem datenzentrierten Blick auf XML verpflichtet ist und die dokumentorientierten Aspekte zu kurz kommen.)

Das Kapitel hat wesentlich von der Unterstützung durch die Mitglieder des IDE, insbesondere Martina Scholger und Ulrike Henny, profitiert.

Georg Vogeler und Patrick Sahle