Verwirrende Vielfalt

Modellierungsvarianten mit XML Schema

Erik Wilde

XML Schema bietet im Gegensatz zu DTDs eine Vielzahl an Modellierungsfeatures und damit realisierbaren Varianten. Oft ist nicht klar, auf welche Weise ein gegebenes Modell am besten als XML Schema umgesetzt werden sollte. In diesem und einem nachfolgenden Artikel wird deshalb der Frage nachgegangen, welche Varianten es gibt, wie sie sich unterscheiden, was ihre Vor- und Nachteile sind, und wie sie sich insbesondere unter dem Blickpunkt der Wiederverwendung und Erweiterbarkeit bewerten lassen.

XML Schema ist als Ablösung für die in XML eingebauten DTDs geschaffen worden. Die zwei wichtigsten Neuerungen von XML Schema sind zum einen die reichhaltige Bibliothek an simplen Datentypen (Teil 2 des Standards, der uns hier nicht weiter beschäftigen wird), und zum anderen die Einführung einer Typschicht, was gegenüber DTDs eine komplett neue Modellierungsebene ergibt (siehe Abbildung 1).

Die Typebene von XML Schema wurde in DTDs oftmals mit Parameter Entities (Entities, die nur in DTDs

	DTD		XML Schema	
	Example	**Terminology**	**Terminology**	**Example**
Schema	`<!ELEMENT test EMPTY>`	Element Type Declaration	Element Declaration	`<xs:element name="test" type="testType"/>`
			Type Definition (Simple or Complex)	`<xs:complexType name="testType"/>`
XML	`<test/>`	Element	Element	`<test/>`

Abbildung 1: Modellierungsebenen von DTDs und XML Schema im Vergleich

verwendbar sind, und als `%name;` referenziert werden) „simuliert“, jedoch war dies immer nur ein sehr bescheidener und auf viel Selbstdisziplin aufbauender Weg, eine gewisse Systematik und Wiederverwendbarkeit in DTDs erreichen zu können. In XML Schema ist die Typebene ein wichtiger Bestandteil der Sprache und erlaubt Wiederverwendung (mehrere Elemente verwenden den gleichen Typ), Erweiterung (ein Typ erweitert einen bestehenden Typ), und Einschränkung (ein Typ schränkt einen bestehenden Typen ein).

Unterschiede in der Verarbeitung

Damit wären wir auch schon bei einem sehr wichtigen Unterschied zwischen der Verarbeitung von DTD und XML Schema basiertem XML: während DTD XML meist einfach basierend auf einem DOM-Modell verarbeitet wird, findet die Verarbeitung von XML Schema basiertem XML idealerweise mit einem Toolset statt, das Typinformationen bietet, zu Elementen und Attributen also gleich auch immer noch die Information liefert, welchem Typ sie angehören, so dass die eigentliche Verarbeitung typbasiert stattfinden kann. Hier stossen wir dann auch gleich an die Grenzen bestehender Technologien: zwar kommt man über das in DOM3 eingeführte Interface `TypeInfo` an die Typinformationen zu einem Element oder Attribut, aber es gibt noch kein DOM Modul, mit dem man dieser Typinformation dann auf dem XML Schema Modell folgen könnte, um z.B. festzustellen, auf welchem Typ der gefundene Typ basiert. Hier ist man noch auf proprietäre Interfaces angewiesen wie das *Xerces Schema Component Model API* oder IBM’s XSD.

In der Realität sieht es daher so aus, dass im absoluten Grossteil aller Anwendungen nicht sauber typbasiert gearbeitet wird, sondern die Typinformation als fest verdrahtet mit den Elementen angesehen wird. Diese

Annahme wird uns besonders im zweiten Artikel beschäftigen, wenn es darum geht, XML Schemas so zu gestalten, dass sie (und die darauf aufbauende Software) langlebig und wiederverwendbar sind. Zunächst einmal wollen wir uns jedoch auf XML Schema selber beschränken und genauer anschauen, in welcher Weise die Element/Typ-Trennung (die im übrigen genauso für Attribute gilt) Auswirkungen auf die Modellierung hat.

Lokale und Globale Definitionen

Sowohl Elemente/Attribute als auch Typen können in XML Schema lokal oder global definiert werden. Lokale Definitionen treten innerhalb anderer Definitionen auf (Elemente/Attribute in Typen oder Named Groups, Typen in Elementen oder Attributen), globale Definitionen dagegen sind auf dem Top Level des Schemas definiert, tragen einen Namen, und werden an anderer Stelle referenziert. Was sind die Unterschiede dieser Konzepte?

- *Lokale Definitionen.* Diese Definitionen kommen innerhalb anderer Definitionen vor, bei Elementen z.B. in Typen oder Named Model Groups, bei Typen in Elementen oder Attributen. Lokale Definitionen können nicht wiederverwendet werden (da sie keinen referenzierbaren Namen tragen), machen ein Schema oftmals aber besser lesbar (Schema Design Tools relativieren diesen Punkt allerdings, da sie oftmals auch globale Definitionen gut erkennbar in eine graphische Darstellung des Schemas einbeziehen).
- *Globale Definitionen.* Diese Definitionen treten immer auf dem Top Level des Schema auf, sind also eigenständig (d.h. nicht in andere Definitionen eingebettet) und über ihren Namen referenzierbar.

Wichtig ist neben der Referenzierbarkeit zu verstehen, dass sich insbesondere bei Elementen und Attributen die Frage lokal oder global auf die Namensgebung (in den Dokumenten!) auswirkt. Während globale Namen in einem Dokument immer vollqualifiziert erscheinen müssen (also mit dem *Namespace Prefix*, sofern das Schema einen *Target Namespace* deklariert), wird dies bei lokalen Namen über die `elementFormDefault` und `attributeFormDefault` Attribute des Schemas gesteuert (die als Default unqualifizierte Namen verlangen). Dies nur als Vorbemerkung, mehr zur Frage von Namespaces und ihrer Verwendung in XML Schema wird es im nächsten Artikel zu lesen geben.

Lustiges Permutieren

Ausgehend von lokalen und globalen Elementen und Typen lassen sich nun verschiedene Design-Varianten definieren. Basierend auf folgendem kleinen Beispiel-Dokument sollen diese Varianten beschrieben und diskutiert werden.

```
<person>
  <name>
    <givenname>Erik</givenname>
    <givenname>Thomas</givenname>
    <surname>Wilde</surname>
  </name>
  <address>
    <company>ETH Zürich</company>
    <email>net.dret@dret.net</email>
  </address>
</person>
```

- *Russian Doll.* Dieses Schema schachtelt alles ineinander soweit möglich, aus diesem Grund gibt es nur eine einzige globale Definition, und alle anderen Elemente und Typen sind als lokale Definitionen darin enthalten. Vorteil dieses Schemas ist, dass man dem Schema selber die Struktur der Instanzen sehr gut ansehen kann. Nachteile sind die grosse Schachtelungstiefe und die prinzipielle Unmöglichkeit, rekursive (also in sich selbst geschachtelte) Strukturen auf diese Art abbilden zu können.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="person">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="name">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="givenname" type="xs:token" maxOccurs="unbounded"/>
              <xs:element name="surname" type="xs:string"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="address" minOccurs="0">
```

```
          <xs:complexType>
            <xs:sequence>
              <xs:element name="company" type="xs:string"/>
              <xs:element name="email" minOccurs="0">
                <xs:simpleType>
                  <xs:restriction base="xs:string">
                    <xs:pattern value=".*@.*\..*"/>
                  </xs:restriction>
                </xs:simpleType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

- *Garden of Eden.* Dies ist der genaue Gegenentwurf zur Russian Doll. Wie im Garten Eden erhält alles einen Namen, wird also global definiert. Auf diese Weise sind sowohl Elemente/Attribute als auch Typen wiederverwendbar, aber das Schema wird recht voluminös und unübersichtlich (zu erkennen an den vielen `ref` und `type` Attributen). Bei Verwendung eines Schema Design Tools wird die Unübersichtlichkeit durch eine graphische Oberfläche oftmals versteckt, aber es lässt aich auf jeden Fall feststellen, dass wir es hier mit einem in seiner Philosophie extremen Schema zu tun haben.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="person" type="personType"/>
  <xs:element name="name" type="nameType"/>
  <xs:element name="givenname" type="givennameType"/>
  <xs:element name="surname" type="surnameType"/>
  <xs:element name="address" type="addressType"/>
  <xs:element name="company" type="companyType"/>
  <xs:element name="email" type="emailType"/>
  <xs:complexType name="personType">
    <xs:sequence>
      <xs:element ref="name"/>
      <xs:element ref="address" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="nameType">
    <xs:sequence>
      <xs:element ref="givenname" maxOccurs="unbounded"/>
      <xs:element ref="surname"/>
    </xs:sequence>
  </xs:complexType>
  <xs:simpleType name="givennameType">
    <xs:restriction base="xs:token"/>
  </xs:simpleType>
  <xs:simpleType name="surnameType">
    <xs:restriction base="xs:string"/>
  </xs:simpleType>
  <xs:complexType name="addressType">
    <xs:sequence>
      <xs:element ref="company"/>
      <xs:element ref="email" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
  <xs:simpleType name="companyType">
    <xs:restriction base="xs:string"/>
  </xs:simpleType>
  <xs:simpleType name="emailType">
    <xs:restriction base="xs:string">
      <xs:pattern value=".*@.*\..*"/>
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

- *Salami Slice.* Dieses Design definiert alle Elemente als global, verwendet für ihre Definition aber jeweils lokale Typen. Auf diese Weise sind die Elemente als Komponenten wiederverwendbar, die Typen aber nicht. Seinen Namen trägt es, weil die einzelnen Elementdefinitionen wie Salamischeiben nebeneinander aufgereiht sind.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="person">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="name"/>
        <xs:element ref="address" minOccurs="0"/>
      </xs:sequence>
```

```
    </xs:complexType>
  </xs:element>
  <xs:element name="name">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="givenname" maxOccurs="unbounded"/>
        <xs:element ref="surname"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="givenname" type="xs:token"/>
  <xs:element name="surname" type="xs:string"/>
  <xs:element name="address">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="company"/>
        <xs:element ref="email" minOccurs="0"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="company" type="xs:string"/>
  <xs:element name="email">
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:pattern value=".*@.*\..*"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:element>
</xs:schema>
```

- *Venetian Blinds.* Die letzte Design-Permutation dreht das Salami Slice Design sozusagen auf den Kopf, anstatt sich auf Elemente als wiederverwendbare Komponenten zu konzentrieren, werden alle Typen konsequent global definiert, während die Elemente innerhalb der Typen dann lokal definiert werden. Folglich gibt es nur ein global definiertes Elemente, nämlich das Document Element.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="person" type="personType"/>
  <xs:complexType name="personType">
    <xs:sequence>
      <xs:element name="name" type="nameType"/>
      <xs:element name="address" type="addressType" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="nameType">
    <xs:sequence>
      <xs:element name="givenname" type="givennameType" maxOccurs="unbounded"/>
      <xs:element name="surname" type="surnameType"/>
    </xs:sequence>
  </xs:complexType>
  <xs:simpleType name="givennameType">
    <xs:restriction base="xs:token"/>
  </xs:simpleType>
  <xs:simpleType name="surnameType">
    <xs:restriction base="xs:string"/>
  </xs:simpleType>
  <xs:complexType name="addressType">
    <xs:sequence>
      <xs:element name="company" type="companyType"/>
      <xs:element name="email" type="emailType" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
  <xs:simpleType name="companyType">
    <xs:restriction base="xs:string"/>
  </xs:simpleType>
  <xs:simpleType name="emailType">
    <xs:restriction base="xs:string">
      <xs:pattern value=".*@.*\..*"/>
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

In der Praxis ist es häufig so, dass keines dieser Design-Muster hundertprozentig angewendet wird. Jedoch ist es wichtig, sich dieser Design-Alternativen und ihrer Konsequenzen bewusst zu sein. Dies wird insbesondere wichtig im Zusammenhang mit der Frage, ob ein Schema wiederverwendet wird als Basis für ein darauf aufbauendes Schema, oder als Grundlage für eine Weiterentwicklung. In beiden Fällen ist es wichtig, darauf achtzugeben, dass die Elemente und Typen ja nicht nur in ihrer einfachsten Form zusammenhängen (also als Standalone-Typen, die als Grundlage für Elemente oder Attribute benutzt werden), sondern XML Schema eine

ganze Anzahl weiterer Wechselbeziehungen zulässt, die eben insbesondere beim Zusammenspiel mehrerer Schemas interessant werden.

Weitergehende Beziehungen

Was sind nun diese weitergehenden Beziehungen? Und wie kann man sie benutzen? Essentiell ist zu verstehen, dass all diese Beziehungen per Default erlaubt sind, will man sie also nicht verwenden bzw. ihre Verwendung auch für spätere Anwendungen des Schemas verbieten (warum man das machen will und warum es es oft sogar machen sollte, dazu im nächsten Artikel mehr), so muss man das explizit tun. Doch schauen wir zuerst einmal diese Beziehungen an.

- *Typ-Verwendung.* Typen werden als Grundlage für Elemente oder Attribute verwendet. Manche Typen sollen aber nicht direkt für Elemente oder Attribute verwendet werden, sondern nur als Grundlage für weitere, von ihnen abgeleitete Typen. In diesem Fall kann man einen Typen als `abstract` deklarieren, was bedeutet, dass dieser Typ nicht für Elemente oder Attribute verwendet werden darf. In der Praxis bedeutet dies, dass ein solcher Typ niemals in einem Dokument erscheinen wird.
- *Typ-Ableitung.* Typ-Ableitung in XML Schema kennt Typ-Erweiterung und -Einschränkung für komplexe Typen, und Typ-Einschränkung, Listen und Unions für Simple Typen. Typ-Ableitung kann man per `final` pauschal verbieten, oder nur spezielle Arten, so dass man z.B. einen Typ deklarieren kann, der zwar eingeschränkt, nicht jedoch erweitert werden darf.
- *Ersetzungsgruppen.* Die *Substitution Groups* sind ein spezieller Mechanismus in XML Schema, der es erlaubt, anstelle eines Elementes ein anderes verwenden zu dürfen (sofern es einen abgeleiteten Typ hat und in der Ersetzungsgruppe des zu ersetzenden Elements ist). Dies kann hilfreich sein für heterogene Listen, zwingt aber zu sauberer Programmierung bei der Verarbeitung, die typbasiert erfolgen muss, denn dem Content Model des Elementes, in dem ein Element einer Ersetzungsgruppe verwendet wurde, ist nicht anzusehen, dass auch andere Elemente vorkommen können. Will man den Aufwand für die Unterstützung von Ersetzungsgruppen nicht betreiben, sollte man diesen Mechanismus per `block` und/oder `final` von vornherein verbieten. Will man dagegen Ersetzungsgruppen benutzen, kann sich andersherum anbieten, bestimmte Elemente zwar in Content Models anzugeben, durch ein `abstract="true"` in der Deklaration dieser Elemente aber zu erzwingen, dass sie durch ein Element einer Ersetzungsgruppe ersetzt werden müssen.
- *Typersetzung.* Die soeben besprochenen Ersetzungsgruppen erlauben es, ein Element anstatt eines anderen zu verwenden. Dies muss explizit im Schema erlaubt sein. Durch *Type Substitution* (die das `xsi:type` Attribut im Dokument verwendet) kann ein ähnlicher Effekt erreicht werden, indem der Typ eines Elements explizit im Dokument angegeben wird. Da es sich hier (auf der für XML Schema wichtigen Typebene) um den identischen Menchanismus handelt wie bei der Ersetzungsgruppe, können beide Mechanismen über das gleiche Attribut im Schema (`block="substitution"`) gesteuert werden.

Wie man an diesen Ausführungen sieht, sind viele der Features von XML Schema auf der Typebene definiert. Wie am Anfang des Artikels erwähnt, ist es jedoch heute so (und wird wohl auch noch für längere Zeit zumindest zu berücksichtigen sein), dass ein grosser Teil an XML-verarbeitender Software nicht sauber typbasiert programmiert ist, was aufgrund der fehlenden standardisierten APIs auch verständlich ist, sondern einfach Element- bzw. Attributnamen interpretiert. Dieses Dilemma lässt sich eigentlich nur auf zwei Arten lösen:

- *Verzicht auf typbasierte Verarbeitung.* In diesem Fall verbietet man im Schema alle Dinge, die sich auf die Typebene beziehen (auf jeden Fall Typersetzung und damit auch Ersetzungsgruppen). Damit werden die Fehlermöglichkeiten für „naive“ Programmierer stark reduziert.
- *Explizite typbasierte Verarbeitung.* An den Stellen im Schema, wo Typersetzung zugelassen wird (und das ist per Default überall!), müssen Programmierer prinzipiell davon ausgehen, dass sie in diesem oder nachfolgendenden Versionen des Schemas verwendet wird, und die Software entsprechend schreiben. Dies bedeutet einen Mehraufwand für die Programmierer, bringt dafür aber höhere Flexibilität was die Weiterentwicklung des Schemas angeht.

Um die bisherigen Ausführungen zu illustrieren, soll ein kleines Beispiel die Verwendung (und Anforderungen an die verarbeitende Software) von Typersetzungen demonstrieren. Nehmen wir an, wir wollen

das oben vorgestellt Schema so erweitern, dass auch eitle Personen dargestellt werden können, die auf ihren Titel wert legen. Es sollen von der Idee her also Namen dieser Art möglich werden:

```
<person>
  <name>
    <givenname>Helmut</givenname>
    <surname>Kohl</surname>
    <title>Dr.</title>
  </name>
</person>
```

Zur Realisierung dieses Zieles lassen sich nun drei Wege denken, die alle ihre Vor- und Nachteile haben.

- *Wiederverwendung des Typs des name Elements.* In diesem Fall wird der Typ des `name` Elements verwendet, um ihn per Erweiterung als Grundlage eines `vainname` Elements zu machen. Damit die Einbettung in das Gesamtschema nach wie vor funktioniert, muss dann natürlich auch das `person` Element geändert werden, um das neuen `vainname` Element zuzulassen. Dieses Modellierungsmuster macht die Erweiterung also sehr schön im Schema und in der Instanz ersichtlich, verlangt aber nach Modifikationen, die sich im Schema nach oben (bis zum Document Element) fortsetzen. Auf diese Weise sind „naive“ Programmierer gezwungen, ihre Programme anzupassen, die anderfalls die neuen Elemente nicht kennen und mit ihnen nicht umgehen können.

  ```
  <vainperson>
    <vainname>
      <givenname>Helmut</givenname>
      <surname>Kohl</surname>
      <title>Dr.</title>
    </vainname>
  </vainperson>
  ```

- *Verwendung einer Ersetzungsgruppe.* In diesem Fall wird ebenso wie zuvor ein neues Element `vainname` definiert, es wird jedoch als Mitglied der Ersetzungsgruppe für das name Element deklariert, und damit kann die Modifikation des `person` Elementes vermieden werden, da das `vainname` Element als Mitglied der Ersetzungsgruppe des `name` Elements automatisch überall dort erlaubt ist, wo das `name` Element erlaubt ist. Hier bleibt Grossteil des Schemas unangetastet, und es gibt nur eine lokale Modifikation dort, wo das Content Model eines Elementes erweitert wurde. „Naive“ Software funktioniert vielleicht zum Teil noch (ausserhalb der Verarbeitung von Personen), wird sich aber u.U. daran stossen, dass es kein `name` Element mehr gibt.

  ```
  <person>
    <vainname>
      <givenname>Helmut</givenname>
      <surname>Kohl</surname>
      <title>Dr.</title>
    </vainname>
  </person>
  ```

- *Verwendung von Typersetzung.* Hier ändern sich die Elementnamen überhaupt nicht, sondern das neue Element für den Titel wird dadurch erlaubt, dass das `name` Element per Typersetzung als `vainnameType` deklariert wird, und deshalb das `title` Element enthalten darf. Nur bei sauberer typbasierter Verarbeitung wird eine solche Lösung funktionieren. Aus Typsicht ist sie identisch mit der vorangegangenen Variante. Aus der Softwaresicht kann ein „naives“ Programm hier sogar noch den Namen einer Person finden, wird aber u.U. Probleme haben, das im Namen nicht vorgesehene `title` Element zu verarbeiten.

  ```
  <person>
    <name xsi:type="vainnameType">
      <givenname>Helmut</givenname>
      <surname>Kohl</surname>
      <title>Dr.</title>
    </name>
  </person>
  ```

Obwohl diese drei Dokumente die gleichen Informationen enthalten, stellen sie also unterschiedliche Anforderungen an die Software, die sie verarbeiten soll. Insbesondere ist es aber auch wichtig, dass je nach dem Design des Ausgangsschemas und der darin erlaubten oder verbotenen Mechanismen manche Varianten u.U. gar

nicht realisiert werden könnten, z.B. wenn das `name` Element mit `block="substitution"` definiert ist. Vor allem, wenn man daran denkt, dass Schemas oft evolutionär und dezentral entstehen, d.h. bei der Weiterentwicklung die früher und von anderen Personen gesetzten Vorgaben nicht beeinflusst werden können, zeigt sich, dass eine bewusste und sorgfältige Auswahl der erlaubten Mechanismen notwendig ist.

Lebensdauer von Schemas, Dokumenten und Software

Auf den ersten Blick mögen diese Betrachtungen etwas übertrieben oder konstruiert wirken. In manchen Anwendungsfällen kann dies auch durchaus stimmen. Man sollte sich aber die sehr unterschiedlichen Anforderungen in Erinnerung rufen, die an die Lebensdauer von Schemas, Dokumenten und Software gestellt werden. Schemas sind tendenziell langlebig und vor allem oft einer evolutionären Entwicklung unterworfen. Gutes Schemadesign sollte daher sicherstellen, dass auch nach einer Weiterentwicklung die alten Dokumente gemäss dem neuen Schema gültig bleiben, und dass die für das alte Schema geschriebene Software auch mit Dokumenten nach gemäss des neuen Schemas umgehen kann. Dies zu ignorieren und anzunehmen, dass man nach einer Schemaweiterentwicklung „einfach“ alle alten Dokumente konvertiert, ist oftmals unrealistisch.

Die Frage der Weiterentwicklung von Schemas und Software ist aber stark abhängig von den Dokumenten selber. Während Schemas und Software fast immer recht lange Lebensdauern haben, sind Dokumente hier sehr unterschiedlich. Das kann von einer Lebensdauer von maximal wenigen Minuten (kurzlebige Web Services) bis zu Jahrzehnten (juristisch relevante Dokumente) reichen, und je nachdem, wo man in diesem Spektrum liegt, wird man andere Anforderungen an die Langlebigkeit und Robustheit von Schemas, Dokumenten und Software haben.

Wichtig ist, dass man sich dieser Frage am Beginn eines Schema-basierten Projektes stellt, und die entsprechenden Konsequenzen für das Schemadesign zieht. Im kommenden Artikel zu diesem Thema soll auf diese Frage detaillierter eingegangen werden, ebenfalls auf die damit zusammenhängende Frage der Verwendung von Namespaces und ihrem Management. Abschliessend soll im vorliegenden Artikel noch kurz eine Frage diskutiert werden, die schon bei DTDs zu nicht endenden Diskussionen führte, und die im Lichte von XML Schema neu bewertet werden sollte.

Elemente oder Attribute?

Eine immer wiederkehrende und nach wie vor nicht komplett beantwortbare Frage ist die nach der Entscheidung, Elemente oder Attribute zu wählen. Zwei wichtige Gründe, die bei DTDs noch für Attribute sprachen, sind bei Verwendung von XML Schema nicht mehr stichhaltig:

- *Attribute haben Typen.* In DTDs haben Attributen Typen, wenn auch nur sehr eingeschränkt (z.B. `NMTOKENS`). Dies kann u.U. ein Vorteil gegenüber Elementen sein, die als einzigen „Typ“ weitere Elemente oder `#PCDATA` haben.
- *Identity Constraints.* `ID/IDREF` ist ein spezieller Attribut-Typ, der Verweise mit referentieller Integrität zwischen Attributen erlaubt. In vielen nicht-trivialen XML-Anwendungen wird `ID/IDREF` verwendet.

Beide Fälle werden von XML Schema auch für Elemente abgedeckt: Das Konzept der *Simple Types* ist orthogonal zu Attributen und Elementen, so dass alle Simple Types auch für Elemente verwendet werden können, und XML Schema’s Identity Constraints lassen sich durch XPath gleichermassen auf Attribute und Elemente anwenden.

Aus dieser Perspektive besteht bei XML Schema kein Grund mehr, überhaupt noch Attribute zu verwenden, ausser dort, wo sie extern vorgegeben sind durch bestehende Standards (z.B. Namespaces oder XLink) oder andere Vereinbarungen. Nach wie vor für Attribute sprechen nur noch “weiche” Kriterien wie Übersichtlichkeit und kompaktere Codierung. Ebenso zu beachten ist die Normalisierung von Attributwerten in XML (durch den XML Standard selber definiert), die dazu führt, dass Whitespace und insbesondere Zeilenenden anders behandelt werden in Elementen und Attributwerten. Dieses Verhalten kann allerdings durch das `whiteSpace` Attribut bei simplen Typen gesteuert werden.

Fazit und Ausblick

Wie zu Beginn angekündigt hat dieser Artikel gezeigt, dass XML Schema zwar eine ganze Reihe neuer und interessanter Features beinhaltet, dies aber auch eine Komplexität mit sich bringt, die auf den ersten Blick

manchmal undurchschaubar erscheint. In diesem Artikel haben wir eine ganze Reihe dieser Features angeschaut hinsichtlich ihrer Verwendung in Schemas und ihrer Auswirkung auf Dokumente. Im folgenden Artikel werden wir diese Features mehr unter dem Gesichtspunkt betrachten, wie sie in der Praxis verwendet (oder vermieden) werden sollten, und wie man dies schon vom Beginn der Schema-Entwicklung an berücksichtigen sollte.