

## DTD-Syntax (DTD)

### **Lernziele**

- Sie wissen, wie eine DTD mit einem XML-Dokument verknüpft wird.
- Sie können Elementtypen, Attribute und Entities deklarieren.
- Sie sind in der Lage, eigene Dokumenttypen zu definieren.
- Sie können Ihre XML-Dokumente validieren.
- Sie kennen die Datentypen für Attributwerte.
- Sie wissen was Notation-Deklarationen sind.
- Sie kennen die Funktion von bedingten Abschnitten.
- Sie setzen das erworbene Wissen praktisch in den Übungen ein!

# Markup-Deklarationen

Innerhalb einer DTD werden eigene Elementtypen, Attribute und Entities definiert. Darüber hinaus legt die DTD fest, wie sich Elemente innerhalb eines XML-Dokuments aufeinander beziehen und wie sie verschachtelt sein dürfen. Dies geschieht durch sogenannte *Markup-Deklarationen*.

## Markup-Deklarationen in einer DTD

- Elementtyp-Deklaration (*Element Type Declaration*)
- Attributlisten-Deklaration (*Attribute-List Declaration*)
- Entity-Deklaration (*Entity Declaration*)
- Notation-Deklaration (*Notation Declaration*)

### **Beispiel DTD-1: Eine erste DTD**

```
<!ELEMENT katalog (produkt+)>
  <!ELEMENT produkt (spezifikation+, preis+,
                    bemerkung?)>
    <!ATTLIST produkt name CDATA #REQUIRED>

    <!ELEMENT spezifikation (#PCDATA)>
      <!ATTLIST spezifikation groesse CDATA #REQUIRED
                        farbe CDATA #REQUIRED>
    <!ELEMENT preis (#PCDATA)>
      <!ATTLIST preis einkaufspreis NMTOKEN #REQUIRED
                    ladenpreis NMTOKEN #REQUIRED
                    mwst NMTOKEN #IMPLIED>
    <!ELEMENT bemerkung>

<!-- Das folgende XML-Dokument folgt obiger DTD-->

<katalog>
  <produkt name="T-Shirt">
    <spezifikation groesse="XL" farbe="weiss"/>
    <preis einkaufspreis="5.00" ladenpreis="10.00"
mwst="1.60"/>
    <bemerkung>Geht nicht in Schlussverkauf</bemerkung>
  </produkt>
```

```
<produkt name="Hemd">  
  <spezifikation groesse="38" farbe="schwarz">  
    <preis einkaufspreis="30.00" ladenpreis="60.00"/>  
  </spezifikation>  
</produkt>  
</katalog>
```

# Einbindung von DTDs

Eine DTD können Sie auf zwei verschiedene Weisen mit einem XML-Dokument verknüpfen:

- Referenzierung einer **internen Teilmenge** (*internal subset*) von Markup-Deklarationen im selben XML-Dokument (*interne DTD*)
- Referenzierung einer **externen Teilmenge** (*external subset*) von Markup-Deklarationen in einer eigenen Datei (*externe DTD*)

In beiden Fällen dient die **Dokumenttyp-Deklaration** als Werkzeug zur Einbindung der Teilmengen.

## Interne Teilmenge

Alle Markup-Deklarationen werden innerhalb des XML-Dokuments gebündelt. Sie werden zwischen zwei eckige Klammern in die Dokumenttyp-Deklaration hineingeschrieben.

### **Interne DTD**

```
<!DOCTYPE Name [  
...Deklarationen der internen Teilmenge...  

```

### **Beispiel DTD-2: XML-Dokument mit interner DTD**

```
<?xml version="1.0" encoding="iso-8859-1"?>  
  
<!DOCTYPE mitarbeiter [  
  <!ELEMENT mitarbeiter (name, vorname, abteilung)>  
  <!ELEMENT vorname (#PCDATA)>  

```

### **Beachten Sie:**

Die Dokumenttyp-Deklaration enthält bei gültigen Dokumenten immer den Namen des Wurzelements. Stimmt dieser Name nicht mit dem Namen des Wurzelements überein, meldet der XML-Prozessor einen Fehler.

## Externe Teilmenge

Eine externe Teilmenge von Markup-Deklarationen kann in eine eigene Datei geschrieben und in der Dokumenttyp-Deklaration referenziert werden.

```
<!DOCTYPE Name SYSTEM System-Identifizier>
```

oder

```
<!DOCTYPE Name PUBLIC Public-Identifizier System-Identifizier>
```

### **Gebrauch des System-Identifiziers**

Das Schlüsselwort **SYSTEM** bedeutet, dass Sie dem XML-Prozessor explizit den Ort angeben möchten, an dem er die DTD findet. Der **System-Identifizier** ist ein URI und gibt den Pfad zur Datei als Literal an.

Die Datei kann sich auf dem Rechner befinden, auf dem sich auch der XML-Prozessor befindet:

#### **Beispiel DTD-3: Lokale Referenzierung einer DTD**

```
<!DOCTYPE auftrag SYSTEM "file:///DTDs/auftrag.dtd">
```

Aber auch auf einem entfernten (Web-)Server:

#### **Beispiel DTD-4: Referenzierung einer DTD auf entferntem Server**

```
<!DOCTYPE auftrag SYSTEM  
"http://www.XYZ.de/DTDs/auftrag.dtd">
```

Selbstverständlich können Sie auch einen relativen URI als System-Identifizier verwenden:

### **Beispiel DTD-5: Relativer URI als System-Identifizier**

```
<!DOCTYPE auftrag SYSTEM "../DTDs/auftrag.dtd">
```

#### **Beachten Sie:**

- Auf den URI des System-Identifiziers sollte kein sogenannter **Fragment-Identifizier** folgen. Folgende Angabe kann bei einigen XML-Prozessoren zu Fehlern führen:

### **Beispiel DTD-6: Fehlerhafter Gebrauch eines Fragment-Identifiziers**

```
<!DOCTYPE auftrag SYSTEM  
"http://www.XYZ.de/DTDs/auftrag.dtd#Teil1">
```

- Die doppelten Anführungszeichen um den URI gehören zum System-Identifizier

### **Gebrauch des Public-Identifiziers**

Das Schlüsselwort **PUBLIC** wird benutzt, um auf öffentliche DTDs zu verweisen, deren Name dann anzugeben ist (**Public-Identifizier**). Der XML-Prozessor versucht daraufhin, die DTD über diesen Public-Identifizier zu laden. Dies bietet Flexibilität hinsichtlich des Ortes, an dem sich die DTD befindet: Es könnte eine lokale Kopie einer DTD sein, aber auch eine DTD, die über ein lokales Netzwerk zugänglich ist oder in einer Datenbank residiert. Selbstverständlich muss der XML-Prozessor wissen, wo er die DTD findet. Daher wird der Gebrauch solcher Public-Identifizier in der Regel auf interne Systeme beschränkt sein, die Kataloge zur Verfügung stellen, welche auf die realen Ressourcen verweisen (*Mapping*).

Auf den Public-Identifizier muss in jedem Fall ein System-Identifizier folgen, der den Ort angibt, an dem der XML-Prozessor die DTD suchen soll, falls er den Public-Identifizier nicht auflösen kann. Dies ist eine Vorsichtsmaßnahme, da es (noch) keine brauchbaren Standards für Public-Identifizier gibt.

### **Beispiel DTD-7: Referenzierung einer externen DTDs über einen Public-Identifizier**

```
<!DOCTYPE html  
PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"  
"DTD/xhtml11-strict.dtd">
```

#### **Beachten Sie:**

*Die Verwendung einer externen Teilmenge erlaubt Ihnen, eine DTD für mehrere Dokumente zu benutzen!*

## Kombination von interner und externer Teilmenge

Eine DTD kann sich auch aus einer internen plus externen Teilmenge zusammensetzen. In einem solchen Fall haben die Markup-Deklarationen in der internen Teilmenge Vorrang vor den Markup-Deklarationen der externen Teilmenge. Auf diese Weise ist es möglich, globale Festlegungen in einer externen DTD zu treffen und diese z.B. in der internen DTD zu überschreiben. Dies gilt allerdings nur für Attributlisten-Deklarationen und Entity-Deklarationen. Elementtyp-Deklarationen können nicht überschrieben werden. Ein solcher Überschreibungsmechanismus ist insbesondere interessant, falls Sie fertige DTDs leicht modifiziert übernehmen möchten. So könnten Sie z.B. eine öffentliche DTD für Adressen übernehmen, aber ihr Adresselement soll noch ein zusätzliches Attribut `typ` erhalten, welches angibt, ob es sich um eine Privat- oder Geschäftsadresse handelt.

### **Beispiel DTD-8: Kombination interne und externe Teilmenge**

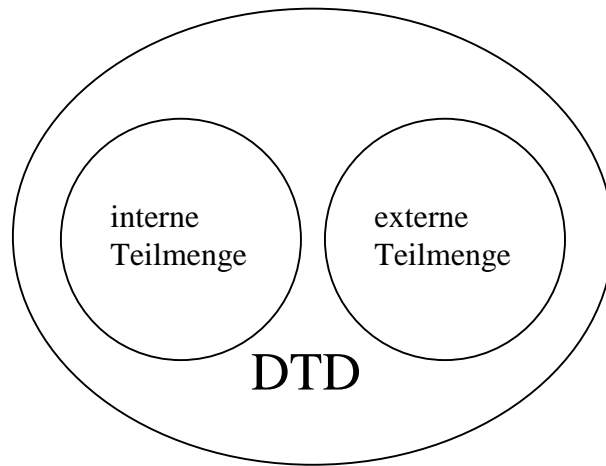
```
<?xml version="1.0"?>

<!DOCTYPE adressBuch SYSTEM "adressen.dtd"
[
  <!ATTLIST ausgabe typ CDATA #IMPLIED>
]>

<adressbuch>
  <adresse typ="privat">
    ...
  </adresse>
</adressbuch>
```

### **Beachten Sie:**

*Eine DTD besteht immer aus beiden Teilmengen zusammen!*



**Abbildung DTD-1:  $DTD = \text{interne} + \text{externe Teilmenge}$**

# Elementtyp-Deklarationen

In einer Elementtyp-Deklaration werden Bezeichnung (**Name**) und Inhalt (**Inhaltsspezifikation**) eines Elements definiert

```
<!ELEMENT Name Inhaltsspezifikation>
```

## Beispiel DTD-9: Elementtyp-Deklaration

```
<!ELEMENT mitarbeiter (vorname, name, foto?, kommentar)>  
<!ELEMENT vorname (#PCDATA)>  
<!ELEMENT name (#PCDATA)>  
<!ELEMENT foto EMPTY>  
<!ELEMENT kommentar ANY>
```

- Für die Namen gelten die üblichen Regeln.
- Doppelte Deklarationen eines Elements sind unzulässig.

## Arten von Inhaltsspezifikationen

Die beiden einfachsten Inhaltsspezifikationen werden durch die reservierten Wörter **EMPTY** und **ANY** repräsentiert.

Inhaltsspezifikation	Bedeutung
EMPTY	Element ist leer
ANY	Element darf Zeichendaten und beliebige in der DTD verfügbare Elemente enthalten

Des Weiteren lassen sich zwei Typen von Inhaltsspezifikationen unterscheiden:

### **Element-Inhalt (*element content*)**

Elemente, die ausschließlich andere Elemente enthalten dürfen (keine Zeichendaten!), haben Element-Inhalt.

### ***Inhaltsmodell***

Die interne Struktur des Element-Inhalts wird durch ein **Inhaltsmodell** definiert. Ein Inhaltsmodell besteht aus einer Gruppe von Klammern, die eine Kombination von Namen der Kindelemente enthält.

### **Gemischter Inhalt (*mixed content*)**

Elemente, die sowohl andere Elemente als auch Zeichendaten enthalten dürfen, haben gemischten Inhalt.

## Element-Inhalt

Die Elemente sind durch **Konnektoren** verbunden und ihnen können **Wiederholungsoperatoren (*repetition operators*)** zur Angabe der Kardinalität nachgestellt werden. Diese Operatoren beschreiben, auf welche Weise Elemente und Text kombiniert werden dürfen.

### ***Konnektoren***

Operator	Anwendung	Bedeutung
,	a , b	Sequenzoperator ( <i>sequence operator</i> )
	a   b	Auswahloperator ( <i>choice operator</i> )

### ***Beispiel DTD-10: Sequenzoperator***

```
<!ELEMENT adresse (name, strasse, ort)>
```

Diese Deklaration besagt: wenn im XML-Dokument das Element `adresse` verwendet wird, dann muss dessen Inhalt mit dem Kindelement `name` beginnen, gefolgt vom Kindelement `strasse`, gefolgt vom Kindelement `ort`:

```

<adresse>
  <name>Lola Blau</name>
  <strasse>Friedrichstr. 23</strasse>
  <ort>10115 Berlin</ort>
</adresse>

```

---

### Beispiel DTD-11 : Auswahloperator

---

```

<!ELEMENT beispiel (grafik | foto | tabelle | liste)

```

Diese Deklaration besagt: wenn im XML-Dokument das Element `beispiel` verwendet wird, muss es alternativ eines der Subelemente `grafik`, `foto`, `tabelle` oder `liste` (genau einmal) enthalten:

```

<beispiel>
  <grafik/>
</beispiel>

```

---

### Wiederholungsoperatoren

<b>?</b>	<b>optionales Element</b>	<b>höchstens einmal</b>
<i>Element kann einmal oder gar nicht vorkommen</i>		
<b>+</b>	<b>erforderliches und wiederholbares Element</b>	<b>mindestens einmal</b>
<i>Element muss mindestens einmal vorkommen</i>		
<b>*</b>	<b>optionales und wiederholbares Element</b>	<b>beliebig</b>
<i>Element kann beliebig oft oder gar nicht vorkommen</i>		
	<b>einmaliges Element</b>	<b>genau einmal</b>
<i>Element muss genau einmal vorkommen</i>		

### **Beispiel DTD-12: Wiederholungsoperator ?**

```
<!ELEMENT kunde (name, vorname, tel, fax?)>
```

Diese Deklaration besagt: wenn im XML-Dokument das Element `kunde` verwendet wird, dann muss dieses Element die Kindelemente `name`, `vorname` und `tel` (in dieser Reihenfolge) enthalten und kann optional im Anschluss auch noch das Element `fax` enthalten. Dies ist sinnvoll, da evtl. nicht jeder Kunde ein Fax hat.

### **Beispiel DTD-13: Wiederholungsoperator +**

---

```
<!ELEMENT kundendatei (kunde+)>
```

Diese Deklaration besagt: wenn im XML-Dokument das Element `kundendatei` verwendet wird, dann muss dieses Element das Subelement `kunde` mindestens einmal enthalten. Darüber hinaus kann es beliebig oft vorkommen. Ein XML-Dokument, das dieser Deklaration genügt wäre z.B.:

```
<?xml version="1.0"?>
<kundendatei>
  <kunde>
    <name>Mann</name>
    <vorname>Dieter</vorname>
    <tel>030/42367899</tel>
  </kunde>
  <kunde>
    <name>Sevenich</name>
    <vorname>Manuela</vorname>
    <tel>030/678999</tel>
    <fax>030/678990</fax>
  </kunde>
</kundendatei>
```

---

### Beispiel DTD-14: Wiederholungsoperator \*

---

```
<!ELEMENT kunde (name, vorname, tel, fax?, email*)>
```

Dieses Beispiel erweitert das Inhaltsmodell für den Elementtyp `kunde` um das Element `email`. Es ist mit dem `*`-Indikator gekennzeichnet, da es vorkommen kann, dass ein Kunde keine, eine oder mehrere Email-Adressen besitzt. Ein gemäß diesem Inhaltsmodell gültiges Element könnte folgendermaßen aussehen:

```
<kunde>
  <name>Mann</name>
  <vorname>Dieter</vorname>
  <tel>030/42367899</tel>
  <email>mann@firma.de</email>
  <email>mann@privat.de</email>
</kunde>
```

---

## Gemischter Inhalt

Ein Element hat **gemischten Inhalt**, wenn Elemente dieses Typs Zeichendaten enthalten dürfen, die optional mit Kindelementen gemischt sind. Gemischter Inhalt lässt sich mit dem Schlüsselwort `#PCDATA` ausdrücken. Das Schlüsselwort `#PCDATA` steht für Zeichendaten (*Parsed Character Data*). Bei Elementen mit gemischtem Inhalt können nur die Typen der Kindelemente beschränkt werden, nicht jedoch ihre Reihenfolge oder ihre Anzahl.

### Beispiel DTD-15: Gemischter Inhalt

```
<!ELEMENT Grabbelsack (#PCDATA | Teil_A | Teil_B)*>
```

Somit müssen die Elemente in einer Inhaltsspezifikation, die gemischten Inhalt zulässt, immer durch den Auswahloperator getrennt werden und die gesamte Gruppe muss durch den `*`-Wiederholungsoperator abgeschlossen werden. Das Schlüsselwort `#PCDATA` darf nur als erstes in der Inhaltsspezifi-

kation vorkommen. Diese Beschränkungen haben ihren Grund darin, dass XML-Prozessoren Steuerzeichen als Zeichendaten interpretieren.

#### **Beispiel DTD-16: Unzulässiges Inhaltsmodell**

---

Somit würde die Deklaration

```
<!ELEMENT abschnitt (titel, #PCDATA)>
```

den Parser veranlassen, die Instanz

```
<abschnitt>¶  
  <titel>Einleitung</titel>  
  Während der letzten Jahrzehnte...  
</abschnitt>
```

für ungültig zu befinden, da auf das Start-Tag <abschnitt> ein Zeilenumbruchzeichen (¶) folgt.

---

Möglich ist auch, dass ein Element nur Zeichendaten enthalten darf. In diesem Falle spricht die XML-Spezifikation jedoch auch von *gemischtem Inhalt*.

#### **Beispiel DTD-17: Nur Zeichendaten als Inhaltsmodell**

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

## Verschachtelung von Inhaltsmodellen

Häufigkeitsoperatoren und Konnektoren können auch auf Elementgruppen angewendet werden. Dadurch können Inhaltsmodelle ineinander verschachtelt werden und komplexere Strukturen ausdrücken.

### **Beispiel DTD-18: Verschachteltes Inhaltsmodell**

---

```
<!ELEMENT Fruchtkorb (Kirsche+, (Apfel | Orange)*)>
```

Dieses Inhaltsmodell besagt, dass ein `Fruchtkorb` mindestens eine `Kirsche` enthalten muss. Es können aber auch noch beliebig viele Kombinationen aus `Apfel` und `Orange` folgen. Zu beachten ist, dass alle Elemente vom Typ `Kirsche` hintereinander auftauchen müssen. Ein Beispiel für einen gültigen `Fruchtkorb` ist:

```
<Fruchtkorb>
  <Kirsche>...</Kirsche>
  <Kirsche>...</Kirsche>
  <Apfel>...</Apfel>
  <Orange>...</Orange>
  <Orange>...</Orange>
</Fruchtkorb>
```

---

# Attributlisten-Deklarationen

In einer Attributlisten-Deklaration werden die Attribute für ein Element festgelegt.

```
<!ATTLIST Elementname Attributdefinition(en)>
```

Die Attributdefinition setzt sich wie folgt zusammen:

```
Attribut-Name + Attribut-Typ + Attribut-Vorgabe
```

## **Beispiel DTD-19: Attributlisten-Deklarationen**

```
<!ATTLIST img
  src      CDATA      #REQUIRED
  alt      CDATA      #REQUIRED
  height   CDATA      #IMPLIED
  width    CDATA      #IMPLIED
>

<!ATTLIST h1 id      ID #IMPLIED>
<!ATTLIST h1 align (left|center|right|justify) "left">
```

## Attribut-Vorgaben

Als Vorgabewerte können sie einen konkreten Wert angeben. Gibt es keinen geeigneten, so können Sie einen Modifikator verwenden, die durch entsprechende Schlüsselwörter repräsentiert werden. Die folgende Tabelle fasst die Möglichkeiten zusammen

### **Attribut-Vorgaben**

"wert"	Standardwert	Ist das Attribut nicht angegeben, setzt der XML-Prozessor es mit dem vorgegeben Wert ein.
#REQUIRED	notwendig	Der Attributwert muss mit dem Element angegeben werden.
#IMPLIED	impliziert	Der Attributwert kann unspezifiziert bleiben.
#FIXED "wert"	fest	Der Attributwert ist festgelegt und kann vom Anwender nicht verändert werden.

### **Beachten Sie:**

Nach dem Schlüsselwort `FIXED` muss der nicht veränderbare Wert im Anschluss gleich mitangegeben werden.

#### **Beispiel DTD-20: Modifikator #FIXED**

```
<!ATTLIST datum jahr CDATA #FIXED "1999">
```

Diese Attributlisten-Deklaration hat für das `<datum>`-Element folgenden Effekt:

- Der XML-Prozessor meldet einen Fehler, wenn im Instanzdokument ein `<datum>`-Element vorkommt, dessen `jahr`-Attribut nicht den Wert 1999 hat.
- Einige nicht-validierende und alle validierenden Parser fügen einem `<datum>`-Element, das kein `jahr`-Attribut besitzt, ein solches hinzu und belegen es mit dem Wert 1999.

## Attribut-Typen

Folgende Attribut-Typen können in einer DTD benutzt werden:

CDATA	Zeichendaten
Aufzählung ( a   b   c )	Eine Aufzählung von Werten, aus der nur ein einziger gewählt werden kann
NMTOKEN	Zeichenkette, die aus beliebigen in XML-Namen zulässigen Zeichen zusammengesetzt ist ( <i>Name Token</i> ).
NMTOKENS	Mehrfache, durch Leerraum separierte Name Token
ID	Ein eindeutiges Elementkennzeichen (Verweisziel für IDREFS)
IDREF	Verweisquelle (Anker) eines Verweises
IDREFS	Verweis auf mehrere Ziele
ENTITY	Ein Entity, das in der DTD deklariert ist
ENTITIES	Mehrfache, durch Leerraum separierte Entities, die in der DTD deklariert sind
NOTATION	Eine Notation, die in der DTD deklariert wird

### CDATA

Als Attributwert kommen alle in XML zugelassenen Zeichendaten in Frage. Entity-Referenzen sind ebenfalls erlaubt.

#### **Beachten Sie:**

- In Attributwerten vom Typ CDATA sind keine Referenzen auf externe Entities erlaubt.
- Das Kleiner-Als-Zeichen (<) ist in Attributwerten nicht erlaubt. Es muss immer mit `&lt;` maskiert werden.

## Aufzählung

Dieser Attribut-Typ ist einer der wichtigsten. Er erlaubt die strengste Kontrolle über die verwendeten Attributwerte.

### **Beachten Sie:**

Alle aufgezählten Werte müssen XML-Namen sein!

## NMTOKEN / NMTOKENS

Der Attributwert ist ein sogenanntes **Name Token** bzw. eine Liste von solchen Name Token. Die einzelnen Token einer Liste sind durch Leerraumzeichen getrennt. Als Wert kommen nur solche Zeichen infrage, die auch in XML-Namen zulässig sind, sogenannte **Name-Zeichen**. Allerdings können diese Zeichen - im Unterschied zu Namen - in beliebiger Reihenfolge auftreten. So ist z.B. 123abc ein mögliches Name Token, aber kein möglicher Name, da Namen nicht mit Zahlen beginnen dürfen.

Beim Attribut-Typ `NMTOKENS` sind mehrere durch Leerraum getrennte Name Token als Attributwerte erlaubt.

### **Beachten Sie:**

Diese Attribut-Typen sind von SGML übernommen und dienen in der Regel der Interoperabilität von XML und SGML-Anwendungen.

### **Beispiel DTD-21: Attribut-Typ NMTOKEN(S)**

```
<!ATTLIST PersonenName
  titel  NMTOKENS  #IMPLIED
  suffix NMTOKEN   #IMPLIED>
```

Ein auf diese Deklaration passendes Element ist:

```
<PersonenName titel="Prof. Dr." suffix="Jr.">
  Peter Richard König
</Personenname>
```

Vorteile der Verwendung von `NMTOKEN(S)` in diesem Beispiel:

- Sie sind nicht eingeschränkt durch vordefinierte Listen von Titeln und Suffixen.
- Sie können mehrere Titel für einen Personennamen nutzen, ohne jede mögliche Kombination definieren zu müssen.

Nachteile der Verwendung von `NMTOKEN(S)` in diesem Beispiel:

- Die fehlende Einschränkung kann auch ein Nachteil sein, da der Attribut-Typ `NMTOKEN(S)` zwar restriktiver ist als `CDATA`, aber doch noch alle möglichen Zeichenkombinationen zulässt, die keine Titel oder Suffixe sind.  
*Folge:* alle weitergehenden Validierungen müssen von der Anwendung gehandelt werden und können nicht vom XML-Prozessor erledigt werden.
- 

## ID

Die Attribut-Typen `ID`, `IDREF` und `IDREFS` sind wie `NMTOKEN` und `NMTOKENS` von SGML vererbt.

Attribute vom Typ `ID` fungieren als eindeutige Bezeichnung für ein Element innerhalb eines Dokuments. Der Wert eines Attributs vom Typ `ID` muss ein XML-Name sein und eindeutig innerhalb des XML-Dokuments, d.h., dass rein numerischen Werte wie z.B. eine Sozialversicherungsnummer oder eine Datensatz-Nummer nicht zulässig sind. Bei der Validierung prüft der XML-Prozessor für jeden Attributwert vom Typ `ID`, ob er eindeutig ist. Der Attribut-Typ `ID` kann sehr nützlich sein, wenn Sie XML als Austauschformat zwischen verschiedenen Datenbanksystemen verwenden.

### *Beispiel DTD-22: Attribut-Typ ID*

---

#### **Attributlistdeklaration**

```
<!ATTLIST mitarbeiter mid ID #REQUIRED>
```

#### **Gebrauch im XML-Dokument**

```
<mitarbeiter mid="m1013">Peter Meier</mitarbeiter>
```

#### **Ungültiger Attributwert (kein XML-Name)**

```
<mitarbeiter mid="1013">Peter Meier</mitarbeiter>
```

Ein Attribut, das als `ID` für ein Element verwendet werden soll, muss mit dem Typ `ID` deklariert werden. Es reicht nicht, ein Attribut namens `ID` zu einem Element hinzuzufügen.

---

**Beachten Sie:**

- Zwei gleiche IDs innerhalb eines XML-Dokuments sind nicht zulässig.
- Ein Attribut vom Typ ID muss nicht den Namen `id` oder `ID` tragen.
- Ein Attribut vom Typ ID muss immer `#REQUIRED` oder `#IMPLIED` als Attribut-Vorgabe haben (`#FIXED` oder ein Literal sind nicht zulässig).
- Pro Element kann es nur ein Attribut vom Typ ID geben.
- Der Attributwert eines Attributs vom Typ ID muss ein XML-Name sein.

**IDREF**

ID-Attribute stellen im Verbund mit IDREF-/IDREFS-Attributen einen einfachen Mechanismus für dokumenteninterne Verweise dar. Dem Element, welches Verweisziel ist, wird ein eindeutiges ID-Attribut zugeordnet. Das Element mit dem IDREF (und entsprechendem Wert) ist die Verweisquelle.

Betrachten wir ein komplettes Beispiel mit DTD:

**Beispiel DTD-23: Attribut-Typ IDREF**

```
<?xml version="1.0"?>
<!DOCTYPE abteilung [
  <!ELEMENT abteilung (mitarbeiter*)>
  <!ELEMENT mitarbeiter (#PCDATA)>
  <!ATTLIST mitarbeiter mid ID #REQUIRED>
  <!ATTLIST mitarbeiter chef IDREF #IMPLIED>
]>

<abteilung>
  <mitarbeiter mid="m1013">Peter Meier</mitarbeiter>
  <mitarbeiter mid="m1014">Dieter Krebs</mitarbeiter>
  <mitarbeiter mid="m1015" chef="m1013">
    Klaus Ohloff
  </mitarbeiter>
  <mitarbeiter mid="m1016" chef="m1014">
    Franz Werfel
  </mitarbeiter>
</abteilung>
```

Alle Mitarbeiter besitzen über das `mid`-Attribut vom Typ ID eine eigene, eindeutige ID, die in der DTD mit dem Schlüsselwort ID und dem Attribut `mid`

definiert wird. Das Attribut `chef` ist vom Typ `IDREF` und darf somit als Attributwert nur die Werte von anderen `ID`-Attributen benutzen.

**Beachten Sie:**

- Verweise, die über `ID/IDREF`-Attribute realisiert werden, gelten nur innerhalb desselben XML-Dokuments.
- Der Wert eines `IDREF`-Attributs muss ein XML-Name sein und mit dem Namen eines `ID`-Attributs übereinstimmen.
- Mehrere `IDREF`-Verweise auf dieselbe `ID` sind nicht zulässig.

**IDREFS**

Ein Mitarbeiter hat in der Regel nur einen (direkten) Chef, kann aber in mehreren Projekten tätig sein. Wollen Sie einen solchen Verweis - von mehreren Projekten auf einen Mitarbeiter - realisieren, benötigen Sie den Attribut-Typ `IDREFS`. Dieser Attribut-Typ ermöglicht es Ihnen, als Attributwert mehrere durch Leerraum getrennte `IDREF`-Attribute zu notieren.

**Beispiel DTD-24: Attribut-Typ `IDREFS`**

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE projektPers [
  <!ELEMENT projektPers (projekte, mitarbeiterListe)>
  <!ELEMENT mitarbeiterListe (mitarbeiter*)>
  <!ELEMENT mitarbeiter (#PCDATA)>
  <!ATTLIST mitarbeiter projekte IDREFS #REQUIRED>
  <!ELEMENT projekte (projekt*)>
  <!ELEMENT projekt (#PCDATA)>
  <!ATTLIST projekt pid ID #REQUIRED>
]>

<projektPers>
  <projekte>
    <projekt pid="p1">Interne Integration</projekt>
    <projekt pid="p2">Servermigration</projekt>
  </projekte>
  <mitarbeiterListe>
    <mitarbeiter projekte="p1 p2">
      Olaf Thomson
    </mitarbeiter>
  </mitarbeiterListe>
</projektPers>
```

**Beachten Sie:**

Für jedes der IDREF-Attribute der IDREFS-Liste gelten natürlich alle Regeln, die wir oben aufgezählt haben.

**ENTITY / ENTITIES**

Der Attributwert ist der Name einer in der DTD definierten nicht analysierten Entity. Attribute vom Typ ENTITY/ENTITIES verknüpfen Elemente mit Objekten der verschiedensten Art (Bilder, Videos, Audio-Files...).

**NOTATION**

Der Attributwert ist der Name einer Notation. Attribute dieses Typs haben den Sinn, eine spezielle Interpretation des Inhalts des betreffenden Elements zu veranlassen. Beispielsweise könnte der Inhalt auf diese Weise einem RTF- oder PS-Interpreter zugeführt werde. Genauso gut könnte das Element Skript-, VRML-, oder Grafikdaten zum Inhalt haben.

**Beispiel DTD-25: Attribut-Typ NOTATION**

```
<!ATTLIST nonxml typ NOTATION (gif | jpg) #REQUIRED>

<nonxml typ="jpg">
  <!-- Base64-kodierte Daten eines JPG-Bilds -->
</nonxml>
```

**Beachten Sie:**

- Der Wert eines Attributs vom Typ NOTATION muss ein XML-Name sein.
- Der Wert eines Attributs vom Typ NOTATION muss mit einer Notation-Deklaration in der DTD übereinstimmen.
- Es kann nur ein NOTATION-Attribut für jeden Elementtyp geben.
- Ein NOTATION-Attribut darf nicht mit leeren Elementen benutzt werden.

*Die Attribut-Typen ENTITY/ENTITIES und NOTATION werden selten genutzt. Wir gehen darauf später im Zusammenhang mit den Entity-Deklarationen noch genauer ein.*

## Normalisierung von Attributwerten

Wie bereits im Modul SYN angedeutet, muss ein XML-Prozessor Attributwerte nach bestimmten Regeln normalisieren. Die Regeln lassen sich im wesentlichen folgendermaßen zusammenfassen:

- Es werden die Regeln zur Behandlung von Zeilenenden angewendet.
- Jedes Leerraumzeichen wird durch ein Leerzeichen (#x20) ersetzt.
- Bei Attributen, die nicht vom Typ CDATA sind, werden zusätzliche Folgen von Leerzeichen (#x20) durch ein einzelnes Leerzeichen ersetzt sowie Leerzeichen am Anfang und Ende des Attributwerts entfernt.

### *Beispiel DTD-26: Normalisierung von Attributwerten*

---

Das Attribut in den folgenden Beispielen sei vom Typ NMTOKENS und der XML-Parser sei ein validierender.

#### **Attribut vor der Normalisierung des Attributwerts**

```
mitarbeiter="    meier
              mueller
              schmitz    "
```

#### **Attribut nach der Normalisierung des Attributwerts**

```
mitarbeiter="meier mueller schmitz"
```

---

#### **Beachten Sie:**

Ein nicht-validierender XML-Prozessor, sollte (muss also nicht!) alle Attributwerte, für die er keine Deklaration gelesen hat, als CDATA behandeln.

Diese Regel bringt es mit sich, dass eine Anwendung in Abhängigkeit vom Attribut-Typ und XML-Prozessor unterschiedliche normalisierte Attributwerte erhalten kann. Dies kann zu Problemen beim Austausch von XML-Dokumenten führen, außer alle Parteien benutzen validierende XML-Prozessoren.

### **Beispiel DTD-27: Unterschiedliche Normalisierung**

---

Das Attribut in den folgenden Beispielen sei vom Typ NMTOKENS. Achten Sie auf die unterschiedlichen Ergebnisse bei einem validierenden und nicht-validierenden XML-Prozessor.

#### **vor der Normalisierung**

```
mitarbeiter="      meier
              mueller
              schmitz      "
```

#### **nach Normalisierung durch validierenden Parser**

```
mitarbeiter="meier mueller schmitz"
```

#### **nach Normalisierung durch nicht-validierenden Parser**

```
mitarbeiter="      meier      mueller      schmitz      "
```

---

## Notation-Deklarationen

Notationen dienen dazu, das Format externer Nicht-XML-Daten mit einem XML-Dokument zu verknüpfen. Notation-Deklarationen stellen einen Namen zur Verfügung, der dann z.B. in Attributwerten verwendet werden kann, um das Format des Element-Inhalts zu spezifizieren. Die weitaus häufigste Verwendung finden solche Notationsnamen in Entity-Referenzen, die direkt auf Daten im definierten Format verweisen (Bilder, Videos...). Einem XML-Prozessor steht es frei, diese Informationen selbst zu benutzen oder sie zu ignorieren. In jedem Fall muss er sie jedoch an die Anwendung weiterreichen, die dann in der Lage sein muss, die Daten entsprechend weiterzuverarbeiten.

Notation-Deklarationen haben eine ähnliche Form wie Dokumenttyp-Deklarationen.

```
<!NOTATION Name SYSTEM System-Identifizier>
```

oder

```
<!NOTATION Name PUBLIC Public-Identifizier (System-Identifizier)>
```

### Beispiel DTD-28: Notation-Deklarationen

```
<!NOTATION jpg SYSTEM "jpgviewer.exe">

<!NOTATION JPEG PUBLIC "ISO/IEC 10918:1993//NOTATION
    Digital Compression and Coding
    of Continuous-tone Still Images
    (JPEG)//EN"
>
```

Die über den **System-Identifizier** angegebene Datei enthält Informationen, die die XML-Anwendung zur Identifikation und ggf. zur Verarbeitung des Datenformats benötigt. Dabei kann es sich um eine Handbuchdatei, einen offiziellen Standard oder um ein Programm handeln. Der System-Identifizier kann auch leer bleiben, wenn Sie einen **Public-Identifizier** angeben. Dies zeigt, dass es nicht Aufgabe der XML-Applikation ist, für die Verarbeitung der Daten zu sorgen. Insbesondere muss eine XML-Applikation nicht selbst zur Verarbeitung der Daten fähig sein. Sie stellt lediglich die formale Integrität der XML-Instanz sicher und identifiziert damit für mögliche Hilfsprogramme die von diesen zu verarbeitenden Daten.

# Entity-Deklarationen

## Allgemeine analysierte Entities

### Interne analysierte Entities

```
<!ENTITY Name "Ersetzungstext">
```

#### *Beispiel DTD-29: Anwendung allgemeine interner Entities*

```
<!ENTITY xml "Extensible Markup Language">  
<definition>XML bedeutet &xml;</definition>
```

#### **Beachten Sie:**

- Kein internes analysiertes Entity kann sich direkt oder indirekt selbst referenzieren

#### *Beispiel DTD-30: Zirkuläre Entity-Deklaration*

```
<!ENTITY entityX "Dies ist &entityX; mit großem X">  
<!ENTITY entitya &entityb; ist sehr schön">  
<!ENTITY entityb &entitya; ist auch nicht übel">
```

- Ein internes analysiertes Entity darf keine Attributwert-Delimiter ( ' und " ) enthalten.

### Externe analysierte Entities

```
<!ENTITY Name SYSTEM System-Identifizier>
```

oder

```
<!ENTITY Name PUBLIC Public-Identifizier System-Identifizier>
```

Die Syntax ist bis auf das Schlüsselwort ENTITY identisch mit der Dokumenttyp-Deklaration. Dies ist nicht verwunderlich, da eine externe DTD-Teilmenge ein Spezialfall einer externen Entity ist.

Mit externen Entities können Sie den Inhalt einer externen Datei durch Angabe eines URI in ein XML-Dokument kopieren.

**Beispiel DTD-31: Anwendungsfall externe Entities**

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

<!DOCTYPE buch [
  <!ENTITY kapitel1 SYSTEM "kapitel1.xml">
  <!ENTITY kapitel2 SYSTEM "kapitel2.xml">
]>

<buch>
  <einleitung>Dies wird ein tolles Buch</einleitung>
  &kapitel1;
  &kapitel2;
  .....
  <fazit>Das war ein tolles Buch</fazit>
</buch>
```

Relative URIs sind zulässig. Relativ meint dabei relativ zu der Ressource, in der das Entity deklariert wird.

Ein externes analysiertes Entity kann mit einer sogenannten **Text-Deklaration** beginnen. Eine solche Text-Deklaration sieht aus wie eine XML-Deklaration, mit dem einzigen Unterschied, dass das `version`-Attribut optional ist. Über das (notwendige) `encoding`-Attribut einer Text-Deklaration hat man die Möglichkeit, Inhalte mit anderer Kodierung als das importierende Dokument einzubinden.

**Beispiel DTD-32: Text-Deklaration**

```
<?xml encoding="iso-8859-1"?>

<kapitel nr="1">
  <titel>Einleitung</titel>
  <text>
    Hier steht der Text von Kapitel 1, in dem auch Umlaute wie Ä, Ö oder Ü auftauchen können, die zum ISO-Latin-1-Zeichensatz gehören.
  </text>
</kapitel>
```

**Beachten Sie:**

- Ein externes analysiertes Entity, das nicht in der Standardkodierung UTF-16 oder UTF-8 gespeichert ist, *muss* eine Text-Deklaration mit einer Kodierungsdeklaration enthalten.
- Externe, analysierte Entities dürfen keine Dokumenttyp-Deklaration enthalten.

## Nicht-analyisierte Entities

Nicht-analyisierte Entities sind immer externe Entities.

```
<!ENTITY Name SYSTEM System-Identifizier NDATA Name>
```

oder

```
<!ENTITY Name PUBLIC Public-Identifizier System-Identifizier NDATA Name>
```

Die Syntax ist identisch zu der externer Entities bis auf die bei nicht-analyisierten Entities notwendige Angabe der zu der Entity gehörigen Notation. Dies geschieht über des Schlüsselwort NDATA (*not XML-Data*) und den Namen der Notation.

**Beispiel DTD-33: Anwendungsfall nicht analysierte Entities**

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

<!DOCTYPE produktdatenbank [
  <!NOTATION gif SYSTEM "iexplorer.exe">
  <!ENTITY handy SYSTEM "../img/handy.gif" NDATA gif>
  <!ELEMENT produktdatenbank (datensatz)+>
    <!ELEMENT datensatz (name, bild)>
      <!ELEMENT name (#PCDATA)>
      <!ELEMENT bild (#PCDATA)>
      <!ATTLIST bild quelle ENTITY #REQUIRED>
]>

<produktdatenbank>
  <datensatz>
    <name>Mobiltelefon</name>
    <bild quelle="handy"/>
  </datensatz>
</produktdatenbank>
```

Ein nicht-analisiertes Entity darf nur in Attributwerten referenziert werden. Dieses Attribut ist in der DTD mit dem Typ `ENTITY` oder `ENTITIES` deklariert. Im obigen Beispiel wird ein Bild im GIF-Format eingebunden. Achten Sie darauf, dass in Attributen kein Und-Zeichen (&) vorkommen darf! Die Referenzierung in Attributwerten erfolgt also nicht - wie man vermuten könnte - über die gängige Syntax mittels `&name;`. Stattdessen wird nur der Name des nicht-analysierten Entity einfach als Attributwert in Anführungszeichen angegeben.

### **Alternativen zur Einbindung binärer Daten**

Der Weg, binäre Daten in XML über nicht-analisierte Entities einzubinden ist recht komplex und man benötigt immer eine DTD. Angesichts der Tatsache, dass die Verarbeitung letztlich doch der Anwendung überlassen bleibt, werden in der Praxis einfachere Mechanismen zur Einbindung binärer Daten genutzt.

#### ***Einbindung durch Referenzierung über URI***

Der einfachste Weg ist die Referenzierung der binären Daten über einen URI. Der URI kann z.B. der Attributwert eines leeren Elements sein, das als Platzhalter für die binären Daten fungiert.

##### ***Beispiel DTD-34: Einbindung binärer Daten über Referenzierung per URI***

```
<bild quelle="../img/portrait.jpg"/>
```

Als zusätzliches Attribut kann der MIME-Typ angegeben werden:

##### ***Beispiel DTD-35: MIME-Typ-Angabe für binäre Daten***

```
<bild mimeType="image/jpeg" quelle="/img/portrait.jpg"/>
```

#### ***Einbettung durch Codierung in base64***

Bei diesem Verfahren werden die binären Daten base64-codiert direkt in das XML-Dokument geschrieben. Zusätzlich kann auch hier der MIME-Typ als Attribut des Container-Elements angegeben werden.

##### ***Beispiel DTD-36: Einbettung base64-codierter binärer Daten***

```
<bild mimeType="image/jpeg">  
  /9j/4AAQSkZJRgABAQAAQABAAAD/2wBDAAoHBwgHBgoICAgLCgoLDh  
  gQDg0NDh0VFhEYIx8lJCIfIiEmKzcJik0KSEiMEExNDk7Pj4+JS5ES  
  UM8SDc9Pjv/2wBDAQoLCw4NDhwQEBw7KCIoOzs7Ozs7Ozs7Ozs  
  7Ozs7Ozs7Ozs7Ozs7Ozs7Ozs7Ozs7Ozs7Ozs7Ozs7Ozs7Ozs7Ozv/wAARC  
  ABzAHMDASIAAhEBAxEB/8QAHwAAAQUBAQEBAQEAAAAAAAAAAAECAwQ  
</bild>
```

Ein ähnliches Verfahren bietet unter Verwendung einer DTD die Kombination aus Notation-Deklaration und Attribut vom Typ NOTATION, die wir ja bereits kennen gelernt haben.

**base64** ist ein textbasiertes Datenformat, das zur Codierung von Binärdaten dient. Es dient zur Einbettung von binären Daten in textbasierte Übertragungsformate (Email, XML). base64 nutzt zu diesem Zweck einen eingeschränkten Zeichensatz aus 64 Zeichen, die alle gängigen Zeichensätze "verstehen". Für XML ist dieser Zeichensatz v.a. deswegen geeignet, weil er keine Markup-Symbole wie < oder > beinhaltet. Mittlerweile bietet fast jeder XML-Parser einen base64-Encoder und -Decoder.

**URL:**

<http://www.ietf.org/rfc/rfc2045.txt>

## Parameter-Entities

Parameter-Entities sind immer analysierte Entities.

### Interne Parameter-Entities

```
<!ENTITY % Name "Ersetzungstext">
```

*Parameter-Entities werden nur in DTDs verwendet!*

Ein häufiger Anwendungsfall besteht darin, Inhaltsmodelle, die für mehr als einen Elementtyp verwendet werden, unter einem Namen zusammenzufassen.

#### **Beispiel DTD-37: Anwendung Parameter-Entities**

```
<!ENTITY % block "absatz | listing | abbildung | kasten">

<!ELEMENT buch (kapitel+)>
<!ELEMENT kapitel (ueberschrift, (%block;)*, abschnitt+)>
<!ELEMENT abschnitt (ueberschrift, (%block;)*, unterabschnitt+)>
<!ELEMENT unterabschnitt (ueberschrift, (%block;)+)>
```

### **Beachten Sie:**

- Parameter-Entities werden mit %-Zeichen deklariert und referenziert.
- Parameter-Entities müssen vor ihrer Referenzierung deklariert werden.
- Parameter-Entity-Referenzen dürfen in einer internen Teilmenge nicht innerhalb von Markup-Deklarationen auftauchen.

### **PRAXIS-TIPP**

Schauen Sie sich einmal eine der XHTML-DTDs (z.B. unter <http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd>) an. Dort werden verstärkt interne Parameter-Entities eingesetzt, um die DTD zu normalisieren und damit lesbarer und wartbarer zu machen.

### **Externe Parameter-Entities**

Externe Parameter-Entities werden analog zu allgemeinen, externen, analysierten Entities deklariert, allerdings mit zusätzlichem %-Zeichen.

```
<!ENTITY % Name SYSTEM System -Identifizier>
```

oder

```
<!ENTITY % Name PUBLIC Public-Identifizier  
SYSTEM System-Identifizier>
```

Während interne Parameter-Entities in der Regel dazu dienen wiederkehrende Attributlisten oder Inhaltsmodelle zusammenzufassen, so lassen sich über externe DTDs einzelne DTD-Module bilden oder allgemeine Entity-Deklarationen in separate Dateien auslagern.

Ein Beispiel für die Verwendung von externen Parameter-Entities bietet XHTML 1.1. Mit der Version 1.1 wurde XHTML modularisiert, d.h. Software-Hersteller und XHTML-Anwender können sich ihr eigenes XHTML aus verschiedenen Modulen zusammenstellen. In der Regel benötigen nämlich sowohl Software-Hersteller als auch XHTML-Autoren nicht den gesamten Vorrat an HTML-Tags. Mit der Modularisierung können sie nun Teilmengen herauspicken, z.B. für Browser in Mobiltelefonen.

### **Mehr Informationen zum Thema finden Sie unter:**

<http://www.w3.org/TR/xhtml11/>

**und**

<http://www.w3.org/TR/xhtml-modularization/>

## Regeln für die Referenzierung von Entities

- Entities müssen deklariert werden (Ausnahme: vordefinierte)
- Werden mehrere Entities mit gleichem Namen deklariert, ist die erste bindend.
- Referenzen auf extern-analyisierte, allgemeine Entities dürfen nicht in Attributwerten auftauchen.
- Ein analysiertes Entity darf weder direkt noch indirekt eine rekursive Referenz auf sich selbst enthalten.
- Referenzen auf nicht-analyisierte Entities dürfen nur als Wert eines Attributs auftauchen, das vom Typ `ENTITY` oder `ENTITIES` ist.
- Referenzen auf Parameter-Entities werden nur in der DTD erkannt, in einem XML-Dokument werden sie wie Text ausgegeben.
- Referenzen auf Parameter-Entities können in einer internen Teilmenge nicht innerhalb von Markup-Deklarationen auftauchen.

## Klassifizierung von Entities (Zusammenfassung)

Jedes Entity lässt sich anhand dreier Kriterien klassifizieren:

- allgemeines oder Parameter-Entity
- intern oder extern
- analysiert oder nicht analysiert

Aus den drei Merkmalen lassen sich theoretisch  $2*2*2=8$  Entity-Arten kombinieren. Tatsächlich gibt es aber wie wir gesehen haben nur 5. Folgende drei Entity-Arten kommen niemals vor:

- nicht-analyisierte, interne Parameter-Entities
- nicht-analyisierte, externe Parameter-Entities
- allgemeine, interne, nicht-analyisierte Entities

Mit anderen Worten: Parameter-Entities sind immer analysiert und (allgemeine) nicht-analyisierte Entities sind immer extern.

Zusätzlich gibt es die vordefinierten Entities als Escape-Zeichen für die Markup-Symbole.

## Vorteile der Modularisierung mittels Entities

Wir haben gesehen, wie Sie mittels Entities sowohl Ihre XML-Instanzdokumente als auch ihre DTDs modularisieren können. Modularisierung bedeutet die Eliminierung von Redundanzen und damit etwas, das mit Fachjargon Normalisierung genannt wird. Dies hat v.a. folgende Vorteile:

- Wiederverwendung mehrfach auftauchender Bestandteile (Module)
- leichtere Wartbarkeit
- Möglichkeit der Arbeitsteilung (gleichzeitiges Arbeiten an einzelnen Modulen)
- Flexibilität und leichte Anpassbarkeit (insbesondere von DTDs) durch Austausch von Modulen bzw. Erweiterung um neue Module.

# Entity Management

Das Referenzieren von Entities über System-Identifizier - also die direkte Angabe des Ortes in der Entity-Deklaration - bringt ein Interoperabilitätsproblem mit sich. Die Angaben sind nämlich in der Regel nicht auf einen anderen Rechner portierbar. Auch eine Auslagerung der Entities auf einen von überall her zugänglichen Web-Server ist nicht unproblematisch, da dabei eine offene Internet-Verbindung vorausgesetzt wird.

Eine Lösung dieses Problems bietet die Trennung des Namens einer Resource von der Adresse. Dies lässt sich mittels der Public-Identifizier und eines Katalogs (**Entity-Resolution-Katalog**), der Public-Identifizier auf Speicherorte abbildet, gewährleisten. Diese Praxis wird auch **Entity Management** genannt.

## Einheitliche Syntax für Public-Identifizier

Die XML-Empfehlung unterstützt Public-Identifizier, aber definiert kein Format für sie. Eine übliche Praxis ist, sich an die Syntax für **Formal Public Identifier (FPI)** aus SGML zu halten. Portabilität ist dadurch allerdings auch nicht garantiert.

Ein alltägliches, aber oft nicht wahrgenommenes Beispiel ist der FPI in der Dokumenttyp-Deklaration eines HTML-Dokuments:

*Beispiel DTD-38: FPI in der Dokumenttyp-Deklaration eines HTML-Dokuments*

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0 Strict//EN">
```

Folgende Abbildung zeigt die Bedeutung der einzelnen Teile eines FPI:



### ***Implementierungen von Entity-Resolvern***

<b>Name</b>	<b>Entwickler</b>	<b>URL</b>
xml-commons-resolver	Apache Group	<a href="http://xml.apache.org/dist/commons/">http://xml.apache.org/dist/commons/</a>
Entity Resolver	Sun	<a href="http://www.sun.com/software/xml/developers/resolver/">http://www.sun.com/software/xml/developers/resolver/</a>

## Bedingte Abschnitte

Innerhalb einer *externen* DTD können Sie mit der Direktive `IGNORE` eine Gruppe von Deklarationen bilden, die ignoriert werden sollen:

```
<![IGNORE [Deklarationen]]>
```

Umgekehrt können Sie Deklarationen explizit in Ihre DTD einbinden:

```
<![INCLUDE [Deklarationen]]>
```

### *Beispiel DTD-39: Anwendungsfall I für bedingte Abschnitte*

---

Wozu das gut sein soll, lässt sich an folgendem Szenario am besten erläutern:

Stellen Sie sich vor, Sie schreiben einen umfangreichen Text, z.B. einen längeren Bericht. In der Entwurfsphase können Sie z.B. Erläuterungstexte hinzufügen, für die Sie einen Elementtyp `kommentar` deklarieren.

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

Um Kommentare nun vor Abschluss der Arbeit zu löschen, setzen Sie die Deklaration des Elements auf `IGNORE`.

```
<![IGNORE[<!ELEMENT kommentar (#PCDATA)]]>
```

Enthält ihr Dokument nun noch ein Element vom Typ `kommentar`, so meldet der XML-Prozessor Ihnen einen Fehler. Auf diese Weise können Sie sicher gehen, keinen Kommentar zu vergessen.

---

Eine weitere, ähnliche Anwendungsmöglichkeit ist die Verwendung einer "lockeren" DTD während des Schreibens, wenn z.B. noch nicht der gesamte Inhalt eingegeben wurde, sie aber das Dokument trotzdem auswerten wollen. Für die Endversion verwenden Sie dann eine weitere strengere DTD. Sie

können nun beide Deklarationsvarianten der DTD in Blöcke verpacken und jeden Block mit einem Verweis auf ein Parameter-Entity versehen. Die Parameter-Entities sind dabei als die Schlüsselwörter `IGNORE` und `INCLUDE` deklariert. Jedes Dokument, das nun diese (externe) DTD referenziert, kann nun die Entity-Deklarationen in der internen Teilmenge so setzen wie gewünscht.

### **Beispiel DTD-40: Anwendungsfall II für bedingte Abschnitte**

---

#### **Externe DTD**

```
<![%entwurf;[
    <!ELEMENT kommentar (#PCDATA)>
]]>

<![%fertig;[
    <!ELEMENT kommentar EMPTY>
]]>
```

#### **XML-Dokument**

```
<!DOCTYPE wurzelement SYSTEM "meineDTD.dtd" [
    <!ENTITY % entwurf 'INCLUDE'>
    <!ENTITY % fertig 'IGNORE'>
]>
<wurzelement>
    ...
</wurzelement>
```

---

#### **Vorteile dieses Vorgehens:**

- Jedes XML Dokument kann für sich festlegen, ob es Entwurf oder Endversion ist, allgemeiner formuliert, welche Version der DTD es verwenden will.
- Die Namen der Parameter-Entities sind selbstdokumentierend und reflektieren somit die Verwendung des jeweiligen bedingten Abschnitts.

#### **Hinweis:**

Von der soeben demonstrierten Verwendungsweise her haben die Parameter-Entities ihren Namen.

Architektonisch sehr lehrreich ist in diesem Zusammenhang die Betrachtung der Modularisierung von XHTML 1.1. Hier wird reger Gebrauch von bedingten Abschnitten gemacht. Mittels des oben beschriebenen "Umschaltens" Parameter-Entities kann man die für eigene Bedürfnisse notwendigen Module "anschalten".

**Mehr Informationen zum Thema finden Sie unter:**

*<http://www.w3.org/TR/xhtml11/>*

**und**

*<http://www.w3.org/TR/xhtml-modularization/>*

## Wann benötigen Sie eine DTD?

Sie benötigen eine DTD, wenn Sie

- Vorgabewerte für Ihre Attribute verwenden,
- Entities nutzen,
- Ihre Dokumente validieren

möchten.

### **Beachten Sie:**

- Auch nicht-validierende Parser müssen die interne Teilmenge einer DTD bearbeiten, d.h. sie müssen die Markup-Deklarationen dort nutzen, um Entity-Referenzen aufzulösen, Attributwerte zu normalisieren und feste Werte als Attributvorgaben einzusetzen.
- Nicht-validierende Parser müssen allerdings keine externen Parameter-Entities lesen, auch wenn sie in der internen Teilmenge stehen.
- Nicht-validierende Parser können jedoch auch die externe Teilmenge lesen und auswerten.

# Globalität bei Elementen und Attributen

## Element-Deklarationen sind global

Beachten Sie, dass alle Element-Deklarationen global sind. Das bedeutet, dass Sie deklarierte Elemente in anderen Element-Deklarationen wiederverwenden können. So kann Sie z.B. ein deklariertes Element `<name>` in verschiedenen Inhaltsmodellen anderer Element-Deklarationen referenziert werden.

### **Beispiel DTD-41: Globalität von Element-Deklarationen**

```
<!ELEMENT auto (name, baujahr)>
<!ELEMENT besitzer (name, geburtsjahr)>

<!ELEMENT baujahr (#PCDATA)>
<!ELEMENT geburtsjahr (#PCDATA)>
<!ELEMENT name (#PCDATA)>
```

Beachten Sie auch, dass die Reihenfolge der Deklarationen beliebig ist. Obwohl das `<name>`-Element an letzter Stelle deklariert wird, referenzieren die Inhaltsmodelle der Element `<auto>` und `<besitzer>` bereits auf das Element.

Neben dieser globalen Deklaration gibt es jedoch keine Möglichkeit, Elemente lokal zu deklarieren. Dies bedeutet, ein deklariertes Element `<name>` kann nicht in dem einen Inhaltsmodell eine andere innere Struktur haben als im anderen. Folgendes Beispiel wäre also keine wohlgeformte DTD:

### **Beispiel DTD-42: Unzulässige Deklaration mehrerer Elemente mit gleichem Namen**

```
<!ELEMENT auto (name, baujahr)>
  <!ELEMENT name (#PCDATA)>
  <!ELEMENT baujahr (#PCDATA)>
<!ELEMENT besitzer (name, geburtsjahr)>
  <!ELEMENT name (vorname, nachname)>
    <!ELEMENT vorname (#PCDATA)>
    <!ELEMENT nachname (#PCDATA)>
  <!ELEMENT geburtsjahr (#PCDATA)>
```

Auch wenn die Einrückungen im obigen Beispiel eine Lokalität der Element-Deklarationen suggerieren, muss ein XML-Prozessor hier einen Fehler melden.

## Attributlisten-Deklarationen sind lokal

Attribute werden dagegen immer lokal, d.h. in Verbindung mit dem jeweiligen Element, deklariert.

### **Beispiel DTD-43: Lokalität von Attributlisten-Deklarationen**

```
<!ELEMENT auto EMPTY>
  <!ATTLIST auto name CDATA #REQUIRED
                baujahr CDATA #REQUIRED>
```

Es ist somit nicht möglich, ein globales Attribut `name` zu deklarieren und es in mehreren Element-Deklarationen wiederzuverwenden. Andererseits können sie aber - anders als bei Element-Deklarationen - für verschiedene Elemente mehrere Attribute `name` deklarieren, die z.B. alle einen unterschiedlichen Attribut-Typ besitzen.

### **Beispiel DTD-44: Deklaration mehrerer Attribute mit gleichem Namen**

```
<!ELEMENT auto EMPTY>
  <!ATTLIST auto name CDATA #REQUIRED
                baujahr CDATA #REQUIRED>
<!ELEMENT besitzer EMPTY>
  <!ATTLIST besitzer name ID #REQUIRED>
                geburtsjahr CDATA #REQUIRED>
```

Beachten Sie, dass mehrere Attribute gleichen Namens für *ein* Element natürlich unzulässig sind.

Um gleiche Attribute verschiedener Elemente nicht immer wieder aufs Neue für jedes Element deklarieren zu müssen, können Sie Parameter-Entities verwenden. Das Verfahren ist parallel zu der Wiederverwendung von gleichen Inhaltsmodellen, nur dass sie hier wiederkehrende Attributlisten als Entity deklarieren.

### **Beispiel DTD-45: Parameter-Entities und Attributlisten-Deklarationen**

```
<!ENTITY % kernattr
  "id ID #IMPLIED
  name CDATA #IMPLIED"
>

<!ELEMENT auto (#PCDATA)>
  <!ATTLIST auto %kernattr;>
<!ELEMENT besitzer (#PCDATA)>
  <!ATTLIST besitzer %kernattr;>
```