Lose gekoppelt hält länger

Entwurf erweiterbarer XML Schemas

Erik Wilde

XML bietet verschiedene Möglichkeiten, erweiterbare Datenstrukturen zu definieren und zu verwenden, aber es bleibt dennoch der Umsicht und vor allem der Planung von Designern überlassen, XML tatsächlich so zu verwenden, dass es diese Vorteile ausspielen kann. In diesem Artikel betrachten wir, wieso Erweiterbarkeit bei der Verwendung von XML ein wichtiger Aspekt ist, und wie sich diese Erweiterbarkeit mit XML Schema erreichen lässt. Als weiteren Aspekt betrachten wir die Offenheit eines Schemas, also die Frage, inwieweit ein Schema Erweiterungen in Dokumenten zulässt.

Im Artikel "Verwirrende Vielfalt — Modellierungsvarianten mit XML Schema" im XML & Web Services Magazin 6.03 haben wir XML Schema hauptsächlich unter dem Aspekt betrachtet, wie man mit Typen und Elementen/Attributen umgeht, und wie man diese Mechanismen einsetzen kann, um mit unterschiedlichen "Stilen" die an sich gleichen konzeptuellen Modelle in unterschiedliche logische XML Schema Implementierungen abzubilden. In diesem Artikel wollen wir diese etwas theoretischen Betrachtungen ganz praktisch unter dem Aspekt betrachten, dass ein XML Schema in der Praxis meistens nicht den Endpunkt einer Entwicklung darstellt, sondern einen Punkt in einem Kontinuum fortwährender Weiterentwicklung von Systemen, Formaten, und Protokollen, der das Erreichen des nächsten Teilschrittes möglichst einfach machen sollte.

Loose Coupling

Warum sind erweiterbare Schemas überhaupt wichtig? Web Services wie sie heute mit viel Geräuschentwicklung verkauft werden, bieten etwas provokativ ausgedrückt zunächst einmal nicht viel anderes als eine recht einfach Infrastruktur, die eine plattformunabhängige Kommunikation verteilter Komponenten ermöglicht. Die Art der Interaktionen und vor allem die Strukturen der ausgetauschten Daten sind dabei völlig frei. Das ist natürlich ein Vorteil, hat aber auch zur Folge, dass man bei der Definition der Interaktionen und Strukturen gut und weniger gut vorgehen kann. Web Services sind von ihrer Natur her für lose gekoppelte Systeme gedacht, also Systeme, die nicht notgedrungen gemeinsam definiert, implementiert und modifiziert werden. Aus diesem Grunde sollte sich das Design eines Web Service wenn immer möglich an einer losen Kopplung orientieren. Da wir uns in dieser Artikelserie mit XML Schema befassen, wollen wir die Definition der Web Service Interaktionen (z.B. RPC vs. REST) aussen vor lassen und uns ausschliesslich mit den verwendeten Datenstrukturen beschäftigen.

Lose Kopplung auf der Ebene der Datenstrukturen hat als Ziel, die getrennte Evolution von Web Service Teilnehmern zu ermöglichen. Betrachten wir eine klassische Client/Server-Architektur, so gibt es zwei unterschiedliche Szenarien:

- *Evolution des Servers:* Der Server wird weiterentwickelt und unterstützt neue Strukturen, sowohl in Anfragen als auch in Antworten. Anfragen alter Clients enthalten diese neue Strukturen nicht und müssen trotzdem noch akzeptiert werden, ebenso müssen die Antworten auf Anfragen auch von alten Clients verstanden werden können.
- *Evolution des Clients:* Der Client wird weiterentwickelt und unterstützt neue Strukturen, sowohl in Anfragen als auch in Antworten. Die Anfragen des Clients müssen jedoch auch von alten Servern noch verstanden werden, und die Antworten alter Server müssen vom Client verarbeitet werden können.

In beiden Fällen erwarten wir von einem gut designten System, dass es weiterhin zuverlässig funktioniert und Clients und Server zusammenarbeiten können. Das Web ist ein hervorragendes Beispiel für eine solche Struktur: Die Evolution eines Servers besteht dort z.B. darin, dass Dokumente in einer neueren HTML-Version zur Verfügung stehen als die vom Browser implementierte, während die Evolution eines Clients darin besteht, dass der Browser eine neuere HTML-Version unterstützt, von einem Server aber Dokumente in einer älteren Version erhält. Man hätte im Design des Web entsprechende Aushandlungsmechanismen auf Protokollebene vorsehen können (die es im übrigen auch gibt mit der *HTTP Content Negotiation*), nur hat man sich anstatt

dessen dafür entschieden, so viel wie möglich bereits auf der Ebene der Datenstrukturen zu lösen. Diese Entscheidung ist sehr sinnvoll und sollte auch bei der Entwicklung von Web Services als Richtschnur dienen. Natürlich wird es immer Weiterentwicklungen geben, die sich nicht auf evolutionäre Entwicklungen der Datenstrukturen abbilden lassen, aber das Ziel sollte sein, eine solche revolutionäre Entwicklung so selten wie möglich durchführen zu müssen. Erst in diesem Fall müssen Aushandlungsmechanismen auf Protkollebene vorgesehen werden, die dann auch dazu führen können, das zwei Systeme feststellen, dass sie nicht mehr miteinander funktionieren können.

Eine ganz zentrale Frage, die es zu beantworten gilt, ist die der Erweiterungsstrategie: Wird es immer nur einen Pfad geben von aufeinanderfolgenden Versionen, die aber alle aufeinander aufbauen, so dass der Entwicklungsweg eine Linie bleibt? Oder ist es möglich und soll unterstützt werden, dass es verschiedene Erweiterungen geben kann und wird, die sich dezentral entwickeln und beliebig aufeinander aufbauen können, so dass der Entwicklungsweg im allgemeinsten Fall ein azyklischer Graph ist, der sehr komplex werden kann? Letzterer Fall ist sicher der schwierigere, kann aber gerade für grosse Softwaresysteme schon auf Grund ihrer Komplexität und der Vielzahl der zu berücksichtigenden und permanent neu entstehenden Anforderungen der einzig realistische sein. Die volle Komplexität einer solchen Umgebung wollen wir im vorliegenden Artikel nicht weiter betrachten, da sie auch hinsichtlich des Schema- und Namespace-Managements hohe Anforderungen stellt, sondern wollen uns dieses Thema für der dritten Teil der Artikelserie aufheben.

Arten der Erweiterung

Spricht man von Erweiterungen, so ist es sinnvoll, diese etwas genauer anzuschauen und in einer Art zu klassifizieren, die den Umgang damit erleichtert. Ohne Anspruch auf Vollständigkeit soll die folgende Liste einen kurzen Einblick geben, welche Arten der Erweiterung unterschieden werden können:

- *Vollständige Unabhängigkeit:* Sind die Daten einer Erweiterung komplett unabhängig, so können sie im einfachsten Fall ignoriert werden von alten Implementierungen. Damit verlieren diese Implementierungen natürlich Informationen, aber zumindest können sie die ursprünglichen Strukturen nach wie vor interpretieren und bleiben daher voll funktionsfähig.
- *Funktionale Abhängigkeiten:* Bestehen funktionale Abhängigkeiten zu einer Erweiterung, so muss dies in einer allen Implementierungen zugänglichen Weise gekennzeichnet werden können, so dass alte Implementierungen die Erweiterung zwar nicht im engeren Sinne verstehen, zumindest jedoch davon in Kenntnis gesetzt werden können, das dort etwas neues ist, das für die Interpretation wichtig ist. Ob die Abhängigkeit von alten Implementierungen ignoriert werden darf, ist von der Erweiterung abhängig und muss ebenfalls in einer allen Implementierungen zugänglichen Weise gekennzeichnet werden können.
- *Semantisch abgegrenzte Beschreibungen:* Eine spezielle Art mit Erweiterungen umzugehen ist eine semantisch abgegrenzte Beschreibung (streng betrachtet ist übrigens auch die im vorangegangenen Abschnitt beschriebene funktionale Abhängigkeit eine solche semantisch abgegrenzte Beschreibung). So könnte es in einer Publishing-orientierten Umgebung z.B. sinnvoll sein, bei Erweiterungen immer auch eine Formatierungsbeschreibung (z.B. via XSLT/FO) zu definieren, so dass die Erweiterungen zwar von alten Implementierungen nicht wirklich verstanden, zumindest aber doch formatiert und damit ausgegeben werden können.

Der Bereich der Erweiterbarkeit von Diensten ist eng verbunden mit der Frage, wie man Komponenten definiert, die sich über die Semantik von Daten verständigen können. Die oben beschriebenen Abhängigkeits- und Formatiersemantiken sind nur einfache und recht triviale Beispiele aus diesem Bereich. Das Modethema des *Semantic Web* wirbelt in diesem Bereich momentan viel Staub auf (aber bietet neben vielen neuen Syntaxen für altehrwürdige KI-Ansätze wenig wirklich neues), aber gewisse Minimalsemantiken (zumindest die "hängt ab von" oder "muss interpretiert werden können" Semantik) sollte man in einem auf Erweiterbarkeit angelegten System berücksichtigen, vielleicht auch noch etwas weitergehende Beschreibungen, sofern sie sich absolut klar identifizieren und definieren lassen. Dies ist einer der wichtigsten Punkte im Bereich semantischer Beschreibungen: Nur wenn sich die Semantik eindeutig und zweifelsfrei beschreiben lässt, sollte man überhaupt in Erwägung ziehen, sie zu modellieren, andernfalls verkompliziert man das System und schafft des weiteren viele Fehlerquellen durch Interpretationsspielräume.

Erweiterbares XML

Bevor wir uns mit dem Entwurf erweiterbarer Schemas befassen, wollen wir zunächst einmal die von XML selber zur Verfügung gestellten Wege ansehen, mit Hilfe derer man Erweiterbarkeit umsetzen kann. Dabei beschränken wir den Blick auf praktikablere und verbreitete Möglichkeiten. Als Beispiel wollen wir dabei die Personenbeschreibung aus dem letzten Artikel nehmen, die wir wiederum um einen Titel erweitern wollen.

```
<person>
  <name>
    <givenname>Helmut</givenname>
    <surname>Kohl</surname>
  </name>
</person>
```

- Attribute: Attribute können Elementen hinzugefügt werden und werden (z.B. aus Sicht des DOM oder XPath Modells) eher als zusätzliche Knoten denn als veränderte Struktur gesehen. Deshalb stellen Attribute eine Art Lightweight Methode dar, um XML zu erweitern. Der grosse Nachteil bei Attributen ist, dass sie nicht weiter strukturiert werden können, so dass die Erweiterungen auf einfache Datentypen beschränkt bleiben.

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

- Elemente: Treten neue Elemente in einem XML Dokument auf, so verändern sie die Kind-Beziehungen im Dokument, so dass der strukturelle Eingriff stärker ist als beim Hinzufügen von Attributen. Dafür lassen sich Elemente beliebig weiter strukturieren, so dass auf diese Weise beliebig komplexe neue Strukturen hinzugefügt werden können.

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

- Namespaces: Dieser Mechanismus in unabhängig von Attributen oder Elementen und kann für beides benutzt werden. Namespaces erlauben es, Attribute und Elemente bestimmten Namensräumen zuzuordnen und auf diese Weise zu erreichen, dass in einem Dokument einfach erkannt werden kann, aus welchem Vokabular Attribute und Elemente stammen. Im Beispiel werden beide Varianten (Attribut und Element) gezeigt.

  ```
  <person xmlns:eitel="http://example.com/eitle_person">
    <name eitel:titel="Graf">
      <givenname>Otto</givenname>
      <surname>Lambsdorff</surname>
    </name>
  </person>

  <person xmlns:eitel="http://example.com/eitle_person">
    <name>
      <givenname>Helmut</givenname>
      <surname>Kohl</surname>
      <eitel:titel>Dr.</eitel:titel>
    </name>
  </person>
  ```

Während die Entscheidung Attribut vs. Element schon oftmals deshalb klar ist, weil man die Einschränkungen der Attributwerte auf einfache Datentypen nicht hinnehmen will, bliebt die Frage hinsichtlich der Namespaces zu klären. Es ist auf jeden Fall sinnvoll, Namespaces zu verwenden, weil sie als Kategorisierung von Namen in Dokumenten sehr nützliche Hilfe leisten. Die Markierung von Erweiterungen mittels Namespaces werden wir weiter unten näher betrachten, in aller Ausführlichkeit aber erst im letzten Teil der Artikelserie unter die Lupe nehmen.

Erweiterbarkeit und Offenheit

Um Schemas sinnvoll erweitern zu können, müssen wir zwei an sich getrennte Aspekte betrachten: Zum einen muss das Design des Ausgangsschemas schon so gestaltet sein, dass es sich überhaupt erweitern lässt. Die dafür wichtigen Aspekte wurden im letzten Artikel betrachtet, wo es unter anderem darum ging, ob Deklarationen von Typen, Elementen und Attributen lokal oder global erfolgen sollten. Nur wenn sie global definiert sind, tragen sie einen Namen, und lassen sich zum Zwecke der Redefinition referenzieren. Zwei weitere im letzten Artikel unterschlagene Aspekte aus diesem Bereich sind *Named Groups* (XML Schema kennt *Attribute Groups* und *Model Groups*), die es ermöglichen, weitere Komponenten einer Typdefinition global zu definieren und damit redefinierbar zu machen. Ein konsequent auf globalen Komponenten basierendes Schema ist zwar recht voluminös, bietet aber am meisten Ansatzpunkte, um Komponenten wiederzuverwenden oder zu redefinieren.

Der zweite Aspekt, der allerdings über die direkte Frage der Erweiterbarkeit des Ausgangsschemas hinausgeht, ist der, inwieweit Dokumente Erweiterungen zulassen, also z.B. Attribute oder Elemente an Stellen gestattet sind, an denen dann in erweiterten Schemas zusätzliche Informationen erscheinen können. Dieser Aspekt eines Schemas soll kurz unter dem Stichwort der Offenheit betrachtet werden und ist im Prinzip vollkommen unabhängig von der Erweiterbarkeit, auch wenn in der Praxis diese beiden Dinge natürlich unbedingt gemeinsam betrachtet werden müssen.

Offene XML Schemas

Die wichtigsten Mechanismen, um ein Schema offen zu gestalten, sind *Wildcards*. XML Schema kennt *Element Wildcards* (`xs:any`) und *Attribute Wildcards* (`xs:anyAttribute`), und beides sind Mechanismen, um in einer Model Group oder in einer Attributliste zu vermerken, dass dort weitere und unspezifizierte Elemente oder Attribute erlaubt sind. Wildcards können auf zwei Arten gesteuert werden: Zum einen kann über `processContents` angegeben werden, inwieweit die Validierung der auf eine Wildcard passenden Elemente oder Attribute verlangt wird. Des weiteren kann über `namespace` angegeben werden, aus welchen Namespaces die für eine Wildcard verwendeten Elemente oder Attribute stammen müssen.

Die andere, subtilere und für viele (Anwender wie auch Programme) oft überraschende Art der Offenheit ist *Type Substitution* und in XML Schema per Default erlaubt. Type Substitution erlaubt es, einem Element (für Attribute ist sie schon aus rein syntaktischen Gründen nicht anwendbar) in einem Dokument einen anderen Typ zuzuordnen, und wir haben sie bereits im letzten Artikel näher kennengelernt. In syntaktisch etwas anderer Form tritt die Type Substitution auch bei den *Substitution Groups* auf. Type Substitution sollte über das `blockDefault` Attribut des Schemas zunächst einmal generell verboten und anschliessend nur an gewünschten und dann auch bei der Implementierung zu beachtenden Stellen wieder zugelassen werden. Dies geschieht über `block` bei Complex Type und bei Element-Deklarationen.

Das Design offener Schemas ist zum einen sehr wichtig, um Erweiterungen in geordnete Bahnen lenken zu können. Andererseits ist es auch grundlegend, um schon in die erste Version der Implementierungen an den Stellen Offenheit zuzulassen, an denen später mit Erweiterungen gerechnet werden muss. So war z.B. schon von der ersten Version an in HTML klar definiert, dass Browser unbekannte Elemente und Attribute ignorieren müssen, was zwar eine eher grobschlächtige Form der Offenheit darstellt, aber überhaupt erst die Möglichkeit schaffte, dass sich das Web mit der Dynamik entwickeln konnte, die es erfolgreich gemacht hat.

Das Ziel des Designs offener Schemas muss es sein, dass auch eine Implementierung der ersten Stunde mit erst später entwickelten und verwendeten Erweiterungen umgehen kann. Das wird teilweise nichts anderes sein, als unbekannte Teile eines Dokuments kontrolliert zu ignorieren, kann aber gerade bei der Verwendung von Type Substitution auch soweit gehen, dass die Implementierung sauber auf der Typenebene arbeiten muss, weil ansonsten mit Mitgliedern von Substitution Groups nicht angemessen umgegangen werden kann.

Erweiterbare XML Schemas

Nach diesen eher theoretischen Ausführungen wollen wir nun untersuchen, wie sich die Erweiterbarkeit in einem XML Schema unterstützen lässt. Dafür gehen wir davon aus, dass im Falle einer Erweiterung eines Web Service nicht ein komplett neues Schema definiert wird, sondern das alte Schema erweitert wird, z.B. indem es mittels `xs:redefine` als Grundlage der Erweiterung verwendet wird. Wie genau man die Erweiterungen

miteinander verbindet, und in welchen Fällen `xs:redefine`, `xs:include` und `xs:import` zum Einsatz kommen, wird uns auch erst im drittel Teil der Artikelserie beschäftigen. Hier wollen wir in einem einfachen Ansatz annehmen, dass das Schema für die erste Version in einem XML Schema definiert wird, und die Erweiterungen dann eigenständige Schemas sind, die die erste Version importieren. Damit bleiben, und das ist für uns zunächst das wichtigste, die Namespaces erhalten. Unser Beispiel soll zur Demonstration an allen nur möglichen Stellen globale Definitionen verwenden:

```
<xs:schema targetNamespace="http://example.com/person" xmlns:ns="http://example.com/person"
xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
blockDefault="#all">
        <xs:element name="person" type="ns:personType"/>
        <xs:complexType name="personType">
                <xs:group ref="ns:personGroup"/>
        </xs:complexType>
        <xs:group name="personGroup">
                <xs:sequence>
                        <xs:element ref="ns:name"/>
                </xs:sequence>
        </xs:group>
        <xs:complexType name="nameType">
                <xs:group ref="ns:nameGroup"/>
        </xs:complexType>
        <xs:group name="nameGroup">
                <xs:sequence>
                        <xs:element ref="ns:givenname"/>
                        <xs:element ref="ns:surname"/>
                </xs:sequence>
        </xs:group>
        <xs:element name="name" type="ns:nameType"/>
        <xs:element name="givenname" type="xs:string"/>
        <xs:element name="surname" type="xs:string"/>
</xs:schema>
```

Nun wollen wir Erweiterungen zulassen, das Schema also offen gestalten. Dazu gibt es wie oben beschrieben verschiedene Varianten. Wildcards können an verschiedenen Stellen verwendet werden. Um den Namen für weitere Attribute zu öffnen, wären die folgenden geänderten bzw neuen Deklarationen sinnvoll:

```
<xs:complexType name="nameType">
        <xs:group ref="ns:nameGroup"/>
        <xs:attributeGroup ref="ns:anyAttributeGroup"/>
</xs:complexType>
<xs:attributeGroup name="anyAttributeGroup">
        <xs:anyAttribute processContents="lax" namespace="##other"/>
</xs:attributeGroup>
```

Das `anyAttribute` Element wurde als eigenständige Attribute Group definiert, um gegebenenfalls an anderer Stelle wiederverwendet werden zu können. In sehr ähnlicher Weise kann das Schema für weitere Elemente geöffnet werden:

```
<xs:complexType name="nameType">
        <xs:sequence>
                <xs:group ref="ns:nameGroup"/>
                <xs:group ref="ns:anyGroup"/>
        </xs:sequence>
</xs:complexType>
<xs:group name="anyGroup">
        <xs:any processContents="lax" namespace="##other" minOccurs="0" maxOccurs="unbounded"/>
</xs:group>
```

Diese Art der Erweiterbarkeit funktioniert zwar, aber nur, weil wir die erlaubten Namespaces der Element Wildcard mit `##other` auf andere Namespaces als den `targetNamespace` eingeschränkt haben. Hätten wir das nicht getan und Erweiterungen im `targetNamespace` des Schemas zugelassen, und hätte dann eine Erweiterung im selben Namespace stattgefunden, die ein optionales Element anfügt, so wäre die XML Schema Regel verletzt worden, die nicht-deterministische Inhaltsmodelle verbietet.

Will man anstatt der Wildcards eher auf eine Typ-basierte setzen, so müssen die Elemente und Typen an den Stellen, an denen abgeleitete Typen zugelassen werden sollen, entsprechend deklariert werden, denn der Default wurde mit `blockDefault` in `xs:schema` auf `#all` gesetzt. Das `block` Attribut sollten dann bei Typ und Element gleich gesetzt werden, wobei beim Element u.U. zusätzlich auch noch `substitution` gesetzt werden sollte, wenn keine Substitution Groups erlaubt werden sollen.

```
<xs:complexType name="nameType" block="restriction">
        <xs:group ref="ns:nameGroup"/>
</xs:complexType>
<xs:element name="name" type="ns:nameType" block="substitution restriction"/>
```

Auf diese Weise sind jetzt Type Substitutions erlaubt (aber nur mit erweiterten Typen, und nicht durch Substitution Groups). Dass die Elemente und ihre Typen wie in diesem Fall gemeinsam umdeklariert werden müssen, ist alleine der Disziplin des Schemadesigners überlassen, XML Schema bietet hier keine Hilfe.

Wie erweitert man am besten?

Die Beispiele haben gezeigt, dass man verschiedene XML Schema Mechanismen einsetzen kann, um das Schema zu öffnen und damit erweiterbar zu machen. Welchen Mechanismus man wählt, ist natürlich bis zu einem gewissen Grad Geschmackssache. Allerdings ist es so, dass der Type Substitution Mechanismus zwangsläufig davon ausgeht, dass man einen zu erweiternden Typ als Basis kennt und zur Verfügung hat, was insbesondere dann wenig realistisch ist, wenn man von einer eher verteilten und nicht in einer geraden Linie verlaufenden Evolution des Schemas ausgeht. Das offenste und flexibelste Modell ist das der Erweiterung durch Elemente, auf das wir uns deshalb im folgenden beschränken wollen. Die Erweiterung durch Type Substitution reservieren wir für die Fälle, in denen es eine zentrale Koordination der Entwicklungen gibt.

Im Beispiel ermöglicht die Element Wildcard das Vorkommen beliebiger Elemente am vorgesehen Punkt für Erweiterungen. Dies ist sehr wichtig, denn ein Ziel ist ja, dass alle Dokumente (also insbesondere die mit Erweiterungen) auch vom ursprünglichen Schema validiert werden. Nur so kann die beliebige Erweiterbarkeit eines Systems sichergestellt werden. Aber natürlich sollen die Dokumente nicht nur validiert werden, sonder auch verarbeitet werden können. Damit sind wir beim eingangs erwähnten Punkt, dass Semantik notwendig sein kann. Default könnte sein, alles unbekannte zu ignorieren, doch das ist sehr simpel und kann zum Teil auch unerwünscht sein, wenn z.B. eine gewisse Erweiterung essentiell ist und im Dokument ausgedrückt werden soll, dass sie nicht ignoriert werden darf. Diese Information kann auf verschiedene Arten ausgedrückt werden, von denen hier zwei vorgestellt werden sollen. Im ersten Fall wird für das Document Element ein spezielles Attribut definiert, dass als Inhalt alle die Namespaces enthält, die nicht ignoriert werden dürfen:

```
<xs:element name="person" type="ns:personType"/>
<xs:complexType name="personType">
        <xs:group ref="ns:personGroup"/>
        <xs:attribute name="mustUnderstand">
                <xs:simpleType>
                        <xs:list itemType="xs:anyURI"/>
                </xs:simpleType>
        </xs:attribute>
</xs:complexType>
```

Auf diese Weise kann ein Programm sehr einfach feststellen, ob in einem Dokument Erweiterungen enthalten sind, die es verstehen muss. Falls das Dokument unbekannte aber im `mustUnderstand` Attribut aufgeführte Erweiterungen enthält, darf keine Verarbeitung erfolgen, sondern es muss auf eine Art vorgegangen werden, die im Verarbeitungsmodell des Systems vorgesehen sein muss, z.B. ein besonderer Fehlerfall oder eine Neuaushandlung von ausgetauschten Formaten.

Etwas eleganter, aber aus Sicht des Schemas aufwendiger ist es, die Information in die Erweiterungen selber zu stecken und beispielsweise zu verlangen, dass jede Erweiterung von einem im ursprünglichen Schema vorgegebenen speziellen Typ abgeleitet werden muss, der z.B. leer ist und nur Attribute enthält, die für die Verarbeitung von Erweiterungen notwendig sind:

```
<xs:complexType name="extensionType" abstract="true">
        <xs:attribute ref="ns:mustUnderstand" use="required"/>
        <xs:attribute ref="ns:dependsOn"/>
</xs:complexType>
<xs:attribute name="mustUnderstand" type="xs:boolean"/>
<xs:attribute name="dependsOn">
        <xs:simpleType>
                <xs:list itemType="xs:anyURI"/>
        </xs:simpleType>
</xs:attribute>
```

Dieser Typ (der `abstract` ist und deshalb nie direkt in einem Dokument auftauchen kann) muss nun als Grundlage für alle Erweiterungen genommen werden, so dass die beiden Attribute bei allen Erweiterungen definiert sind. Das Attribut `dependsOn` ermöglicht für jede Erweiterung, eventuelle Abhängigkeiten zu anderen Erweiterungen zu definieren, so dass etwas mehr Informationen über die Notwendigkeit und Abhängigkeiten von Erweiterungen ausgedrückt werden kann als mit dem blossen `mustUnderstand`. Die Attribute sind als globale Attribute definiert, damit sie in Dokumenten mit ihrem Namespace verwendet werden müssen. Dies entspricht eher der Erwartung von Benutzern, als wenn die Attribute dort unqualifiziert erscheinen müssten.

Damit diese Lösung funktioniert, müssen sich Erweiterungen natürlich an die Konvention halten, vom `extensionType` abzuleiten. Diese Bedingung kann im Schema selber nicht formuliert werden, da die Wildcards nur auf Namespaces eingeschränkt werden können, nicht jedoch auf Typen. Mit diesem Nachteil im Kopf wäre auch ein auf Type Substitution basierender Ansatz denkbar, der folgendermassen aussähe (der soeben definierte `extensionType` wird in diesem Beispiel ebenfalls verwendet):

```
<xs:complexType name="nameType" block="restriction">
	<xs:sequence>
		<xs:group ref="ns:nameGroup"/>
		<xs:element ref="ns:extensionElement" minOccurs="0" maxOccurs="unbounded"/>
	</xs:sequence>
</xs:complexType>
<xs:element name="extensionElement" type="ns:extensionType" abstract="true"
block="restriction"/>
```

Dies ist ein Prinzip zwar eleganter Ansatz, der aber den grossen Nachteil hat, dass er nicht zu einem Schema führt, mit dem man beliebige Erweiterungen erfolgreich validieren kann. Es handelt sich hier also um ein gutes Beispiel eines erweiterbaren Schemas, das aber nicht die oft ebenso gewünschte Offenheit besitzt. Aus diesem Grund empfiehlt es sich, eher Wildcards zu verwenden als Type Substitution. Es muss bei den Wildcards dann über ein nicht im Schema definierbares Verarbeitungsmodell sichergestellt werden, dass zum Beispiel eine Erweiterung wie oben beschrieben immer von einem vorgegebenen Typ abgeleitet wird.

Überhaupt ist es sehr ratsam, in einem Umfeld mit erweiterbaren und offenen Schemas ein gut definiertes Verarbeitungsmodell zu haben, dass nicht nur die Schemas und deren Erweiterung beschreibt, sondern auch den Umgang mit Dokumenten. Die Attribute `mustUnderstand` und `dependsOn`, die in den vorangegangenen Beispielen auftauchten, deuten auch schon in diese Richtung, denn sie tragen Informationen, die sich auf die Verarbeitung auswirken sollen und sie müssen deshalb in jedem Fall beachtet werden. Weil dem Verarbeitungsmodell einige Beachtung geschenkt werden sollte, wird dieser Aspekt neben der Definition und Verwaltung erweiterbarer Schemas und Namespaces den zweiten Schwerpunkt des dritten Artikels dieser Artikelserie bilden. Dieser wird den Abschluss der Serie bilden, die sich im ersten Teil mit Schema Modellierungsfragen im Allgemeinen und im vorliegenden Teil mit Fragen der Erweiterbarkeit und Offenheit von Schemas beschäftigt hat.