Drum prüfe, wer sich ewig bindet...

Namespaces und Versionierung von XML Schemas

Erik Wilde

Für die Versionierung von XML Schemas ist es notwendig, sich Gedanken über den Umgang mit Versionen zu machen, und zwar aus zweierlei Sicht. Die erste Sicht ist die des Schema-Entwickler, dem sich die Frage stellt, wie er die Namespaces handhabt, die für die verschiedenen Schemas verwendet werden. Die andere Sicht ist die der Software-Entwickler, die in ihre Software Wissen darum einbauen müssen, wie mit Instanzen verschiedener Versionen umgegangen wird. Beide Sichten sollten gemeinsam dazu beitragen, ein möglichst robustes und flexibles Szenario zu implementieren, in dem verschiedene Schemaversionen koexistieren können.

In den vorangegangenen Artikeln dieser Serie wurde XML Schema hauptsächlich unter dem Aspekt betrachtet, wie man mit Typen und Elementen/Attributen modellieren kann ("Verwirrende Vielfalt — Modellierungsvarianten mit XML Schema", XML & Web Services Magazin 6.2003), und wie man Schemas erweiterbar und offen gestalten kann ("Lose gekoppelt hält länger — Entwurf erweiterbarer XML Schemas", XML & Web Services Magazin 3.2004). Im vorliegenden dritten Teil der Artikelserie wird nun der Frage nachgegangen, in welcher Art man diese beiden grundlegenden Aspekte anwenden sollte, um Schemas zu definieren, die nicht nur gut modelliert und erweiterbar sind, sondern auch robust und flexibel funktionieren in einer sich ändernden IT-Umgebung.

Generell lässt sich beobachten, dass trotz gegenteiliger Behauptungen die Frage lose gekoppelter Systeme mit Web Services nicht per Definition gelöst wird. Auf eine etwas polemische Art könnte man sagen, dass mit Web Services u.U. das Problem von der reinen Schnittstellendefinition hin zur XML Message verlagert wird, die über eine Schnittstelle geht. Oder, anders ausgedrückt, mit Web Services ist es trivial, einen Dienst zu definieren, der beliebige XML Messages akzeptiert, so dass die Schnittstelle an sich unproblematisch ist. Dies verlagert jedoch einfach das Problem der Versionierung von Schnittstellen zur Versionierung der Message-Formate: die Schnittstelle ist ohne Probleme in der Lage, beliebig unterschiedliches XML zu akzeptieren, zu transportieren, und auszuliefern, aber wird es beim Empfänger auch verstanden und korrekt verarbeitet?

Diese Frage nach XML-basierter Interoperabilität hat viele unterschiedliche Facetten, von denen viele z.B. in Steve Vinoski's lesenswerten "Toward Integration" Kolumnen in der IEEE Internet Computing (die Kolumnen sind online verfügbar unter `http://www.iona.com/hyplan/vinoski/#ic`) aufgegriffen werden. Der vorliegende Artikel beschäftigt sich mit der Frage, wie sich eine Schnittstelle (d.h. das Message-Format und die Verarbeitungsvorschriften für diese Messages) so definieren lässt, dass tatsächlich ein lose gekoppeltes System implementiert werden kann, in dem sich Komponenten individuell weiterentwickeln können, ohne die Stabilität des Gesamtsystems zu gefährden. Ziel dieses Vorgehens ist, die vielmals fälschlicherweise als Grundeigenschaften vorausgesetzten positiven Eigenschaften einer XML & Web Services basierten Architektur auch tatsächlich zu realisieren.

Versionierung vs. Offenheit/Erweiterbarkeit

Die im vorangegangenen Artikel diskutierten Methoden zur Definition erweiterbarer und offener Schemas sind nur ein Aspekt der Versionierung von Schemas (und damit Schnittstellen). Zur Erinnerung: Erweiterbarkeit stellt durch das Schemadesign sicher, dass neue Schemas definiert werden können, die Konzepte des Ausgangsschemas wiederverwenden, während die Offenheit eines Schemas sicherstellt, dass eine Instanz eines erweiterten Schemas auch gegen das Ausgangsschema erfolgreich validiert werden kann. Erweiterbarkeit und Offenheit greifen also eng ineinander. Die Versionierung baut darauf auf, indem sie festlegt, wie erweitert werden darf, und in welcher Weise mit Erweiterungen umgegangen werden kann und muss. Die Versionierung ist also aus einer Sicht ein Spezialfall der Erweiterung, aus der anderen Sicht aber auch etwas darüberhinausgehendes, weil sie im Idealfall sicherstellt, dass Schemas und Instanzen in verschiedenen Versionen interoperabel koexistieren können.

Damit Versionierung optimal unterstützt werden kann, muss ein Schema also offen sein (damit Software auch Instanzen anderer Versionen erfolgreich validieren und weiterverarbeiten kann), und es muss erweiterbar sein, damit neue Versionen inkrementelle Änderungen des Ausgangsschemas sind. Ein oft vernachlässigter Kernaspekt ist jedoch, dass zum Schema auch noch Verabrbeitungsvorschriften existieren müssen, die für alle Komponenten, die mit den Schemas und deren Instanzen umgehen, verbindliche Vorschriften festlegen. Nur wenn diese Verarbeitungsvorschriften so gestaltet sind, dass sie die Fallstricke und Probleme späterer Entwicklungen vorhersehen und vermeiden, lässt sich eine stabile und flexible Versionierung realisieren.

Vor den Verarbeitungsvorschriften soll jedoch ein etwas genauerer Blick auf die Versionierung von Schemas geworfen werden, vor allem auf die Frage, wie unterschiedliche Versionen markiert und erkannt werden. In diesem Zusammenhang spielen XML Namespaces eine sehr wichtige Rolle.

XML Namespaces und Schemas

Namespaces und Schemas sind im Prinzip unabhängig voneinander, doch für Schemas im produktiven Einsatz hat sich sinnvollerweise eingebürgert, sie immer mit einem Namespace Name zu versehen. XML Schema unterstützt dies explizit durch das `targetNamespace` Attribut, das es direkt im Schema erlaubt, den Namen des Namespaces für alle globalen Definitionen (Elemente, Attribute, Typen, Attributgruppen, Modellgruppen und Notations) des Schemas anzugeben.

Sobald das Schema versioniert wird, stellt sich nun die Frage, ob im neuen Schema auch ein neuer Namespace Name verwendet werden soll. Aus der Sicht der XML Schema und XML Namespaces Standards hat man hier freie Hand, es gibt keinerlei vorgeschriebene Vorgehensweisen in diesem Bereich. Die Frage der Namespace Names für XML Schema Versionen wird vor allem interessant im Zusammenhang mit der Frage, wie Komponenten reagieren können und sollen, die eine neue Schemaversion nicht unterstützen, aber XML Instanzen dieses Schemas verarbeiten sollen (Vorwärtskompatibilität). Es gibt hier drei verschiedene Vorgehensweisen:

1. *Der Namespace Name ändert sich immer:* In diesem Fall bekommt jede neue Version eines Schemas einen neuen Namespace Name. Der Namespace Name trägt somit keine nützliche Information über Schemaversionen hinweg und ist aus diesem Grund zwar nicht sinnlos (er taugt ja noch zur Identifikation von Namen aus diesem Namespace), aber weit weniger nützlich als er es sein könnte.
2. *Der Namespace Name ändert sich nur für Major Versionen:* Solange nur Minor Versionen erzeugt werden, bleibt der Namespace Name der gleiche. Eine Komponente kann also bedenkenlos eine XML Instanz verarbeiten, auch wenn sie das dazugehörige neuere Schema nicht kennt, solange der Namespace Name bekannt ist. Dies setzt voraus, dass Minor Versionen immer rückwärtskompatibel sind, so dass das neue Schema nicht bekannt sein muss. Das normale Vorgehen in diesem Fall ist, dass aus dem Ausgangsschema stammende Element/Attribute normal verarbeitet werden, während unbekannte neue Elemente/Attribute ignoriert werden. Die Offenheit des Ausgangschemas stellt dabei sicher, dass unbekannte Elemente/Attribute auftreten dürfen, die Validierung der Instanz aber trotzdem zulassen.
3. *Der Namespace Name ändert sich nie:* In diesem Fall bleibt der Namespace Name über Minor und Major Versionen hinweg konstant. Das heisst aber auch, dass Komponenten im Falle einer unbekannten Major Version in der Lage sein müssen, auf Informationen des neuen Schema zuzugreifen, damit sie diese Informationen (im Sinne des im vorangegangenen Teil diskutierten `mustUnderstand` Attributs) zur Interpretation der Instanz verwenden können. In diesem Fall sind Komponenten also robuster, müssen aber auch komplexer implementiert werden.

Keine dieser drei Varianten ist frei von Problemen, und es hat sich bisher keine komplett durchgesetzt. Am häufigsten wird im Moment wohl die mittlere Variante gewählt. Es lässt sich aber argumentieren, dass bei einem sauberen und robusten Design des Schemas und vor allem der verarbeitenden Komponenten die dritte Variante die bessere ist, weil sie eine losere Kopplung von Software-Komponenten an das Schema zulässt.

Bei der zweiten und dritten Variante ist zu bedenken, dass dem Namespace einer Instanz nicht mehr anzusehen ist, ob sie u.U. einem neueren Schema gehorcht, als eine Software-Komponente unterstützt. Im Falle der zweiten Variante ist das noch nicht so ein Problem, weil es hier immer rückwärtskompatible Änderungen sind; im Falle der dritten Variante wird dies jedoch sehr wichtig, weil eine Instanz mit einem bekannten Namespace Name durchaus Daten einer nicht rückwärtskompatiblen neueren Schemaversion enthalten kann.

Dies ist ein Risiko hinsichtlich der Verarbeitung, weil naiv implementierte Software auf diese Art u.U. Probleme erzeugt, wenn eine neue Schemaversion vorwärtskompatible Verarbeitung nötig machen würde (weil sich z.B. die Bedeutung eines Elementes verändert hat), die Software dies jedoch ignoriert (und das Element gemäss der alten Bedeutung verarbeitet).

Namespace Names

Bevor im Detail angeschaut wird, wie die drei Ansätze realisiert werden können, eine kurze Bemerkung zu Namespace Names: Namespace Names sind einfach URIs, was Benutzern grosse Freiheiten lässt hinsichtlich der Wahl der Namen. Die beiden wichtigsten Aspekte der Namespaces Names sind, dass es keine organisierte Registration oder Verwaltung dieser Namen gibt (sich jeder Benutzer also einfach einen Namen ausdenken kann), und dass sich hinter dem Namen nichts verbergen muss (die URI also nicht funktionieren muss, wenn man versucht, sie zu dereferenzieren).

Vor der Verwendung von Namespaces ist man gut beraten, eine Namespace Name Policy zu definieren, damit Namespace Names gewissen Regeln gehorchen und sich einfacher verwalten und verwenden lassen. Dass dies nicht immer funktioniert, kann man selbst beim W3C sehen, dessen Namespace Names offensichtlich verschiedenen Regeln gehorchen und nicht immer konsistent aufgebaut sind (`http://www.w3.org/XML/1998/namespace` und `http://www.w3.org/2001/XMLSchema` zeigen das). Je nach der verfolgten Strategie (die oben aufgezählten Varianten 1-3) sollte die für den Namespace Name relevante Information Bestandteil des Namens werden, also für Variante 1 `http://example.com/vocabulary/1/4`, für Variante 2 `http://example.com/vocabulary/1`, und für Variante 3 `http://example.com/vocabulary` (wobei die Nummern Major/Minor bzw. Major Versionsnummer sind). Diese Art, die mit dem Namespace verbundene Versionierungsinformation in den Namespace Name zu integrieren, ist reine Selbstdisziplin, macht Umgang und Verwaltung mit dem Namespaces aber wesentlich einfacher und weniger fehleranfällig.

Vom Namespace zum Schema

Wie kommt man nun vom Namespace Name zum Schema? Allgemein lässt sich dazu nichts sagen, im Prinzip muss man das Schema und den darin definierten Namespace Name kennen, so dass man ihn wiedererkennen kann, wenn man in einer Instanz auf ihn trifft. In der Praxis werden jedoch oft Dokumente an der Adresse des Namespace Names hinterlegt. Das sind manchmal sehr minimalistische Dokumente, wie beim Namespace für XSLT (`http://www.w3.org/1999/XSL/Transform`), oder manchmal auch eine informativere Web-Seite wie beim XHTML Namespace (`http://www.w3.org/1999/xhtml`).

Es empfiehlt sich jedoch, auch bei diesem Punkt Regeln aufzustellen, die allen Beteiligten das Leben erleichtern, und eine sinnvolle Regel in diesem Zusammenhang wäre die Verpflichtung, an der URI des Namespaces Names immer eine Web-Seite zu hinterlegen, die der *Resource Directory Description Language (RDDL)* gehorcht. RDDL ist eine recht simple Sprache, die es ermöglicht, in eine menschenlesbare XHTML-Seite maschinenlesbare Information einzubetten, die auf weitere Informationen zu einem Namespace verweist. Dazu bedient sich RDDL der XLink-Sprache, um diese Informationen einzubetten. Ein gutes Beispiel ist die RDDL Seite selber, die RDDL beschreibt und zudem die RDDL-Beschreibung des RDDL-Namespaces (`http://www.rddl.org/`) ist. Ein Auszug aus dieser Seite:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:rddl="http://www.rddl.org/"
      xmlns:xlink="http://www.w3.org/1999/xlink">
      ...
  <rddl:resource xlink:type="simple"
                 xlink:title="DTD for validation"
                 xlink:arcrole="http://www.rddl.org/purposes#validation"
                 xlink:role="http://www.isi.edu/in-notes/iana/assignments/media-
types/application/xml-dtd"
                 xlink:href="rddl-xhtml.dtd">
    <h3>7.4 Document Type Definition</h3>
      <p>A DTD <a href="http://www.rddl.org/rddl-xhtml.dtd">rddl-xhtml.dtd</a> for RDDL,
defined as an extension of XHTML Basic 1.0
using Modularization for XHTML</p>
  </rddl:resource>
```

Im HTML-Element werden drei Namespaces deklariert, XHTML (der Default-Namespace) dafür, dass es sich um eine XHTML-Seite handelt, RDDL weil es ebenso eine RDDL-Seite ist, und XLink als von RDDL benutzter Namespace. RDDL wird anschliessend verwendet, um mittels des `rddl:resource` Elements auf eine Ressource (in diesem Fall eine DTD) zu zeigen, die mit XLink-Attributen in maschinenlesbarer Weise beschrieben wird. Eingebettet in das `rddl:resource` Element ist dann eine menschenlesbare XHTML-Beschreibung des gleichen Sachverhalts. Anfangs wirkt die Mixtur aus XHTML, RDDL und XLink etwas irritierend, an sich ist RDDL aber eine sehr einfache Sprache, die es ermöglicht, Namespaces und Schemas besser zu verwalten und einfacher zugänglich zu machen.

Das Problem bei der Verwendung von RDDL ist, dass RDDL keine Unterstützung für Schema-Versionen bietet. Das lässt sich aber recht einfach nachbessern, indem man eigene Attribute (am besten aus einem eigenen Namespace) definiert, die das `rddl:resource` Element um Versionsinformation ergänzen:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:rddl="http://www.rddl.org/"
      xmlns:xlink="http://www.w3.org/1999/xlink"
      xmlns:rddl-versioning="http://example.com/rddl-versioning-extension">
      ...
  <rddl:resource xlink:type="simple"
                 xlink:title="DTD for validation"
                 xlink:arcrole="http://www.rddl.org/purposes#validation"
                 rddl-versioning:major="1" rddl-versioning:minor="5"
                 xlink:role="..."
                 xlink:href="...">
    ...
  </rddl:resource>
```

Mit dieser kleinen Erweiterung von RDDL besteht jetzt die Möglichkeit, die komplette Information, die von einem Namespace Name zu einem Schema führt, in maschinenlesbarer Weise abzulegen. Dass die Schemas, auf deren Versionen aus dem RDDL gezeigt wird, auch tatsächlich aufeinander aufbauen und dabei gewissen Regeln gehorchen (z.B. Minor Versionen für rückwärtskompatible Versionen und Major Versionen andernfalls) bleibt natürlich nach wie vor der Disziplin der Entwickler vorbehalten. Die folgenden Überlegungen zum Thema Wiederverwendung in XML Schema und Namespaces können dabei helfen.

Namespaces und Wiederverwendung

Die eigentliche Versionierung der Schemas bleibt trotz der Namespace/RDDL-Markierung natürlich eine Aufgabe der Entwickler. Dabei gibt es verschiedene Strategien, wobei davon ausgegangen werden soll, dass die grundlegenden Regeln für Erweiterbarkeit und Offenheit eingehalten werden. Diese Strategien sind allerdings davon abhängig, für welche der Varianten 1, 2 oder 3 bezüglich der Versionierung von Namespace Namen man sich entschieden hat. Hat man sich auf Variante 3 festgelegt, so muss man sich keine Gedanken machen, wie man XML Schema Komponenten in verschiedenen Namespaces wiederverwenden kann, weil ohnehin für alle Schema-Versionen der gleiche Namespace Name Verwendung findet. Bei den Varianten 1 und 2 dagegen wird diese Überlegung wichtig, weil man bedenken muss, wie man XML Schema Komponenten in verschiedenen Namespaces wiederverwenden kann.

Ausgehend von einem ursprünglichen Schema sollte eine neue Version nicht ein komplett unabhängiges Schema sein, sondern Teile wiederverwenden (per XML Schema `include` und/oder `redefine`). Wie dies geschehen kann, hängt also davon ab, in welcher Weise das Ausgangsschema modularisiert wurde. Basiert es z.B. auf einem separaten Typenmodul, so kann dieses Modul und damit die Typen in ihm wiederverwendet werden. Für diese Modularisierung ist es wichtig, den Zusammenhang mit Namespaces zu berücksichtigen. XML Schema beschränkt `include` und `redefine` darauf, dass nur Komponenten mit dem gleichen oder keinen Namespace wiederverwendet werden dürfen. Dies verleitet dazu, insbesondere bei den oben erwähnten Varianten 1 und 2 einen Ansatz zu wählen, bei dem wiederverwendete Module keinen Namespace definieren, und die in ihnen definierten Komponenten erst bei der Wiederverwendung den Namespace des sie verwendenden Schemas erhalten. Diese Design-Methode ist bekannt als "Chamäleon-Design", weil die Komponenten wiederverwendeter Module verschiedene Namespaces annehmen, je nach dem Ort ihrer Wiederverwendung.

Für Risiken und Nebenwirkungen...

Das Problem der an sich praktischen und oft verwendeten Chamäleon Schemas ist, dass die in wiederverwendete Module ausgelagerten Komponenten in Instanzen mit einem anderen Namespace auftreten, und aus der Sicht der Instanz deshalb nicht ersichtlich ist, dass es sich an sich um die gleiche Komponente handelt. Eine weitere Konsequenz der Namespace-Änderung der Komponenten in Chamäleon Schemas ist, dass Chamäleon Schemas und Identity Constraints nicht zusammen funktionieren, weil für diese die XPaths in `xpath` und `select` Attributen nicht auch entsprechend den neuen Namespaces interpretiert werden, sondern immer noch als XPaths ohne Namespace. Hier zeigt sich ein weiteres mal, das die Identity Constraints eher schräg in der XML Schema Landschaft stehen (andere Probleme mit ihnen sind *Substitution Groups* und überhaupt alles, was mit Typen zu tun hat). Zusammengefasst lassen sich also die folgenden Warnungen zu Chamäleon Schemas aussprechen:

- Der gemeinsame Ursprung wiederverwendeter Komponenten geht verloren und sollte daher auf andere Art wiederhergestellt werden (z.B. Applikationswissen).
- Identity Constraints in Chamäleon Schemas sollten vermieden werden.

Ausgehend von diesen beiden Einschränkungen lassen sich Chamäleon Schemas durchaus sinnvoll verwenden, aber es sei nochmals darauf hingewiesen, dass man sich diese (und andere) Unschönheiten und Fehlermöglichkeiten erspart, wenn man von Anfang an auf Variante 3 setzt, bei der sich der Namespace Name niemals ändert.

Informationsaustausch

In allen drei Fällen ist es natürlich notwendig, dass Informationen zwischen der XML-Instanz und der Applikation übermittelt werden, mit welchen die Instanz mitteilt, zu was für einer Version des Schemas sie gehört. Dies kann am einfachsten über ein Attribut geschehen, in welcher in der Instanz die Version des Schemas markiert wird. Im Schema wird dieses Attribut als `required` und `fixed` definiert, so dass es in jeder Instanz des Schemas vorhanden sein muss.

```
<example version="1.0" xmlns="http://example.com/vocabulary">
      ...
</example>
```

Dies ist also eine Instanz des ersten Schemas, der Namespace Name ist hier so angegeben, als wäre Variante 3 ausgewählt (anderfalls wäre wohl die Major oder die Major/Minor Versionsnummer als Teil des Namespace Name angegeben worden). Wie sehen nun Instanzen folgender Versionen aus? Für Variante 1 sieht das folgendermassen aus:

```
<example version="1.1" xmlns="http://example.com/vocabulary/1/1">
      ...
</example>
```

Wie weiter oben erwähnt, ist hier die Versionsinformation doppelt vorhanden, weil der Namespace Name sich mit jeder Schemaversion ändert. Anders bei Variante 2:

```
<example version="1.1" xmlns="http://example.com/vocabulary">
      ...
</example>
```

Der Namespace ist nach wie vor der gleiche, weil es sich um eine neue Minor Version handelt, und erst bei einer neuen Major Version wird der Namespace geändert. Variante 2 sieht bei einer neuen Major Version also folgendermassen aus:

```
<example version="2.0" xmlns="http://example.com/vocabulary/2">
      ...
</example>
```

Für diesen Fall (neue Major Version) sieht die Instanz bei Variante 3 allerdings immer noch weniger unterschiedlich aus, weil in diesem Fall der Namespace Name auch über Major Versionen hinweg erhalten bleibt, und auch die Major Version nur im `version` Attribut vermerkt wird:

```
<example version="2.0" xmlns="http://example.com/vocabulary">
        ...
</example>
```

Und wieso werden in all diesen Beispielen keine `xsi:schemaLocation` Attribute verwendet? Weil die gesamte Versionierungsinformation über die Namespace Names und RDDL erfolgt, besteht keine Notwendigkeit, direkt aus der Instanz auf das Schema zu verweisen. Der XML Schema Standard lässt XML Schema Prozessoren diverse Möglichkeiten offen, zum zu einer Instanz gehörenden Schema zu gelangen, und aus diesem Fall empfiehlt es sich ohnehin, die Regeln selber zu definieren und dem Prozessor dann explizit zu jeder Instanz das dazugehörige Schema zu übergeben.

Dynamische Versionierung

Das letzte der vorangegangenen Beispiele übergeht nun jedoch einen Aspekt, der sehr wichtig ist, nämlich dass Version 2.0 des Schemas nicht-rückwärtskompatible Änderungen des Schemas definiert hat. Wie kann dies nun eine ältere Software erfahren, die Version 1.x implementiert und keine eingebaut Kenntnis hat über die nicht-rückwärtskompatible Änderung des Schemas? Dazu gibt es grundsätzlich zwei Möglichkeiten: Die Applikation kann über den Namespace Name und das RDDL das Schema 2.0 holen, und in diesem Informationen vorfinden über nicht-rückwärtskompatible Änderungen des Schemas. Dies ist allerdings schwerfällig und vor allem davon abhängig, dass das Schema 2.0 dynamisch verfügbar ist.

Eine einfachere und pragmatischere Lösung ist, im Schema von der ersten Version an Attribute zuzulassen (in Anlehnung an SOAP hier `mustUnderstand` genannt), mit denen Elemente markiert werden können, die nicht-rückwärtskompatible Änderungen unterworfen wurden (wohlgemerkt, mit nicht-rückwärtskompatiblen Änderungen von Attributen funktioniert das dann nicht mehr). Ein Beispiel dafür ist die folgende Instanz:

```
<example version="2.0" xmlns="http://example.com/vocabulary">
        ...
        <person mustUnderstand="2.0"> ... </person>
        ...
</example>
```

Die Verarbeitungsvorschriften für das Schema verlangen nun, dass das `person` Element gemäss den der Bedeutung im Schema 2.0 verarbeitet werden muss. Ob die 1.x Software nun einen Fehler erzeugt oder aber das Element einfach ignoriert, hängt vom Anwendungsfall ab, aber es ist zumindest sichergestellt, dass keine Software, die sich an die Verarbeitungsvorschriften hält, das Element fälschlicherweise gemäss der alten Bedeutung verarbeitet.

Mit dieser Art, Bedeutung in der Instanz zu transportieren und auf diese Weise alte Software über die Bedeutung neuer Versionen zu informieren, kann die Interoperabilität beträchtlich erhöht werden. Es wird natürlich auch bei dieser Strategie immer Fälle geben, wo Software-Updates notwendig werden, aber man kann diese zumindest reduzieren. Essentiell ist für dieses Vorgehen allerdings, dass sich alle Software-Komponenten an die Verarbeitungsvorschriften halten und die Versionierungsinformation (im `version` Attribut und im `mustUnderstand` oder ähnlichen Attributen) wie vorgesehen verarbeiten.

Initialaufwand vs. Training on the Job

Die in diesem Artikel beschriebenen Technologien und Strategien, XML Schema Versionierung zu betreiben, sind sicher nicht ganz einfach und brauchen eine gewisse Vorbereitung. Es stellt sich natürlich bald einmal die Frage, ob dieser Aufwand gerechtfertigt ist. Das hängt sicher sehr stark von der Anwendung ab. in einem geschlossenen Umfeld mit kontrolliertem Software-Deployment muss weniger Rücksicht darauf genommen werden, dass sich Softwarekomponenten unabhängig voneinander entwickeln. In einem stark vernetzten und heterogenen Umfeld kann es dagegen durchaus sinnvoll sein, sich vor der Definition von Schnittstellen bereits Gedanken über deren Weiterentwicklung zu machen.

In einem Umfeld, in dem Langlebigkeit, Heterogenität und dezentrale Entwicklung wichtige Faktoren sind, kann sich der grössere Initialaufwand einer geplanten XML Schema Versionierung sehr schnell rentieren, sobald die ersten Änderungen akut werden und ein einfacheres Design bereits Nachbesserungen im Schemadesign oder sogar erzwungene Softwareupdates (mit den entsprechenden Folgekosten) zur Folge gehabt hätte.

Wie bereits in den beiden vorangegangenen Artikeln fällt das Fazit also auch diesmal so aus, dass XML Schema in dermassen vielen Anwendungsszenarien benutzt wird, dass es unmöglich ist, überall gleichermassen passende Designregeln zu definieren. Die hier vorgestellten Verfahren sind eher für anspruchsvolle Benutzer gedacht, können diesen langfristig dafür aber sehr viele sonst unvermeidliche Probleme ersparen.

Fortsetzung folgt?

Dies war der dritte und an sich letzte Teil der Artikelserie über XML Schema Design. Ein weiterer Teil wäre möglich, der sich mit den Möglichkeiten befassen könnte, wie man die bisher nur ansatzweise erwähnten Informationen zur vorwärtskompatiblen Erweiterung von Schemas so gestalten kann, dass sich die lose Kopplung der Software-Komponenten länger aufrechterhalten lässt, ohne eine harten Schnitt mit erzwungenen Softwareupdates zu machen. Sind Sie der Meinung, dass ein solcher Artikel erscheinen sollte, dann senden Sie bitte eine Email an `xsd4@dret.net` (Inhalt egal, es wird einfach gezählt).