Verschiedene Bestandteile einer XML-Datei

Eine XML-Datei kann nicht nur Auszeichnungen (d.h. Tags wie z.B. `<adresse/>` oder `<adresse>` oder `</adresse>`) und Text enthalten, sondern noch zahlreiche weitere Bestandteile. Im folgenden Abschnitt werden einige dieser Bestandteile anhand eines Beispiel-Dokuments erläutert.

1. Ein Beispiel für ein XML-Dokument

Das folgende XML-Dokument enthält im Wesentlichen Informationen über eine Person: Alan Turing war Informatiker, Mathematiker und Kryptograf ("Spezialist für das Verschlüsseln und Entschlüsseln von Daten"). Außerdem enthält das Dokument einige weitere Bestandteile, die nur dazu dienen, einige Möglichkeiten von XML zu verdeutlichen. Die Zeilennummern am linken Rand und die halbfett gedruckten Anmerkungen am rechten Rand gehören *nicht* zum Inhalt des Dokuments.

```
 1 <?xml version="1.0" encoding="UTF-8" standalone="no"?>   XML-Deklaration
 2 <!-- - - - - - - - - - - - - - - - - - - - - - - - - -   Kommentar Anfang
 3    Datei daten02B.xml:
 4    Wohlgeformt und gueltig (valid) nach personA.dtd und
 5    personA.xsd.
 6    Enthaelt einen Hinweis auf personA.dtd
 7 - - - - - - - - - - - - - - - - - - - - - - - - - -   -->  Kommentar Ende
 8 <!DOCTYPE person SYSTEM "personA.dtd">                   Dokumenten-Typ-Dekl.
 9
10 <person>                                                 Wurzel-Element Anf.
11   <name>                                                 Element-Anf
12     <vor_name>Alan</vor_name>                            Element
13     <nach_name>Turing</nach_name>                        Element
14   </name>                                                Element-Ende
15   Normaler                                               G-Text, 2 Zeilen
16   Text Nr 1
17   <!--Kommentar zwischen 2 Texten-->                     Kommentar
18   Normaler Text Nr 2                                     G-Text, 1 Zeile
19   <beruf>Informatiker</beruf>                            Element
20   <![CDATA[<beruf>Dozent</beruf>]]>                      CDATA-Sektion
21   <beruf>Mathematiker</beruf>                            Element
22   Entity Ref.: &amp;lt; steht fuer &lt;                  G-Text mit E-Refs
23   <beruf land="England" zeit="WK2">Kryptograf</beruf>     Element (mit 2 Attr.)
24   <?SuperEditor Zeilenwechsel ignorieren!?>              Verarbeit. Anweis.
25   &hi;                                                   E-Ref
26 </person>                                                Wurzel-Element Ende
```

Die Bestandteile dieser Datei werden im Folgenden kurz erläutert.

1.1. Prolog und Wurzel-Element

Jedes XML-Dokument muss ein *Wurzel-Element* (engl. root element) enthalten. Im Beispiel ist das ein `person`-Element (in den Zeilen 10 bis 26). Alles was *vor* dem Wurzelelement steht, gehört zum *Prolog* ("Vorwort") des Dokuments. Im Beispiel sind das die Zeilen 1 bis 9.

Das Wurzel-Element kann weitere Elemente und andere Bestandteile enthalten. Im Beispiel enthält das `person`-Element ein `name`-Element, drei `beruf`-Elemente und sieben weitere Bestandteile.

1.2. Der Prolog

Im Beispiel enthält der Prolog (Zeilen 1 bis 9)
eine *XML-Deklaration* (in Zeile 1)
einen *Kommentar* (in den Zeilen 2 bis 7),
eine *Dokumenten-Typ-Deklaration* (in Zeile 8) und
eine Leerzeile (Zeile 9).

Die XML-Deklaration beschreibt
- die verwendete *Version von XML* (bis auf weiteres ist das die Version 1.0),
- den *Zeichencode*, in dem das XML-Dokument abgespeichert wurde bzw. werden soll (im Beispiel: "UTF-8", 8-Bit Unicode Transformation Code, der am weitesten verbreitete Zeichencode für Unicode-Zeichen), und
- dass das Dokument "nicht selbständig" ist, sondern zu seinem Verständnis weitere Dateien benötigt werden (`standalone="no"`).

Das Beispiel-Dokument ist *nicht selbständig*, weil in Zeile 25 eine *E-Referenz* (engl. entity reference) `&hi;` steht. Das ist eine Art *Abkürzung*, deren Bedeutung in einer DTD (siehe unten) definiert werden muss.

Ein Kommentar muss mit `<!--` beginnen und mit `-->` enden und darf keine zwei unmittelbar aufeinander folgende Minuszeichen `--` enthalten.

Die Dokumenten-Typ-Deklaration (in Zeile 8) gibt an, wo eine DTD (document type definition) steht, die die Struktur ("den Typ") dieser XML-Datei beschreibt und hoffentlich auch eine Definition der E-Referenz ("Abkürzung") `&hi;` enthält.

Der Prolog einer XML-Datei kann auch leer sein, enthält in aller Regel aber mindestens eine XML-Deklaration.

1.3. Elemente

Ein Element beginnt mit einem *Anfangs-Tag* (z.B. `<name>` in Zeile 11) und wird durch einen *End-Tag* (z.B. `</name>` in Zeile 14) abgeschlossen. Ein solches Element kann leer sein oder weitere Bestandteile (Elemente, Text, Kommentare, ... etc.) enthalten.

Außer solchen 2-Tag-Elementen gibt es auch Elemente, die nur aus *einem* Tag bestehen (z.B. so: `<einfach/>` oder `<farben hintergrund="rot" vordergrund="schwarz"/>`).

Jedes XML-Dokument muss genau ein *Wurzel-Element* enthalten. Weitere Elemente dürfen nur innerhalb des Wurzel-Elements vorkommen (aber weder davor noch danach).

Sowohl 1-Tag-Elemente als auch 2-Tag-Elemente können *Attribute* besitzen (siehe unten).

1.4. Der Inhalt eines Elements: Gemischt oder ungemischt

Wenn ein Element weitere Elemente *und* Text (direkt) enthält so ist sein Inhalt *gemischt* (engl. the element is in mixed mode). Enthält ein Element nur Text (und keine weiteren Elemente) oder nur weitere Elemente (aber keinen Text), so ist sein Inhalt *ungemischt*.

Im Beispiel ist der Inhalt des `person`-Elements gemischt, die Inhalte aller anderen Elemente sind dagegen ungemischt.

Die Texte in einem Element mit gemischtem Inhalt bezeichnen wir hier auch als *gemischte Texte* (kurz: G-Texte). Wenn ein Element mit ungemischtem Inhalt einen Text enthält, so bezeichnen wir den als *ungemischten Text* (kurz: U-Text).

Im Beispiel sind der Text `Normaler Text Nr 1` in Zeile 15 bis 16 und der Text `Normaler Text Nr 2` in Zeile 18 G-Texte. Dagegen ist der Text `Kryptograf` in Zeile 23 ein U-Text. Das Element `name` (in Zeile 11 bis 14) enthält direkt keine Texte, nur ein `vor_name`- und ein `nach_name`-Element. Diese Elemente enthalten ihrerseits die U-Texte `Alan` bzw. `Turing`.

1.5. Attribute eines Elements

Ein Element kann (nicht nur weitere Elemente und/oder Text enthalten, sondern auch) *Attribute* besitzen. Diese werden im Anfangs-Tag (bzw. im einzigen Tag) des Elements notiert. Ein Attribut besteht aus einem *Namen* und einem *Wert*.

Im Beispiel hat nur das `beruf`-Element in Zeile 23 zwei Attribute, eines namens `land` mit dem Wert `"England"` und eines namens `zeit` mit dem Wert `"WK2"`.

1.6. Entitäts-Referenzen (E-Refs)

Innerhalb eines Elements wird das Kleiner-Zeichen < normalerweise als Anfang eines Tägs interpretiert. Mit der Fluchtsequenz `<` ("lt" wie "less than") kann man ein "einfaches" Kleiner-Zeichen darstellen.

Diese (etwas unglückliche) Festlegung funktioniert nur, weil das Ampersand & normalerweise als Anfang einer Fluchtsequenz interpretiert wird. Ein "einfaches" Ampersand muss man deshalb mit der Fluchtsequenz `&` darstellen.

Ganz entsprechend sind in XML noch 3 weitere Fluchtsequenzen vordefiniert:

`>`	("gt" wie "greater than")	bezeichnet ein Größer-Zeichen >
`&quot;`	("quot" wie "quotation mark")	bezeichnet ein doppeltes Anführungszeichen "
`&apos;`	("apos" wie "apostroph")	bezeichnet ein einfaches Anführungszeichen '

Diese 5 Fluchtsequenzen (engl. escape sequences) werden in XML (ziemlich pompös) als *Entitäts-Referenzen* (engl. entity references) bezeichnet. In diesem Papier wird dafür die Abkürzung *E-Ref* verwendet.

Innerhalb einer *Dokumenten Typ-Definition* (DTD) kann man weitere Fluchtsequenzen definieren, z.B. so:

```
<!ENTITY hi "Hi Alan, how are you?">
```

Im Wirkungsbereich dieser Definition kann man die E-Ref `&hi;` als Abkürzung für den Text `Hi Alan, how are you?` verwenden. Im obigen XML-Dokument geschieht das in Zeile 23.

Mit Hilfe solcher Definitionen und E-Refs kann man bewirken, dass ein XML-Dokument nicht mehr selbständig ist und nur zusammen mit einer geeigneten DTD verstanden werden kann (sonst könnte ja jeder XML-Dokumente erstellen und verstehen :-).

1.7. CDATA-Sektionen

Werden in den Texten eines XML-Dokuments viele Ampersands &, Kleiner-Zeichen < etc. durch E-Refs wie `&` und `<` etc. dargestellt, so werden sie dadurch oft schwer lesbar. Für solche Fälle gibt es einen weiteren Fluchtmechanismus: Eine CDATA-Sektion beginnt mit `<![CDATA[` und wird mit `]]>` abgeschlossen. Innerhalb einer solchen Sektion werden alle Zeichen "einfach" interpretiert (mit Ausnahme der Zeichenkette `]]>`, die die CDATA-Sektion abschliesst).

Im obigen XML-Dokument steht in Zeile 20 die CDATA-Sektion

```
<![CDATA[<beruf>Dozent</beruf>]]>
```

Sie bewirkt, dass ihr Inhalt `<beruf>Dozent</beruf>` als einfacher *Text* interpretiert wird, und nicht als ein `beruf`-Element. Wenn man diesen Text statt mit einer CDATA-Sektion wie folgt darstellt:

```
&lt;beruf&gt;Dozent&lt;/beruf&gt;
```

ist viel schwerer zu erkennen, dass er wie ein `beruf`-Element aussieht.

1.8. Verarbeitungsanweisungen (engl. processing instructions)

Eine Verarbeitungsanweisung besteht aus einem *Namen* und einer *Anweisung*. Der Name bezeichnet typischerweise ein Programm zum Bearbeiten von XML-Dokumenten (z.B. einen Editor oder einen-Parser).

Das obige XML-Dokument ethält in Zeile 24 die Verarbeitungsanweisung

```
<?SuperEditor Zeilenwechsel ignorieren!?>
```

Sie besteht aus dem Namen `SuperEditor` und der Anweisung `Zeilenwechsel ignorieren!`. Hoffentlich gibt es ein Programm namens `SuperEditor` welches diese Anweisung versteht. Programme zum Bearbeiten von XML-Dokumenten dürfen jede einzelne Verarbeitungsanweisungen wahlweise *befolgen* oder *ignorieren*.

Verarbeitungsanweisungen gehören zu den *Auszeichnungen* (engl. markup) eines Dokuments, zählen aber nicht als *Elemente*. Sie dürfen (ähnlich wie Kommentare) im Prolog, im Wurzel-Element oder auch hinter dem Wurzel-Element stehen.

Warum stellt man Verarbeitungsanweisungen nicht einfach durch spezielle Kommentare dar, z.B. so:
`<!-- ProcInst SuperEditor Zeilenwechsel ignorieren! --> ?`

1.9. Kommentare

Ein Kommentar muss mit `<!--` beginnen und mit `-->` enden und darf keine zwei unmittelbar aufeinanderfolgende Minuszeichen `--` enthalten.

Kommentare dürfen (ähnlich wie Verarbeitungsanweisungen) im Prolog, im Wurzel-Element oder auch hinter dem Wurzel-Element eines XML-Dokuments stehen.

2. Wohlgeformte und gültige (well-formed and valid) XML-Dokumente

Wenn ein XML-Dokument genau ein Wurzel-Element enthält, darin zu jedem Anfangs-Tag an der richtigen Stelle ein End-Tag steht und ein paar ähnliche Bedingungen erfüllt sind, dann ist das Dokument *wohlgeformt* (engl. well-formed).

Um ein XML-Dokument auf Wohlgeformtheit zu prüfen, braucht man normalerweise nur das Dokument selbst zu untersuchen. Ausnahme: Wenn das Dokument E-Refs wie z.B. `&hi;` enthält, braucht man zusätzlich eine geeignete Dokumenten Typ-Definition (DTD) in der definiert wird, wofür die E-Refs stehen.

Manchmal will man genauer prüfen, ob ein Dokument nicht nur irgendein, sondern *das richtige* Wurzel-Element enthält (z.B. ein `person`-Element und nicht ein `fahrzeug`-Element) und ob das Wurzel-Element die richtigen Elemente und anderen Bestandteile enthält etc. Bevor man eine solche weitergehende Prüfung durchführen kann, muss man den gewünschten *Typ des Dokuments* beschreiben.

Für solche Typ-Beschreibungen wurden eine Reihe verschiedener Sprachen entwickelt, unter anderen die folgenden:

- Dokumenten Typ-Definitionen (DTDs)
- XML-Schemas
- RELAX NG (REgular LAnguage for XML, Next Generation)
- Schematron

DTD war historisch die erste Sprache zur Beschreibung von XML-Typen. Sehr bald stellte sich heraus, dass sie für einige Anwendungen unzureichend ist. Deshalb wurden und werden weitere, mächtigere Typ-Beschreibungssprachen entwickelt.

3. Ein Beispiel für eine Dokumenten Typ-Definition (DTD)

Die folgende DTD ist ein Versuch, den Typ des Beispiel-Dokuments im Abschnitt 1. zu beschreiben:

```
 1 <?xml version="1.0" encoding="UTF-8"?>
 2 <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
 3    Datei personA.dtd:
 4    Document Type Definition fuer die Dateien daten02x.xml
 5 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -  -->
 6 <!ELEMENT person    (name, beruf+)>
 7 <!ELEMENT name      (vor_name, nach_name)>
 8 <!ELEMENT vor_name  (#PCDATA)>
 9 <!ELEMENT nach_name (#PCDATA)>
10 <!ELEMENT beruf     (#PCDATA)>
11
12 <!ATTLIST beruf land CDATA #IMPLIED
13                 zeit CDATA #IMPLIED
14 >
15
16 <!ENTITY hi "Hi Alan, how are you?">
```

Diese Typ-Definition drückt Folgendes aus:

Ein `person`-Element muss genau ein `name`-Element und ein oder mehr `beruf`-Elemente enthalten (Zeile 6. Das Pluszeichen + bedeutet hier "1 oder mehr". Alternativ könnte man auch ein Sternchen * für "0 oder mehr" oder ein Fragezeichen ? für "0 oder 1" angeben).

Ein `name`-Element muss genau ein `vor_name`-Element und genau ein `nach_name`-Element enthalten (Zeile 7).

Ein `vor_name`-Element darf einen beliebigen Text enthalten (Zeile 8). "`PCDATA`" ist eine Abkürzung für "parsed character data". Die Zeilen 9 und 10 bedeuten ganz entsprechendes.

Ein `beruf`-Element darf ein Attribut namens `land` und ein Attribut namens `zeit` besitzen (Zeile 12 bis 14). Als Werte dieser Attribute sind beliebige Texte erlaubt ("`CDATA`" steht hier für "character data"). `#IMPLIED` bedeutet hier soviel wie *optional*, d.h. man darf jedes der beiden Attribute auch weglassen. Anstelle von `#IMPLIED` kann man auch `#REQUIRED` angeben. Dann *muss* jedes `beruf`-Element ein entsprechendes Attribut besitzen.

In Zeile 16 steht, dass eine E-Ref `&hi;` für den Text `Hi Alan, how are you?` steht.

Leider kann man mit einer DTD nicht beschreiben, dass vor oder nach jedem `beruf`-Element auch beliebige Texte stehen dürfen. Nur wenn man aus dem obigen XML-Dokument alle gemischten Texte (G-Texte) entfernt (Zeile 15 bis 16, 18, 20, 22, 24 und 25) wird es gültig (engl. valid) entsprechend dieser DTD.

Häufig schreibt man eine Dokumenten Typ-Definition in eine separate Datei (z.B. eine namens `personA.dtd`) und verweist am Anfang aller XML-Dateien, die ihr entsprechenden sollen, auf diese DTD-Datei, etwa so:

```
<!DOCTYPE person SYSTEM "personA.dtd">
```

Man kann die DTD aber auch auch ganz oder teilweise in die XML-Datei integrieren, etwa so:

```
 1 <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
 2 <!-- Datei daten02I.xml -->
 3
 4 <!DOCTYPE name [
 5   <!ELEMENT name (vor_name, nach_name)>
 6   <!ELEMENT vor_name  (#PCDATA)>
 7   <!ELEMENT nach_name (#PCDATA)>
 8 ]>
 9
10 <name>
11    <vor_name> Alan  </vor_name>
12    <nach_name>Turing</nach_name>
13 </name>
```

In den Zeilen 6 bis 10 wird der Typ `name` vollständig definiert.

Im folgenden Beispiel wurde nur ein Teil der DTD in die XML-Datei integeriert und ein anderer Teil in eine separate Datei (namens `nameC.dtd`) ausgelagert:

```
14 <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
15 <!-- Datei daten02J.xml -->
16
17 <!DOCTYPE name SYSTEM "nameC.dtd" [
18   <!ELEMENT name (vor_name, nach_name)>
19 ]>
20
21 <name>
22    <vor_name> Alan  </vor_name>
23    <nach_name>Turing</nach_name>
24 </name>
```

Die DTD-Datei `nameC.dtd` muss die noch fehlenden Definitionen der Typen `vor_name` und `nach_name` enthalten etwa so:

```
25 <?xml version="1.0" encoding="UTF-8"?>
26 <!-- Datei nameC.dtd -->
27
28   <!ELEMENT vor_name  (#PCDATA)>
29   <!ELEMENT nach_name (#PCDATA)>
```

Die XML-Datei `daten02J.xml` hat eine *interne* und eine *externe DTD-Teilmenge*. Die interne Teilmenge (in Zeile 18) definiert den Typ `name`. Die externe Teilmenge (in Zeile 25 bis 29) definiert die Typen `vor_name` und `nach_name`.

4. Zwei Beispiele für XML-Schema-Definitionen

Die Typ-Beschreibungssprache *XML-Schema* ist deutlich mächtiger als die DTD-Sprache. Das folgende Schema beschreibt einen Typ, zu dem auch das obige XML-Dokument (so wie es ist, einschließlich aller G-Texte) gehört:

```
 1 <?xml version="1.0" encoding="UTF-8"?>
 2 <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
 3    Datei personA.xsd
 4    XML-Schema-Datei fuer die Dateien daten02x.xml
 5    Hier werden alle Typen "so lokal wie moeglich" definiert
 6 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -  -->
 7 <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 8     elementFormDefault="qualified">
 9
10   <xsd:element name="person">
11     <xsd:complexType mixed="true">
12       <xsd:sequence>
13
14         <xsd:element name="name">
15           <xsd:complexType>
16             <xsd:sequence>
17               <xsd:element name="vor_name"  type="xsd:string"/>
18               <xsd:element name="nach_name" type="xsd:string"/>
19             </xsd:sequence>
20           </xsd:complexType>
21         </xsd:element>
22
23         <xsd:element name="beruf" maxOccurs="unbounded">
24           <xsd:complexType>
25             <xsd:simpleContent>
26               <xsd:extension base="xsd:string">
27                 <xsd:attribute name="land"  type="xsd:string"/>
28                 <xsd:attribute name="zeit"  type="xsd:string"/>
29               </xsd:extension>
30             </xsd:simpleContent>
31           </xsd:complexType>
32         </xsd:element>
33
34       </xsd:sequence>
35     </xsd:complexType>
36   </xsd:element> <!-- person -->
37
38 </xsd:schema>
```

Dieses Schema drückt Folgendes aus:

Ein Element namens `person` (Zeile 10) gehört zu einem komplexen Typ (Zeile 11) und kann außer Elementen auch gemischte Texte (G-Texte) enthalten (`mixed="true"`).

Der komplexe Typ des `person`-Elements ist ein Folgen-Typ (Zeile 12). Eine Folge dieses Typs besteht aus *einem* Element namens `name` (beschrieben in Zeile 14 bis 21) und *beliebig vielen* Elementen (`maxOccurs="unbounded"`) namens `beruf` (beschrieben in Zeile 23 bis 32).

Das Element namens `name` gehört zu einem komplexen Typ (Zeile 15), weil es eine Folge von Elementen enthält (Zeile 16 bis 19).

Jedes Element namens `beruf` gehört ebenfalls zu einem komplexen Typ (Zeile 24), aber aus einem anderen Grund: Weil es zwei *Attribute* besitzt (Zeile 27 und 28). Attribute sind standardmäßig *optional*, könnten aber auch als `required` beschrieben werden.

In diesem Schema kommt nur *ein* Typ-Name vor: Der Name `xsd:string` des einfachen, vordefinierten String-Typs. Alle anderen Typen (der Typ des `person`-Elements, der Typ des `name`-Elements, der Typ der `beruf`-Elemente) werden zwar beschrieben, bleiben aber namenlos.

Das folgende Schema definiert einen gleichen Typ von XML-Dokumenten wie das obige Schema, aber auf eine ganz andere Weise: Alle Typen werden mit Namen versehen und "global definiert" (d.h. direkt im `schema`-Element und nicht in anderen, tiefer geschachtelten Elementen):

```
 1 <?xml version="1.0" encoding="UTF-8"?>
 2 <!-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
 3    Datei personB.xsd
 4    XML-Schema-Datei fuer die Dateien daten02x.xml
 5    Hier werden alle Typen "so global wie moeglich" definiert
 6 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -  -->
 7 <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 8     elementFormDefault="qualified">
 9
10   <xsd:element name="person" type="personTyp"/>
11
12   <xsd:complexType name="personTyp" mixed="true">
13     <xsd:sequence>
14       <xsd:element name="name"  type="nameTyp"/>
15       <xsd:element name="beruf" type="berufTyp" maxOccurs="unbounded"/>
16     </xsd:sequence>
17   </xsd:complexType> <!-- personTyp -->
18
19   <xsd:complexType name="nameTyp">
20     <xsd:sequence>
21       <xsd:element name="vor_name"  type="xsd:string"/>
22       <xsd:element name="nach_name" type="xsd:string"/>
23     </xsd:sequence>
24   </xsd:complexType> <!-- nameTyp -->
25
26   <xsd:complexType name="berufTyp">
27     <xsd:simpleContent>
28       <xsd:extension base="xsd:string">
29         <xsd:attribute name="land"  type="xsd:string"/>
30         <xsd:attribute name="zeit"  type="xsd:string"/>
31       </xsd:extension>
32     </xsd:simpleContent>
33   </xsd:complexType> <!-- berufTyp -->
34
35 </xsd:schema>
```

Dieses Schema drückt Folgendes aus:

Das `person`-Element (Zeile 10) gehört zu einem Typ namens `personTyp`. Dieser Typ wir in den Zeilen 12 bis 17 definiert.

Das `name`-Element (Zeile 14) gehört zum Typ namens `nameTyp`. Dieser Typ wird in den Zeilen 19 bis 24 definiert.

Jedes `beruf`-Element (Zeile 15) gehört zum Typ namens `berufTyp`. Dieser Typ wird in den Zeilen 26 bis 33 definiert.

Jede der beiden hier angedeuteten Vorgehensweisen beim Definieren von XML-Schemen hat spezielle Vor- und Nachteile. Im Zweifelsfall sollte man aber die zweite Vorgehensweise (bei der die Typen mit Namen versehen wurden) vorziehen.

5. Krtik an XML (positive und negative)

1. Dokumente können (erheblich einfacher) maschinell analysiert und bearbeitet werden, wenn man ihre Struktur durch *Auszeichnungen* (engl. markup) deutlich macht. Diese einfache Grundidee, auf der auch XML beruht, ist sehr gut.

2. XML hat sich bereits weit verbreitet und verbreitet sich noch weiter. Auch das ist im Prinzip sehr gut an XML.

Ursprünglich wurde XML für Dokumente entwickelt, die hauptsächlich aus freiem Text bestehen, in dem ein paar strukturierte Elementen vorkommen (*Text-Dokumente* oder *narrative Dokumente*). Benutzt wurde und wird XML aber auch sehr häufig für Dokumente, die nur aus strukturierten Elementen bestehen und keine freien Texte ausserhalb von Elementen enthalten (*Element-Dokumente* oder *datensatzartige Dokumente*).

Text-Dokumente und Element-Dokumente stellen ziemlich unterschiedliche Anforderungen an eine Auszeichnungssprache. Die Entwickler von XML haben versucht, beide "Bündel von Anforderungen" zu erfüllen. Das Ergebnis ist eine ziemlich komplexe Sprache, die sehr komplexe Werkzeuge (Editoren, Parser etc.) erfordert und in vielen Anwendungsfällen weit komplizierter ist, als es in diesem Fall nötig wäre.

Ein Vergleich: Ein flugfähiges U-Boot wird wahrscheinlich nicht so gut tauchen, wie ein nur-U-Boot und weniger gut fliegen als ein nur-Flugzeug.

3. XML ist insgesamt viel zu kompliziert.

Diese pauschale Kritik wird im Folgenden ein bisschen differenziert:

4. Für Text-Dokumente sind *Attribute* (von Elementen) nützlich. In Element-Dokumenten könnte man dagegen leicht auf Attribute verzichten, indem man sie durch Elemente darstellt. Das würde vieles erheblich vereinfachen.

5. XML enthält zuviele Fluchtsymbole: `<` und `<![CDATA[` für Elemente, einfache und doppelte Anführungszeichen (`&apos;` und `&quot;`) für Attribute und und ein Ampersand (`&`) für Elemente und Attribute. Besser wäre eine Notation, die mit einem einzigen Fluchtsymbol für alle Fälle auskommt.

6. Die Regeln für die Behandlung von *transparenten Zeichen* (engl. white space) sind zu kompliziert und trotzdem unvollständig. Für Element-Dokumente wäre es wesentlich einfacher, wenn transparente Zeichen nur innerhalb von String-Literalen signifikant wären (ähnlich wie in den Quelltexten der meisten Programmiersprachen).

7. Es ist nicht einfach, eine sinnvolle *Gleichheitsfunktion* für XML-Dokumente zu programmieren. Das beeinträchtigt die Klarheit und Verständlichkeit des Grundbegriffs *XML-Dokument.*

8. "Rundfahrten" (engl. roundtrips) sind problematisch: Liest man eine XML-Datei d1 als XML-Dokument ein und gibt das Dokument in eine Datei d2 aus, so können sich d1 und d2 erheblich unterscheiden.

9. Dokumenten-Typ-Definitionen (DTDs) erledigen zwei Aufgaben: Sie beschreiben einen Typ und enthalten Definitionen von Abkürzungen (d.h. von Entitäts-Referenzen). Diese beiden Aufgaben haben nichts miteinander zu tun. Besser wäre es, eine DTD würde nur eine Aufgabe erledigen.

10. Weil DTDs für viele Dokumente zu schwach sind, wurden zahlreiche stärkere Typ-Beschreibungssprachen entwickelt. Das zwingt Anwender dazu, sich mit den Eigenheiten mehrerer Sprachen vertraut zumachen. Wünschenswert wäre es, wenn sich *eine* "ausreichend starke" Typ-Beschreibungssprache als Standard durchsetzen würde.

11. DTDs sind ein *integraler Bestandteil* von XML statt eine *modulare Ergänzung*. Selbst wenn man keinerlei Gebrauch von DTDs machen will, muss man in aller Regel XML-Werkzeuge verwenden, die auch DTDs "berücksichtigen und bearbeiten" können und deshalb komplexer sind als nötig.

12. Die Regeln für DTDs sind zu kompliziert (insbesondere die Regeln für interne und externe DTDs und für Entitäts- und Parameter-Referenzen. Die Regeln für *externe DTDs* unterscheiden sich auf subtile und verwirrende Weise von den Regeln für *interne DTDs* etc.).

13. XML-Dateien sind nur sehr bedingt "von Menschen lesbar". Die Tags dominieren häufig das Erscheinungsbild eines Dokuments, nehmen zuviel Raum ein und "erschlagen" die Daten, um die es eigentlich geht.

14. In Java gibt es zu viele Werkzeuge zur Bearbeitung von XML-Dateien, und alle scheinen eine Daseinsberechtigung zu haben, weil keines alle Bedürfnisse aller Benutzer erfüllt. Die Komplexität von XML hat vermutlich wesentlich zum Entstehen dieser Situation beigetragen.

Ein Beispiel für eine vorbildlich *einfache* Auszeichnungssprache ist JSON (Java Script Object Notation). Eine vollständige Grammatik von JSON läßt sich leicht auf einer halben DIN-A4-Seite angeben und in wenigen Minuten lesen und verstehen (siehe `http://www.json.org/`). Es ist interessant, diese Beschreibung von JSON mit der in etwa entsprechenden Beschreibung von XML zu vergleichen (siehe `http://www.w3.org/TR/2006/REC-xml-20060816/`).

Eine andere Datenbeschreibungsprache, die deutlich einfacher ist als XML aber Ähnliches leistet, ist YAML (reimt sich mit "camel". "YAML" ist eine Abkürzung für "YAML Ain't Markup Language", siehe `http://yaml.org/`).