eCH-0035: Design von XML Schemas

Name	Design von XML Schemas
Standard-Nummer	eCH-0035
Kategorie	Best Practice
Reifegrad	Experimentell
Version	1.00
Status	Aufgehoben
Beschluss am	
Ausgabedatum	
Ersetzt Standard	--
Sprachen	Deutsch (Original), Französisch (Übersetzung)
Autoren	Fachgruppe XML Erik Wilde, ETH Zürich, net.dret@dret.net
Herausgeber / Vertrieb	Verein eCH, Amthausgasse 18, 3011 Bern T 031 560 00 20, F 031 560 00 25 www.ech.ch info@ech.ch

Zusammenfassung

Das vorliegende Dokument wurde in den eCH-0018 ab Version 2.0 integriert.

Inhaltsverzeichnis

1 Status des Dokuments

Aufgehoben: Das Dokument wurde von eCH zurückgezogen. Er darf nicht mehr genutzt werden.

1.1 Terminologie der Empfehlungen

Richtlinien in diesem Dokument werden gemäss der Terminologie aus [RFC2119] angegeben, dabei kommen die folgenden Ausdrücke zur Anwendung, die durch GROSSSCHREIBUNG als Wörter mit den folgenden Bedeutungen kenntlich gemacht werden (Zitat aus [RFC2119]):

- **MUST**: This word, or the terms "**REQUIRED**" or "**SHALL**", mean that the definition is an absolute requirement of the specification.
- **MUST NOT**: This phrase, or the phrase "**SHALL NOT**", mean that that definition is an absolute prohibition of the specification.
- **SHOULD**: This word, or the adjective "**RECOMMENDED**", mean that there may exist valid reasons in particular circumstances to ignore a particular item, but the full implications must be understood and carefully weighed before choosing a different course.
- **SHOULD NOT**: This phrase, or the phrase "**NOT RECOMMENDED**" mean that there may exist valid reasons in particular circumstances when the particular behavior is acceptable or even useful, but the full implications should be understood and the case carefully weighed before implementing any behavior described with this label.
- **MAY**: This word, or the adjective "**OPTIONAL**", mean that an item is truly optional. One vendor may choose to include the item because a particular marketplace requires it or because the vendor feels that it enhances the product while another vendor may omit the same item. An implementation which does not include a particular option MUST be prepared to interoperate with another implementation which does include the option, though perhaps with reduced functionality. In the same vein an implementation which does include a particular option MUST be prepared to interoperate with another implementation which does not include the option (except, of course, for the feature the option provides.)

2 Zusammenfassung der Empfehlungen

2.1 Versionen von XML und XML Schema

Empfehlungen aus Abschnitt 4

- **SHOULD**: Da XML 1.1 nur in sehr speziellen Anwendungsfällen benötigt wird, sollten soweit möglich XML 1.0 und XML Schema 1.0 verwendet werden.
- **MAY**: Ist die Verwendung von XML 1.1 unbedingt notwendig, so kann die Verarbeitung von XML 1.1 in der durch eine W3C Note [xml11schema10] beschriebenen Weise mit XML Schema 1.0 kombiniert werden.

2.2 XML Schema und XML Namespaces

Empfehlung aus Abschnitt 5

- **SHOULD**: Der Namespace Name eines Schemas (wie im `targetNamespace` Attribut angegeben) sollte, wie in [eCH-0018] verlangt, nicht direkt auf das Schema zeigen, sondern statt dessen auf eine Beschreibung des definierten Namespaces gemäss [eCH-0033]. In dieser Beschreibung finden sich dann neben anderen Informationen Verweise auf das Schema, so dass die Auffindbarkeit des Schemas gewährleistet ist.

2.3 Modellierung mit XML Schema

Empfehlungen aus Abschnitt 6

- **SHOULD**: Bei der maschinellen Verarbeitung von getypten Daten (d.h. nahezu allen Daten in B2B Szenarien) sollte das Schema mittels eines XML Schema definiert werden, da nur in diesem die Datentypen in einer anwendungsnahen Form definiert werden können.
- **MAY**: Handelt es bei den ausgetauschten Daten um mehrheitlich ungetypte Daten (text-orientierte Dokumente), so kann die Verwendung von DTDs ebenfalls angemessen sein, es ist jedoch auch in diesem Fall darauf zu achten, dass DTDs deutlich weniger strenge Randbedingungen unterstützen als XML Schema.

2.4 Lokale/Globale Deklaration von Elementen/Typen

Empfehlungen aus Abschnitt 7

- **SHOULD**: Das vorherrschende Design-Muster sollte das Venetian Blinds Design sein, das Typen generell global, und Elemente und Attribute je nach Bedarf lokal oder global definiert (global nur dann, wenn sie an verschiedenen Stellen wiederverwendet werden). Typen sind also der Aspekt eines Schemas, der speziell hervorge-

hoben wird und durch die generell globale Definition zur Wiederverwendung gut geeignet ist.

- **SHOULD NOT**: Bei der lokalen Definition von Elementen oder Attributen sollte darauf geachtet werden, dass keine Elemente oder Attribute gleichen Namens lokal definiert werden, weil dies bei Benutzern des Schemas Irritationen erzeugen kann. Handelt es sich um das gleiche Element, sollte eine globale Definition erfolgen, die referenziert wird. Handelt es sich um unterschiedliche Konzepte, sollten diese nicht mit dem gleichen Namen benannt werden.
- **MAY**: Sollen Elemente wiederverwendet werden, so können auch diese global definiert werden (dies ist z.B. bei Rekursionen unerlässlich). Dabei ist jedoch darauf zu achten, dass im Venetian Blinds Design die Typen die primäre Art der Wiederverwendung darstellen, so dass jeder dieser Fälle einzeln betrachtet werden sollte.
- **SHOULD**: Soll ein Typ in Zusammenhang mit Identity Constraints (wie in Abschnitt 9.5 beschrieben) wiederverwendet werden, so sollte dies deutlich im Typ vermerkt werden. Dies kann am besten dadurch dokumentiert werden, dass ein Beispielelement dieses Typs definiert wird, das die Identity Constraints enthält. Im Typ kann dann durch einen Kommentar darauf verwiesen werden, dass bei einer Wiederverwendung des Typs die Identity Constraints aus dem Beispielelement kopiert werden müssen.

2.5 Namens- vs. typbasierte Verarbeitung

Empfehlung aus Abschnitt 8.1

- **MUST**: .Werden in einem XML Schema Mechanismen verwendet, die beim Arbeiten mit Instanzen notwendig machen, Typinformationen zu verwenden (es sind dies die in Abschnitt 8.2 beschriebenen Mechanismen), so ist darauf im Schema und in der Dokumentation deutlich hinzuweisen.

2.6 Type Substitution Mechanismen in XML Schema

Empfehlung aus Abschnitt 8.2

- **MUST**: Wird einer der beiden Mechanismen eingesetzt (`xsi:type` oder Ersetzungsgruppen), so muss nur dieser Mechanismus eingesetzt werden. Eine Mischung beider Stile würde ein Schema schwer zu verstehen und den Umgang damit unnötig kompliziert machen. Dies ist keine Einschränkung, da die beiden Mechanismen im Kern ohnehin das gleiche machen, nämlich andere Typen in einem Inhaltsmodell zu ermöglichen, als den Typ des im Inhaltsmodell aufgeführten Elements.

2.7 Das xsi:type Attribut

Empfehlungen aus Abschnitt 8.2.1

- **SHOULD NOT**: Generell sollten Mechanismen der `xsi:type` Type Substitution vermieden werden, solange nicht gewichtige Gründe dafür sprechen. Diese Mechanismen erschweren die Verarbeitung und das Verständnis des Schemas und der durch das Schema definierten Instanzen.
- **MUST**: Ist im Schema explizit dokumentiert oder sogar durch das Schema-Design erzwungen (Typen als `abstract="true"` definiert), dass `xsi:type` verwendet werden kann oder soll, so ist dies zulässig, muss aber angemessen dokumentiert werden. In der Verarbeitung dieser Instanzen ist dann darauf zu achten, dass diese typbasiert erfolgt, so dass die Typzuweisung nicht über den Typ eines Elements im Schema, sondern über den `xsi:type` aus der Instanz erfolgt.
- **MUST**: Werden Mechanismen der Type Substitution verwendet, so ist dies deutlich und angemessen zu dokumentieren, so dass Benutzer des Schemas deutlich über diesen Aspekt des Schemas informiert werden.

2.8 Type Substitution

Empfehlungen aus Abschnitt 8.2.2

- **SHOULD NOT**: Generell sollten Mechanismen der Substitution Groups vermieden werden, solange nicht gewichtige Gründe dafür sprechen. Diese Mechanismen erschweren die Verarbeitung und das Verständnis des Schemas und der durch das Schema definierten Instanzen.
- **SHOULD**: Ist im Schema explizit dokumentiert oder sogar durch das Schema-Design erzwungen (Elemente, die Substitution Group Heads sind, als `abstract="true"` definiert), dass Substitution Groups verwendet werden können oder sollen, so ist dies zulässig, muss aber angemessen dokumentiert werden. In der Verarbeitung dieser Instanzen ist dann darauf zu achten, dass diese konsequent auf die Substitution Groups Rücksicht nimmt.
- **MUST**: Werden Mechanismen der Substitution Groups verwendet, so ist dies deutlich und angemessen zu dokumentieren, so dass Benutzer des Schemas deutlich über diesen Aspekt des Schemas informiert werden.

2.9 Root Elemente

Empfehlungen aus Abschnitt 9.1

- **MAY**: Schemas dürfen so designt werden, dass mehrere Elemente als Root Elemente vorkommen dürfen. Auf diese Weise kann ein Schema dazu dienen, Instanzen mit verschiedenen Root Elementen zu validieren.

- **SHOULD**: Root Elemente in einem Schema sollten markiert werden. Da es dafür keinen durch XML Schema definierten Mechanismus gibt, muss dies in einem Kommentar geschehen, der die potentiellen Root Elemente kennzeichnet.

2.10 Verarbeitungseinheiten

Empfehlung aus Abschnitt 9.2

- **SHOULD**: Die Verarbeitungseinheiten, die bei der Definition eines Schemas gewählt werden, sollten sich daran orientieren, in welcher Weise die Daten aus fachlicher Sicht strukturiert sind. Die Wahl der Verarbeitungseinheiten sollte das Ziel haben, die für die Verarbeitung relevanten Strukturen durch XML Markup zu kennzeichnen, so dass Applikationen soweit möglich keine über XML-Technologien hinausgehenden Methoden brauchen, um die fachlich relevanten Strukturen zu identifizieren und mit ihnen zu arbeiten.

2.11 Elemente oder Attribute?

Empfehlungen aus Abschnitt 9.3

- **SHOULD**: Generell sollten Elemente zur Strukturierung verwendet werden, da diese im Bedarfsfall einfacher erweitert werden können, z.B. mit zusätzlichen Attributen, oder durch die Unterteilung in feiner gegliederte Strukturen.
- **MAY**: Attribute sollten mit Vorsicht verwendet werden. Sie unterliegen deutlichen Einschränkungen (nicht wiederholbar, keine Möglichkeit der weiteren Strukturierung) und bilden daher bei späteren Änderungen einen Schemas u.U. eine Stelle, an der gewünschte Änderungen nicht mehr einfach durchführbar sind

2.12 Repräsentation leerer Werte

Empfehlungen aus Abschnitt 9.4

- **SHOULD**: Die Darstellung von Null-Werten sollte mit optionalen Elementen oder Attributen erfolgen, d.h. durch Nichterscheinen der jeweiligen Komponenten, oder durch leeren Inhalt der jeweiligen Komponenten.
- **SHOULD NOT**: Der `xsi:null` Mechanismus sollte vermieden werden.

2.13 Identifikation und Referenzen

Empfehlungen aus Abschnitt 9.5

- **SHOULD**: Soll die Eindeutigkeit, Existenz, oder Referenzierung von Namen garantiert werden, so sollten XML Schema Identity Constraints benutzt werden, um diese Constraints zu definieren. Dabei ist darauf zu achten, dass die XPaths so restriktiv

wie möglich definiert werden, um keine Konflikte bei späteren Änderungen des Schemas zu riskieren.

- **SHOULD**: Werden Identity Constraints verwendet, so sollten die Strukturen, auf die sie verweisen, eigene Typen für die Namen verwenden. Auf diese Weise können die Typen einfach wiederverwendet werden, wenn die entsprechenden Namen an anderer Stelle des Schemas wiederverwendet werden sollen.
- **SHOULD NOT**: Der DTD Mechanismus der `ID/IDREF` Attribute sollte nicht verwendet werden, da er starken Einschränkungen unterliegt und es zudem nicht erlaubt, die Constraints so genau zu definieren, wie es mit den Identity Constraints möglich ist.
- **MAY**: Sind durch die Applikation Anforderungen gegeben, die sich durch XML Schemas Identity Constraints nicht implementieren lassen (z.B. Eindeutigkeit oder Referenzierung über mehrere Dokumente hinweg), so können diese durch externe Identifikation und externe Constraints definiert werden. Für die durch diese externen Definitionen erfassten Strukturen sollten dann eigene Typen verwendet werden, und im Schema sollte deutlich dokumentiert sein, dass die Werte dieser Typen Constraints unterworfen sind, die ausserhalb des Schemas definiert sind.
- **SHOULD**: Sollen Identity Constraints in zur Wiederverwendung bestimmten Typen vorkommen, so kann dies am besten dadurch dokumentiert werden, dass ein Beispielelement dieses Typs definiert wird, das die Identity Constraints enthält. Im Typ kann dann durch einen Kommentar darauf verwiesen werden, dass bei einer Wiederverwendung des Typs die Identity Constraints aus dem Beispielelement kopiert werden müssen.

2.14 Wertelisten

Empfehlungen aus Abschnitt 9.6

- **SHOULD**: Handelt es sich bei Wertelisten um statische Listen, deren Werte abschliessend bekannt und aller Voraussicht nach stabil sind, so sollten diese Werte im Schema mit einem Simple Type und Enumeration Facets aufgezählt werden. Um eine solche Werteliste besser zu verwalten und bessere Zugriffskontrollen durchführen zu können, ist es im Allgemeinen sinnvoll, sie in ein eigenständiges Schema-Dokument auszulagern (siehe Abschnitt 11.1).
- **SHOULD**: Handelt es sich bei Wertelisten um dynamische Listen, deren Werte nicht abschliessend bekannt sind oder sich verändern können, so sollten sie nur durch eine möglichst exakte lexikalische Einschränkung und den Verweis auf eine externe Liste definiert werden. Auf diese Weise können den Wertelisten neue Werte hinzugefügt werden, ohne dass sich am Schema etwas ändern muss.

2.15 Sprachmarkierung von Inhalten

Empfehlung aus Abschnitt 9.7

- **SHOULD**: Sollen Inhalte mit Sprachmarkierungen versehen werden, so sollten diese Markierungen in einem `xml:lang` Attribut angebracht werden, da dieses im XML Standard selber definiert ist und eine allgemeine Konvention für Sprachmarkierungen darstellt. In [eCH-0050] wird ein entsprechendes Schema definiert, das zur Wiederverwendung im Rahmen von eCH bereitgestellt wird.

2.16 Versionierung von XML Schemas

Empfehlungen aus Abschnitt 10

- **SHOULD**: Ändert sich ein Schema in einer Weise, dass bei der Validierung und Interpretation gemäss dem neuen Schema auch Instanzen nach dem alten Schema korrekt verarbeitet werden können, so sollte eine neue Minor Version erzeugt werden.
- **SHOULD**: Ändert sich ein Schema in einer Weise, dass bei der Validierung und Interpretation gemäss dem neuen Schema Instanzen nach dem alten Schema nicht mehr korrekt verarbeitet werden können, so sollte eine neue Major Version erzeugt werden.
- **SHOULD NOT**: Sind vormals erlaubte Werte oder Strukturen in einer neuen Version verboten, so sollten sie nicht in der Schema-Definition verboten werden, sondern auf der Anwendungsebene als deprecated markiert werden, so dass alte Instanzen weiterhin verarbeitet werden können.
- **MUST**: Um die Markierung der Minor Version eines Schemas und dessen Instanzen zu ermöglichen (die Major Version wird über den Namespace Name markiert), wird das `version` Attribut des Schemas verwendet, das nur die Minor Version enthält. Die Minor Version in Instanzen wird durch eine Markierung vorgenommen, die im Schema selbst vorgesehen sein muss, am besten aber durch die Verwendung des in [eCH-0050] vorgesehenen Attributs erfolgen sollte (dieses Attribut ist ebenso durch [eCH-0018] empfohlen).
- **MUST**: Handelt es sich um einen bidirektionalen Datenaustausch, so muss bei einem Austausch die auf beiden Seiten unterstützte Major Version übereinstimmen. Ist dies nicht der Fall, kann die Kommunikation nicht stattfinden.
- **MAY**: Handelt es sich um einen primär unidirektionalen Datenaustausch, so kann bei der Versionierung u.U. darauf Rücksicht genommen werden und es können bei einer Versionierung durch eine Minor Version Änderungen akzeptiert werden, solange diese auf der Empfängerseite nicht zu Kompatibilitätsproblemen führen. In diesem Fall ist eine gute Dokumentation nötig, die das Szenario deutlich beschreibt und damit die Fälle definiert, in denen eine Kommunikation möglich ist.

- **MAY**: Die obigen Empfehlungen sind nur für produktiv eingesetzte und entwickelte Schemas anwendbar. Für die Entwicklungsphase vor dem produktiven Einsatz und für Schemas ausserhalb des produktiven Einsatzes ist ein einfacheres Vorgehen möglich.

2.17 Strukturierung von Schema-Dokumenten

Empfehlungen aus Abschnitt 11.1

- **SHOULD**: Aus Gründen der Modularisierung sollten Schema-Dokumente, die über eine gewisse Grösse hinausgehen, unter Verwendung von `xs:include` strukturiert werden. Dies erlaubt übersichtlichere Schemas und zudem die Wiederverwendung von Teilen des Schemas (z.B. Wertelisten, wie in Abschnitt 9.6 beschrieben).
- **SHOULD NOT**: Der `xs:redefine` Mechanismus von XML Schema sollte vermieden werden, da er zu komplizierten Abhängigkeiten zwischen verschiedenen Schema-Dokumenten führt und vor allem bei späteren Änderungen von Schema-Dokumenten zu schwierig auffindbaren Fehler führen kann.
- **SHOULD NOT**: Chamäleon Schemas (d.h. Schema-Dokumente ohne `targetNamespace`, die daher den Namespace des Schema-Dokumenten annehmen, in das sie eingebunden werden), sollten vermieden werden, da sie diverse Fehlermöglichkeiten mit sich bringen und bestehende Gemeinsamkeiten nicht offenlegen.
- **MAY**: Arbeiten verschiedene Teile eines Teams an der Entwicklung oder Weiterentwicklung eines Schemas, so kann dies ebenfalls ein Grund für die Modularisierung eines Schemas sein. In diesem Fall wird sich die Modularisierung an der inhaltlichen Zuteilung der verschiedenen Teile des Schemas an verschiedene Teile des Teams orientieren.

2.18 Strukturierung von Schemas

Empfehlungen aus Abschnitt 11.2

- **SHOULD**: Soweit möglich, sollten in einem neuen Schema existierende Lösungen für Teilaspekte des Schemas durch das Importieren der betreffenden Schemas wiederverwendet werden. Dies erleichtert die Entwicklung von Schemas, und ist auch für die Anwender eines Schemas eine Erleichterung, weil diese bereits bekannte Schemas benutzen und auf diese Weise bestehendes Know-how und bestehenden Code wiederverwenden können.
- **SHOULD**: Handelt es sich bei der Entwicklung von Schemas um unterschiedliche Teams oder Projekte, so sollten diese nicht ein gemeinsames Schema entwickeln, sondern verschiedene Schemas. Auf diese Weise ist die Entwicklung besser entkoppelt, und mögliche Probleme durch verschieden schnellen Fortschritt in den Teams oder Projekten werden umgangen.

2.19 Implementierung referentieller Beziehungen

Empfehlungen aus Abschnitt 12

- **MAY**: Ist es aus Sicht von Implementierungsüberlegungen angebracht, auf hierarchische Darstellungen zu verzichten (weil beispielsweise Datenproduktion und -konsum immer auf der Basis eines RDBMS stattfinden), so ist es erlaubt, auf die hierarchische Darstellung von Daten zu verzichten. Dies sollte jedoch nur mit Vorsicht und Bedacht geschehen, weil sich die durch die Implementierungsüberlegungen gegebenen Randbedingungen ändern können (z.B. bei der Umstellung auf XML-Datenbanken), und das Schema dann nicht mehr zur geänderten Umgebung passt.
- **SHOULD**: Werden Referenzen verwendet, so sollten die Randbedingungen der Referenzen so gut wie möglich mit Hilfe von Identity Constraints beschrieben werden (mit den in Abschnitt 9.5 beschrieben Vorbehalten), weitergehende Randbedingungen sollten dokumentiert werden.
- **SHOULD**: Sind Schemas hauptsächlich auf Endbenutzer ausgerichtet, z.B. im Fall von XML-Dokumenten, die von Menschen verwendet (betrachtet oder editiert) werden, z.B. Konfigurationsdateien oder Web-orientierte Dokumente, so sollte die Modellierung an den entsprechenden Stellen eher hierarchisch sein. Hierarchische Strukturen sind für Menschen einfacher zu verstehen als durch viele Referenzen miteinander vernetzte Strukturen.
- **SHOULD**: Sind Schemas hauptsächlich für die maschinelle Weiterverarbeitung bestimmt, so kann das Modell eher flacher definiert werden als für den hauptsächlich auf Endbenutzer ausgerichteten Fall. Allerdings sollte an den Stellen, wo es eine fachlich begründete, dem Basismodell innewohnende hierarchische Struktur gibt, auch eine solche Struktur im XML definiert werden.

2.20 Offene XML Schemas

Empfehlungen aus Abschnitt 13.1

- **MAY**: Ist ein offenes Schema erwünscht, so dass in Instanzen des Schemas Inhalte erscheinen dürfen, die nicht detailliert im Schema definiert sind, so kann die über Wildcards für Elemente und/oder Attribute erreicht werden. Diese Wildcards können einerseits dazu dienen, Offenheit gegenüber nicht bekannten Inhalten zu bewahren, andererseits aber auch dazu, Offenheit gegenüber vorhergesehenen zukünftigen Erweiterungen zu erreichen.

3 Einleitung

Im eCH-Dokument "XML Best Practices " [eCH-0018] werden Richtlinien zum Gebrauch von XML Dokumenten im Allgemeinen, von XML Dokumenten als Instanzen bestehender XML Schemas, und zur Benennung von Komponenten in XML Schemas gemacht. Damit werden die wesentlichen Richtlinien definiert, die für Benutzer bestehender XML Schemas einen Rahmen vorgeben.

Das vorliegende Dokument geht über diese Richtlinien hinaus und legt fest, welche Empfehlungen beim Design von XML Schemas eingehalten werden sollten. Das Ziel dieser Empfehlungen ist es, XML Schemas zu definieren, die für den Verwender eines solchen Schemas möglichst einfach zu verstehen sind, und die Schemas auf eine Weise zu definieren, dass die Wiederverwendung von Schema-Modulen ermöglicht und gefördert wird.

3.1 Überblick

Mit XML Schema hat sich neben einer erhöhten Leistungsfähigkeit gegenüber den Document Type Definitions (DTDs) vor allem auch einen massiven Zuwachs an Komplexität in der Schemasprache ergeben. Dies liegt daran, dass durch die grössere Anzahl an Sprachmitteln nun wesentlich mehr Implementierungsmöglichkeiten für Schemadefinitionen bieten.

Dies hat den Vorteil, dass die unterschiedlichen Randbedingungen der verschiedenen Implementierungsmöglichkeiten gegeneinander abgewogen werden können und dann die Variante gewählt werden kann, die den Wünschen entspricht. Andererseits ist es durch die grosse Komplexität von XML Schema nur wenigen Benutzern möglich, die Randbedingungen alle zu erkennen und in ihren Wirkungen und Wechselwirkungen einschätzen zu können.

Aus diesem Grund wird XML Schema oft nur zögerlich oder auch nicht optimal eingesetzt, weil die Einarbeitung und das Arbeiten mit XML Schema einiges an Aufwand bedeuten. Um diese Aufgaben zu vereinfachen, legt das vorliegende Dokument Richtlinien fest, an Hand derer in vielen Fällen bei verschiedenen möglichen Modellierungsvarianten eine Hilfe geboten werden soll, welche Variante ausgewählt werden sollte.

3.2 Anwendungsgebiet

Das Anwendungsgebiet dieses Dokuments ist das Design von XML Schemas, also nicht die Frage, wie Markup designt werden sollte (diese Fragen werden im eCH-Dokument "XML Best Practices " [eCH-0018] diskutiert), sondern wie die innere Struktur des XML Schemas ist, der Aufbau und der Zusammenhang der sogenannten "XML Schema Komponenten".

Dies ist vor allem dann wichtig, wenn nicht nur Instanzen des Schemas verarbeitet werden sollen, sondern wenn das Schema selbst wiederverwendet werden soll, z.B. indem Teile in einem neuen Kontext wiederverwendet werden sollen, oder indem eine neue Version des

Schemas definiert werden soll. In beiden Fällen wirkt sich die innere Struktur des Schemas stark darauf aus, wie einfach diese Aufgabe ausgeführt werden kann.

3.3 Vorteile

Werden Schemas von Anfang an auf ihre Wiederverwendung hin definiert, so fördert dies ihre Wiederverwendung und hilft damit, das Entstehen von parallelen Lösungen zu vermeiden. Bei der Definition von Schemas sollte immer die Perspektive verfolgt werden, dass dieses Schema als Baustein in einem nicht vorhergesehenen Anwendungsfall nützlich werden könnte, und für diesen Fall sollte das Schema designt werden.

3.4 Schwerpunkte

Der Schwerpunkt des vorliegenden Dokuments sind Richtlinien und Hinweise für das Design von XML Schemas. Die Frage des "guten" und "schlechten" Designs von XML Schema ist eine bisher auch in der Fachwelt nur unzulänglich diskutierte Frage, und aus diesem Grund kann das vorliegende Dokument keine unanfechtbaren Regeln aufstellen.

Stattdessen wird versucht, bei den einzelnen Richtlinien durch eine Begründung darauf hinzuweisen, warum diese Richtlinie existiert, und falls die Argumentation in einem speziellen Anwendungsfall nicht zutreffend ist, so sollte in diese Spezialfällen auch durchaus eine andere Entscheidung getroffen werden.

4 Versionen von XML und XML Schema

XML gibt es mittlerweile in zwei Versionen (1.0 und 1.1, wie in Abschnitt 4.1 beschrieben), und da XML Schema eng mit XML zusammenhängt, stellt sich die Frage, wie sich die Versionen von XML auf die Verwendung von XML Schema (das es bisher nur in einer Version gibt) auswirken.

4.1 Versionen von XML

XML wurde in der ersten Version 1.0 1998 standardisiert, mittlerweile gibt es eine "Third Edition" dieses Standards [xml10third], die 2004 erschienen ist, aber die unterschiedlichen "Editions" des Standards machen keinen funktionalen Unterschied aus. "Editions" von W3C Standards bedeuten lediglich, dass redigierte Versionen des gleichen Standards veröffentlich werden, in denen Fehler und unklare oder missverständliche Formulierungen beseitigt wurden.

Ebenfalls 2004 wurde aber eine neue Version 1.1 des XML Standards veröffentlicht [xml11], grösstenteils für eine Harmonisierung mit einer neuen Unicode-Version, und des weiteren wurden die Namensvorschriften für XML Namen gelockert und ein weiteres Zeichen wurde als Zeilenende zugelassen. Obwohl die Änderungen von XML 1.1 nicht sehr gross sind, beeinflussen sie doch die Definition von well-formed Dokumenten, und aus diesem Grund musste eine neue Version von XML definiert werden. Aufgrund der Regeln, wie sich ein XML 1.0 Prozessor zu verhalten halt, wird er jedes XML 1.1 Dokument zurückweisen, alleine schon deswegen, weil in der XML Deklaration `version="1.1"` steht (falls diese vorhanden ist).

Weil der XML Namespaces Standard [xmlns] eng mit XML verbunden ist, und direkt an die XML Version gebunden ist, musste für XML 1.1 auch eine neue Version des XML Namespaces Standards [xmlns11] geschaffen werden. Diese bietet als einzige funktionale Neuigkeit die Möglichkeit, die Deklaration von Namespaces rückgängig zu machen (was in XML Namespaces 1.0 nur für den Default Namespace erlaubt ist), ansonsten ist es lediglich eine Anpassung von XML Namespaces an XML 1.1.

4.2 Versionen von XML Schema

XML Schema ist zuerst 2001 veröffentlicht worden, und nachdem einige Fehler und offene Fragen in dieser Version entdeckt wurden (vor allem im Zusammenhang mit der Arbeit an XQuery, das stark auf XML Schema beruht), gab es auch von diesem Standard eine "Second Edition" [xmlschema1sec, xmlschema2sec], die 2004 veröffentlicht wurde. Wie bereits bei der Frage der XML-Versionen erwähnt, handelt es sich hierbei also nicht um eine neue Version von XML Schema, sondern lediglich um eine redigierte und verbesserte Fassung des ursprünglichen Standards. XML Schema existiert daher nur in einer Version, und diese bezieht sich explizit auf XML 1.0 und XML Namespaces 1.0, d.h. streng genommen kann man mit einem aktuellen XML Schema Prozessor keine XML 1.1 Dokumente validieren.

4.3 Versionierungsproblematik

Aus der Tatsache, dass XML in zwei Versionen existiert (Abschnitt 4.1), XML Schema jedoch nur in einer (Abschnitt 4.2), folgt, dass die Verarbeitung von XML 1.1 Dokumenten mit XML Schema momentan nicht möglich ist. In einem informellen Dokument des W3C [xml11schema10] wird jedoch beschrieben, wie sich bestehende Implementierungen von XML Schema Prozessoren so anpassen liessen, dass sie zur Validierung von XML 1.1 Dokumenten verwendet werden könnten. Die beschriebenen Änderungen sind sehr gering (anpassen des XML Parsers, der als Frontend dient, und anpassen der XML 1.0 spezifischen Definitionen von Namen und einigen anderen XML Syntax-Konstrukten), so dass der Aufwand zum Ändern eines XML Schema Prozessors relativ klein ist.

Aus diesem Grund stehen der Kombination von XML 1.1 Dokumenten mit XML Schema aus technischer Sicht keine grossen Hindernisse im Weg, es sollte jedoch bedacht werden, dass die Verwendung von XML 1.1 vor allem aus Gründen der Interoperabilität ein Problem darstellt, und aus diesem Grund [eCH-0018] als Empfehlung ausspricht, XML 1.1 ausschliesslich dann zu verwenden, wenn sich keine XML 1.0 basierte Lösung finden lässt.

4.4 Empfehlungen

- **SHOULD**: Da XML 1.1 nur in sehr speziellen Anwendungsfällen benötigt wird, sollten soweit möglich XML 1.0 und XML Schema 1.0 verwendet werden.
- **MAY**: Ist die Verwendung von XML 1.1 unbedingt notwendig, so kann die Verarbeitung von XML 1.1 in der durch eine W3C Note [xml11schema10] beschriebenen Weise mit XML Schema 1.0 kombiniert werden.

5 XML Schema und XML Namespaces

XML Schema ist eine Sprache, mit der Vokabulare für XML Dokumente definiert werden. XML Namespaces [xmlns] haben sich als Mechanismus etabliert, um in XML Dokumenten verwendete Vokabulare mit einem eindeutigen Namen zu kennzeichnen, dem Namespace Namen. Der Namespace Name ist eine URI [RFC3986] und sollte im Kontext von eCH auf eine Namespace Beschreibung gemäss [eCH-0033] verweisen (in der dann auf das Schema verwiesen wird).

XML Schema ermöglicht, mit dem `targetNamespace` Attribut im Schema selber bereits den Namespace des definierten Vokabulars anzugeben. XML Schema verlangt dies sogar, das heisst, wenn das Vokabular in Instanzen mit diesem Namespace verwendet wird, so muss es im Schema auch mit diesem `targetNamespace` definiert sein, ansonsten schlägt die Validierung der Instanz gegen das Schema fehl.

Im folgenden Beispiel (dem XML Schema für das eCH `minorVersion` Attribut) ist die Verwendung des `targetNamespace` Attributes illustriert:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" version="0"
           targetNamespace="http://www.ech.ch/xmlns/minorversion/v1">
  <xs:attribute name="minorVersion" type="xs:token"/>
</xs:schema>
```

In diesem Schema wird ein Attribut definiert, dass als dem XML Namespace `http://www.ech.ch/xmlns/minorversion/v1` zugehörig definiert wird. Aus diesem Grund muss in jeder Instanz, die dieses Attribut verwendet, auch der entsprechende Namespace deklariert und für das `minorVersion` Attribut verwendet werden, andernfalls kann das Attribut nicht gegen das hier gezeigte Schema validiert werden.

5.1 Empfehlungen

- **SHOULD**: Der Namespace Name eines Schemas (wie im `targetNamespace` Attribut angegeben) sollte, wie in [eCH-0018] verlangt, nicht direkt auf das Schema zeigen, sondern statt dessen auf eine Beschreibung des definierten Namespaces gemäss [eCH-0033]. In dieser Beschreibung finden sich dann neben anderen Informationen Verweise auf das Schema, so dass die Auffindbarkeit des Schemas gewährleistet ist.
- **SHOULD NOT**: Es sollte nicht versucht werden, den gesamten Namespace im Schema selbst zu dokumentieren (mit `documentation` Elementen) und damit das Schema als seine eigene Dokumentation zu verwenden. Der Grund dafür ist, dass die angemessene Dokumentation eines Namespaces mehr als nur das Schema umfasst (z.B. begleitende Beispieldokumente), und daher auch nicht auf das Schema alleine reduziert werden sollte.

6 Modellierung mit XML Schema

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 [xmlschema2sec]), und zum anderen die Einführung einer Typschicht (Teil 1 des Standards [xmlschema1sec]), was gegenüber DTDs eine komplett neue Modellierungsebene ergibt (siehe Abbildung 1), die eine Abstraktionsschicht von Typen über den Elementen und Attributen der DTDs legt. Elemente und Attribute sind in XML Schema immer Instanzen von Typen, die definieren, welche Inhalte die Elemente oder Attribute haben dürfen.

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

Abbildung 1: Modellierungsebenen von DTDs und XML Schema im Vergleich

Die Typebene von XML Schema wurde in DTDs oftmals mit Parameter Entities (Entities, die nur in DTDs verwendbar sind, und als `%name;` referenziert werden) „simuliert“, jedoch war dies 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).

6.1 Unterschiede in der Verarbeitung

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 stösst man allerdings schnell 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. Es ist absehbar, dass in diesem Bereich weitere Entwicklungen stattfinden und auch auf Schemainformationen über ein standardisiertes Interface zugegriffen werden kann, aber diese

Entwicklung ist noch nicht abgeschlossen. In der Realität sieht es daher so aus, dass im Grossteil aller Anwendungen nicht sauber typbasiert gearbeitet wird, sondern die Typinformation als fest verdrahtet mit den Elementen angesehen wird. Diese Annahme wird besonders dann interessant und problematisch, wenn es darum geht, XML Schemas so zu gestalten, dass sie (und die darauf aufbauende Software) langlebig und wiederverwendbar sind.

6.2 Empfehlungen

Wie in [eCH-0018] festgelegt, sollte wenn immer möglich der Verarbeitung auf der Basis von XML Schema der Vorzug gegeben werden.

- **SHOULD**: Bei der maschinellen Verarbeitung von getypten Daten (d.h. nahezu allen Daten in B2B Szenarien) sollte das Schema mittels eines XML Schema definiert werden, da nur in diesem die Datentypen in einer anwendungsnahen Form definiert werden können.
- **MAY**: Handelt es bei den ausgetauschten Daten um mehrheitlich ungetypte Daten (text-orientierte Dokumente), so kann die Verwendung von DTDs ebenfalls angemessen sein, es ist jedoch auch in diesem Fall darauf zu achten, dass DTDs deutlich weniger strenge Randbedingungen unterstützen als XML Schema.

7 Lokale/Globale Deklaration von Elementen/Typen

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 (als Kind des `xs:schema` Elements) 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 eigenständige Konstrukte (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.

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>
```

7.1 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>
```

7.2 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 sich 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>
```

7.3 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>
```

7.4 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" minOc-
curs="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>
```

7.5 Diskussion der Varianten

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 (beschrieben in Kapitel 7), die eben insbesondere beim Zusammenspiel mehrerer Schemas interessant werden.

7.5.1 Empfehlungen

- **SHOULD**: Das vorherrschende Design-Muster sollte das Venetian Blinds Design sein, das Typen generell global, und Elemente und Attribute je nach Bedarf lokal oder global definiert (global nur dann, wenn sie an verschiedenen Stellen wiederverwendet werden oder werden können sollen). Typen sind also der Aspekt eines Schemas, der speziell hervorgehoben wird und durch die generell globale Definition zur Wiederverwendung gut geeignet ist.
- **SHOULD NOT**: Bei der lokalen Definition von Elementen oder Attributen sollte darauf geachtet werden, dass keine Elemente oder Attribute gleichen Namens lokal definiert werden, weil dies bei Benutzern des Schemas Irritationen erzeugen kann. Handelt es sich um das gleiche Element, sollte eine globale Definition erfolgen, die referenziert wird. Handelt es sich um unterschiedliche Konzepte, sollten diese nicht mit dem gleichen Namen benannt werden.
- **MAY**: Sollen Elemente wiederverwendet werden, so können auch diese global definiert werden (dies ist z.B. bei Rekursionen unerlässlich). Dabei ist jedoch darauf zu

achten, dass im Venetian Blinds Design die Typen die primäre Art der Wiederverwendung darstellen, so dass jeder dieser Fälle einzeln betrachtet werden sollte.

- **SHOULD**: Soll ein Typ in Zusammenhang mit Identity Constraints (wie in Abschnitt 9.5 beschrieben) wiederverwendet werden, so sollte dies deutlich im Typ vermerkt werden. Dies kann am besten dadurch dokumentiert werden, dass ein Beispielelement dieses Typs definiert wird, das die Identity Constraints enthält. Im Typ kann dann durch einen Kommentar darauf verwiesen werden, dass bei einer Wiederverwendung des Typs die Identity Constraints aus dem Beispielelement kopiert werden müssen.

8 Type Substitution

XML Schema hat, wie in Abschnitt 6 beschrieben, mit der Typenebene eine komplett neue Modellierungseben für die Beschreibung und Verarbeitung von XML-Dokumenten geschaffen. Aus diesem Grund gibt es bei der Sicht auf ein durch ein XML Schema validiertes Dokument zwei Sichten, zum einen die Namenssicht, die die im Markup sichtbaren Element- und Attribut-Namen berücksichtigt, und die Typsicht, die die bei der Validierung festgestellten Typen dieser Elemente und Attribute berücksichtigt.

Anwendungen, die mit XML Schema und XML arbeiten, haben im Prinzip die Wahl, ob sie mit der Namens- oder der Typsicht arbeiten wollen. Abschnit 8.1 beschreibt, welche Konsequenzen diese beiden Arbeitsweisen haben. XML Schema bietet Möglichkeiten, die starre Zuordnung von Namen und Typen zu durchbrechen. Diese Mechanismen sind in Abschnitt 8.2 beschrieben, sie bieten einige interessante Modellierungsmöglichkeiten, machen jedoch ein Schema komplizierter nachzuvollziehen und die Arbeit mit den Instanzen ein wenig anspruchsvoller.

8.1 Namens- vs. typbasierte Verarbeitung

In diesem Kapitel wird diskutiert, was es über die reine Element/Typ-Beziehung (wie in Abschnitt 7 diskutiert) für Beziehungen zwischen XML Schema Komponenten geben kann. 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 man es oft sogar machen sollte, wird in Abschnitt 13 beschrieben), so muss man das explizit tun. Die folgenden Beziehungen sind von Interesse:

- *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 eine Instanz eines solchen Typs 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 verwen-

det wurde, ist nicht anzusehen, dass auch andere Elemente vorkommen können. Will man den Aufwand für die Unterstützung von Ersetzungsgruppen bei der Implementierung 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 diskutierten 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 Mechanismus 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 in Abschnitt 6.1 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 die in Abschnitt 2.6 beschriebene Type Substitution und damit auch Ersetzungsgruppen). Damit werden die Fehlermöglichkeiten für „naive“ Programmierer stark reduziert. Das Schema muss dann aber so ausgelegt sein, dass eine eindeutige Verarbeitung ohne Typinformationen möglich ist (dies ist z.B. bei `union` Types wichtig, bei denen die Typinformation u.U. Auskunft darüber geben muss, wie ein bestimmter Wert zu interpretieren ist)
- *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 nachfolgenden Versionen des Schemas verwendet wird, und die Software entsprechend auslegen. Dies bedeutet einen Mehraufwand für die Programmierer, bringt dafür aber höhere Flexibilität, was die Weiterentwicklung des Schemas angeht.

8.1.1 Empfehlungen

- **MUST**: .Werden in einem XML Schema Mechanismen verwendet, die beim Arbeiten mit Instanzen notwendig machen, Typinformationen zu verwenden (es sind dies die in Abschnitt 8.2 beschriebenen Mechanismen), so ist darauf im Schema und in der Dokumentation deutlich hinzuweisen.

8.2 Mechanismen in XML Schema

XML Schema ist typorientiert, d.h. alle Elemente und Attribute haben einen Typ, der aus ihrer Definition ersichtlich ist, auch wenn man diesen Typ einer Instanz alleine (also dem Dokument ohne das Schema) nicht ansieht. XML Schema definiert Typersetzungsverfahren, die in zwei unterschiedlichen Arten verwendet werden können, und die es ermöglichen, die Zuweisung von Typen zu Elementen zu beeinflussen. In beiden Fällen können die Verfahren nur für Elemente angewendet werden.

In beiden Fällen wird in der Instanz ein anderer Typ anstelle des im Schema angegebenen Typ verwendet, wobei der Typ in der Instanz vom im Schema angegebenen Typen abgeleitet sein muss. Im Fall des `xsi:type` Attributes (Abschnitt 8.2.1) wird der ersetzte Typ explizit in der Instanz angegeben, im Fall der Ersetzungsgruppen (Abschnitt 8.2.2) ergibt sich der ersetzte Typ durch das verwendete Element.

- **MUST**: Wird einer der beiden Mechanismen eingesetzt (`xsi:type` oder Ersetzungsgruppen), so darf nur dieser Mechanismus eingesetzt werden. Eine Mischung beider Stile würde ein Schema schwer zu verstehen und den Umgang damit unnötig kompliziert machen. Dies ist keine Einschränkung, da die beiden Mechanismen im Kern ohnehin das gleiche machen, nämlich andere Typen in einem Inhaltsmodell zu ermöglichen, als den Typ des im Inhaltsmodell aufgeführten Elements.

Aus Sicht des Schema Designs öffnet Type Substitution neue Möglichkeiten und ist daher ein interessanter Mechanismus. Andererseits können "naiv" programmierte Anwendungen mit den Mechanismen der Type Substitution Probleme bekommen, da diese eine saubere typbasierte Verarbeitung erfordern, oftmals aber noch rein namensbasiert programmiert wird (mehr Informationen dazu in Abschnitt 8.1). Aus diesem Grund sollte Type Substitution nur verwendet werden, wenn die Konsequenzen für die Verarbeitung klar sind und sichergestellt werden kann, dass die Verarbeitung immer typbasiert erfolgen wird.

8.2.1 Das xsi:type Attribut

In einer Instanz kann ein Element seinen Typ explizit mit dem `xsi:type` Attribut angeben, damit hat das Element an dieser Stelle einen anderen Typ als im Schema angegeben.

8.2.1.1 Problemstellung

Ist ein Element im Schema nicht explizit mit `block="..."` definiert, oder das `blockDefault` Attribut des `xs:schema` Elements gesetzt, so ist Type Substitution mittels `xsi:type` für dieses Element erlaubt (vorausgesetzt, es gibt einen Typ, der vom Typ des Elements abgeleitet ist und nicht in seiner Ableitungsart durch `block` verboten wurde).

8.2.1.2 Empfehlungen

- **SHOULD NOT**: Generell sollten Mechanismen der `xsi:type` Type Substitution vermieden werden, solange nicht gewichtige Gründe dafür sprechen. Diese Mecha-

nismen erschweren die Verarbeitung und das Verständnis des Schemas und der durch das Schema definierten Instanzen.

- **MUST**: Ist im Schema explizit dokumentiert oder sogar durch das Schema-Design erzwungen (Typen als `abstract="true"` definiert), dass `xsi:type` verwendet werden kann oder soll, so ist dies zulässig, muss aber angemessen dokumentiert werden. In der Verarbeitung dieser Instanzen ist dann darauf zu achten, dass diese typbasiert erfolgt, so dass die Typzuweisung nicht über den Typ eines Elements im Schema, sondern über den `xsi:type` aus der Instanz erfolgt.
- **MUST**: Werden Mechanismen der Type Substitution verwendet, so ist dies deutlich und angemessen zu dokumentieren, so dass Benutzer des Schemas deutlich über diesen Aspekt des Schemas informiert werden.

8.2.2 Substitution Groups

In einer Instanz kann ein Element explizit mit dem `substitutionGroup` Attribut angeben, dass es das referenzierte Element in ~~einer~~ dieser Instanz ersetzt~~ten kann~~. Es darf dann in einer Instanz überall dort vorkommen, wo das im `substitutionGroup` Attribut referenzierte Element erlaubt ist. Der Mechanismus kann auch über mehrere Ebenen definiert werden.

8.2.2.1 Problemstellung

Ist ein Element im Schema nicht explizit mit `final="..."` definiert, oder das `finalDefault` Attribut des `xs:schema` Elements gesetzt, so kann dieses Element als Head einer Substitution Group dienen. Die Members der Substitution Group referenzieren den Kopf im `substitutionGroup` Attribut der Elementdefinition.

8.2.2.2 Empfehlungen

- **SHOULD NOT**: Generell sollten Mechanismen der Substitution Groups vermieden werden, solange nicht gewichtige Gründe dafür sprechen. Diese Mechanismen erschweren die Verarbeitung und das Verständnis des Schemas und der durch das Schema definierten Instanzen.
- **MUST**: Ist im Schema explizit dokumentiert oder sogar durch das Schema-Design erzwungen (Elemente, die Substitution Group Heads sind, als `abstract="true"` definiert), dass Substitution Groups verwendet werden können oder sollen, so ist dies zulässig, muss aber angemessen dokumentiert werden. In der Verarbeitung dieser Instanzen ist dann darauf zu achten, dass diese konsequent auf die Substitution Groups Rücksicht nimmt.
- **MUST**: Werden Mechanismen der Substitution Groups verwendet, so ist dies deutlich und angemessen zu dokumentieren, so dass Benutzer des Schemas deutlich über diesen Aspekt des Schemas informiert werden.

9 Abbildung von Datenstrukturen auf XML Markup

XML ist genau genommen nur ~~eine Syntax~~ein Format, mit der strukturierte Daten ausgetauscht werden können. In welcher Weise man die Daten eines Anwendungsgebietes auf XML-Strukturen abbildet, ist dem Entwickler eines XML Schemas überlassen. Dies ist eine sehr wichtige Frage, weil mit den durch den XML Schema Entwickler definierten Daten meistens eine lange Zeit gearbeitet werden muss, sich eine gewisse Sorgfalt und Vorausschau bei der Abbildung von Datenstrukturen auf XML Markup also lange Zeit bezahlt macht. Die folgenden Abschnitte beschreiben verschiedene Aspekte, die bei der Abbildung von Datenstrukturen auf XML Markup wichtig sind, und geben Empfehlungen für besser und weniger gut geeignete Mechanismen, mit diesen Problemen umzugehen.

9.1 Root Elemente

XML Schema hat keine Möglichkeit, das Root Element (im XML Standard wird es als *Document Element* bezeichnet) zu kennzeichnen. Aus diesem Grunde kann einem Schema nicht angesehen werden, welches Element als Root Element gedacht ist, und in den meisten Fällen gibt es verschiedene Möglichkeiten, welche Elemente als Root Elemente dienen könnten. Aus Anwendungssicht kann es ebenfalls erwünscht sein, dass mehrere Elemente eines Schemas als Root Element vorkommen können, jedoch handelt es sich dabei meist um eine kleine Zahl an Elementen.

9.1.1 Empfehlungen

- **MAY**: Schemas dürfen so designt werden, dass mehrere Elemente als Root Elemente vorkommen dürfen. Auf diese Weise kann ein Schema dazu dienen, Instanzen mit verschiedenen Root Elementen zu validieren.
- **SHOULD**: Root Elemente in einem Schema sollten markiert werden. Da es dafür keinen durch XML Schema definierten Mechanismus gibt, muss dies in einem Kommentar geschehen, der die potentiellen Root Elemente kennzeichnet.

9.2 Verarbeitungseinheiten

XML macht keinerlei Aussage darüber, für welche Granularität der Daten die von XML definierten Mechanismen (Elemente und Attribute sind die wichtigsten) verwendet werden. Es bleibt vollständig dem Schema-Designer überlassen, zu entscheiden, in welcher Weise Inhalte strukturiert werden sollen.

Als generelle Richtlinie lässt sich sagen, dass alles, worauf aus einer Applikation heraus zugegriffen werden soll, durch Markup markiert werden sollte. Ist ein Schema in dieser Weise aufgebaut, dann können Applikationen einfache XML-basierte Methoden (wie DOM oder XPath) verwenden, um die Inhalte zu verarbeiten, und müssen nicht selber nach Strukturen suchen. Als Bespiel für das Problem soll hier die Frage dienen, wie sich Personennamen in XML Markup abbilden lassen.

- `<name>Pieter David van Hoogenband, Sr.</name>`: In diesem Fall wird der gesamte Name als Zeichenkette in einem Element angegeben. Welche Teile des Namens Vor- und Nachnamen sind, ist nicht durch Markup gekennzeichnet. Applikationen müssten also eigene Logik programmieren, um dieses festzustellen, und je nach implementierter Logik könnten verschiedene Applikationen dann auch zu verschiedenen Resultaten gelangen.
- `<name given="Pieter David" family="van Hoogenband, Sr."/>`: In diesem Fall ist die Trennung zwischen Vor- und Nachname gegeben, beide Teile des Namens sind klar getrennt und durch verschiedene Attribute repräsentiert.
- `<name><given>Pieter David</given><family>van Hoogenband, Sr.</family></name>`: Strukturell entspricht diese Variante der vorangegangenen, nur dass hier die verschiedenen Teil durch Elemente und nicht durch Attribute repräsentiert werden. Als nachteilig könnte empfunden werden, dass alle Vornamen in einem gemeinsamen Element angeführt werden.
- `<name><given>Pieter</given><given>David</given><family>van Hoogenband, Sr.</family></name>`: In diesem Beispiel sind die Vornamen separat durch Markup markiert. Durch das Schema kann gesteuert werden, ob auch kein Vorname vorkommen darf, und wieviele Vornamen maximal erlaubt sind. Offen ist noch die Frage, ob die gemeinsame Repräsentation der verschiedenen Bestandteile des Nachnamens wünschenswert ist.
- `<name><given>Pieter</given><given>David</given><family link="van" gen="Senior">Hoogenband</family></name>`: Soll der Nachname getrennt werden in den eigentlichen Namen und zusätzliche Bestandteile, so kann dies in der angegebenen Weise erfolgen. Auf diese Weise kann einfach und deutlich unterschieden werden, was der eigentliche Nachname ist, und was weitere Bestandteile des Nachnamens sind.

Welche dieser Modellierungen in einem Schema verwendet wird, hängt von den Anwendungen ab, aber generell wäre im Fall von Personennamen wohl zumindest die Trennung in Vor- und Nachnamen wünschenswert, da diese wahrscheinlich oft von Applikationen gebraucht werden. Ob die weitergehenden Ansätze wünschenswert oder übertrieben sind, lässt sich ohne Kenntnis des Anwendungsbereiches nicht beantworten.

Zu beachten ist, dass eine grössere Granularität mehr Vorgaben über die Strukturen macht und damit eine grössere Genauigkeit erreicht, oftmals jedoch auch eine grössere Einschränkung, da Daten, die nicht in das vordefinierte feingranulare Schema passen, kaum sinnvoll darauf abgebildet werden können. Im Beispiel mit den Personennamen sind die Annahmen z.B., dass alle Namen den Namensstrukturen der westlichen Welt folgen, anders strukturierte Namen, z.B. aus Kulturen mit dynastischen Namen, passen nicht in die feiner strukturierten Schemas, sehr wohl aber als einfache Zeichenkette in den allgemeinen Container eines `name` Elements.

9.2.1 Empfehlung

- **SHOULD**: Die Verarbeitungseinheiten, die bei der Definition eines Schemas gewählt werden, sollten sich daran orientieren, in welcher Weise die Daten aus fachlicher Sicht strukturiert sind. Die Wahl der Verarbeitungseinheiten sollte das Ziel haben, die für die Verarbeitung relevanten Strukturen durch XML Markup zu kennzeichnen, so dass Applikationen soweit möglich keine über XML-Technologien hinausgehenden Methoden brauchen, um die fachlich relevanten Strukturen zu identifizieren und mit ihnen zu arbeiten.

9.3 Elemente oder Attribute?

Eine immer wiederkehrende und nicht abschliessend 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 (weitere Informationen zu diesem Attributtyp finden sich in Abschnitt 9.5).

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 Schemas Identity Constraints lassen sich durch XPath gleichermassen auf Attribute und Elemente anwenden (sie bieten zudem deutliche Vorteile gegenüber dem ID/IDREF Typ, wie in Abschnitt 9.5 beschrieben).

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. XML 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 (eingeschränkt) gesteuert werden.

9.3.1 Empfehlung

- **SHOULD**: Generell sollten Elemente zur Strukturierung verwendet werden, da diese im Bedarfsfall einfacher erweitert werden können, z.B. mit zusätzlichen Attributen, oder durch die Unterteilung in feiner gegliederte Strukturen.
- **MAY**: Attribute sollten mit Vorsicht verwendet werden. Sie unterliegen deutlichen Einschränkungen (nicht wiederholbar, keine Möglichkeit der weiteren Strukturierung)

und bilden daher bei späteren Änderungen einen Schemas u.U. eine Stelle, an der gewünschte Änderungen nicht mehr einfach durchführbar sind.

9.4 Repräsentation leerer Werte

Oftmals besteht in einem Datenmodell die Möglichkeit, dass gewisse Angaben optional sind, also in Instanzen nicht zwingend angegeben werden müssen. Es stellt sich die Frage, wie sich solches optionale Vorkommen am besten in einem XML Schema umsetzen lässt. XML und XML Schema erlauben verschiedene Arten der Realisierung, die im folgenden beschrieben werden.

9.4.1 Problemstellung

Null-Werte können in XML und im speziellen bei der Verwendung von XML Schema durch unterschiedliche Vorgehensweisen dargestellt werden. Um eine allgemein gültige Syntax zu erhalten, sollten jedoch auch in diesem Bereich gewisse Standards angestrebt werden. Technisch gibt es die folgenden Möglichkeiten:

- Nichterscheinen der jeweiligen Komponenten (Elemente oder Attribute)
    - Vorteil: Einfacher Mechanismus
    - Nachteil: Elemente bzw. Attribute müssen im Schema als optional deklariert werden. Dies ist nicht immer erwünscht.
- Leerer Inhalt der jeweiligen Komponenten (Elemente oder Attribute)
    - Vorteil: Einfacher Mechanismus
    - Nachteil: Eine Komponente kann nicht leer sein, wenn auf der Komponente eine Restriktion definiert ist (also der Typ z.B. vorschreibt, dass eine Zeichenkette eine bestimmte Länge haben muss).
- Durch das `xsi:nil="true|false"` Attribut innerhalb der Instanz, in diesem Fall muss das Element (für Attribute funktioniert dieser Mechanismus nicht) im Schema mit `nillable="true"` deklariert werden.
    - Vorteile: (1) der `xsi:nil` Wert ist zusätzlich zum leeren Element definiert, es kann also ein leerer Wert von einem nicht-vorhandenen Wert unterschieden werden, wenn dies gewünscht wird. (2) Erlaubt leeren Inhalt auf Elementen, die mit Restriktionen versehen sind. z.B. einen `xs:string` Type mit minimaler Länge von 2 Zeichen haben.
    - Nachteile: (1) Nur für Elemente verwendbar. (2) Nicht sehr verbreitet eingesetzt, da es starke Restriktionen beinhaltet. (3) Applikationen müssen diesen Mechanismus explizit unterstützen, sonst kann es Probleme bei der Interpretation geben.

9.4.2 Empfehlung

- **SHOULD**: Die Darstellung von Null-Werten sollte mit optionalen Elementen oder Attributen erfolgen, d.h. durch Nichterscheinen der jeweiligen Komponenten, oder durch leeren Inhalt der jeweiligen Komponenten.
- **SHOULD NOT**: Der `xsi:null` Mechanismus sollte vermieden werden.

9.5 Identifikation und Referenzen

Häufig ist es in Datenmodellen notwendig, strukturelle Teile des Datenmodells zu identifizieren und damit referenzierbar zu machen. Der häufigste Weg, diese Aufgabe zu lösen, ist die Vergabe von Namen, mit denen die zu identifizierenden Strukturen benannt werden, und die in den Referenzen verwendet werden können.

Der bekannteste Mechanismus dazu sind die `ID/IDREF` Attribute aus XML DTDs, die es ermöglichen, zum einen Elemente mit einem eindeutigen Namen zu versehen (durch ein als `ID` deklariertes Attribut), und zum anderen Referenzen auf bestehende IDs zu machen (durch ein als `IDREF` deklariertes Attribut). Ein XML Prozessor überprüft dann bei der Validierung, ob die Eindeutigkeit (`ID`) und Existenz (`IDREF`) gegeben sind. Dieser Mechanismus kann auch in XML Schema benutzt werden, wenn die durch [xmlschema2sec] definierten Typen `xs:ID` und `xs:IDREF` verwendet werden.

Der `ID/IDREF` Mechanismus hat zwei grosse Nachteile, die seinen Nutzen zum Teil stark einschränken: Durch die Verwendung der `ID/IDREF` Typen wird nicht nur die Eindeutigkeit bzw. Existenz erzwungen, sondern es wird auch festgelegt, dass die verwendeten Namen XML Namen sein müssen. Das heisst insbesondere, dass keine Zahlen erlaubt sind, was häufig dann störend ist, wenn bereits Namen existieren und diese z.B. aus einer Datenbank als Zahlen zur Verfügung gestellt werden. Der andere Nachteil ist, dass die Konzepte global für ein Dokument gelten, d.h. dass zwei verschiedene als `ID` deklarierte Attribute nicht den gleichen Wert annehmen dürfen, und dass bei `IDREF` Referenzen nicht sichergestellt ist, dass sie auf einen Namen aus dem gewünschten Namensbereich verweisen. Beide Nachteile lassen sich mit den XML Schema Identity Constraints vermeiden.

Aufgrund der bekannten und schwerwiegenden Nachteile des `ID/IDREF` Konzeptes wurde in XML Schema ein neuer Mechanismus eingeführt, die Identity Constraints. Diese sind für den gleichen Anwendungsbereich gedacht wie `ID/IDREF`, vermeiden aber deren Nachteile. Insbesondere kann bei ihnen der Typ der Namen frei gewählt werden, und sie können sich spezifisch auf bestimmte Namen im Schema beziehen (über einen XPath ausgewählt), so dass sie nicht mehr global sind. Weiterhin bestehen bleibt der Nachteil, dass auch die Identity Constraints nur für ein Dokument gelten, also keine Abhängigkeiten zwischen verschiedenen Dokumenten ausdrücken können.

ACHTUNG: Die Identity Constraints sind zwar eine deutliche Weiterentwicklung gegenüber den `ID/IDREF` Konzepten der DTDs, durch die Verwendung von XPath sind sie jedoch an die Namen von Elementen gebunden, nicht an deren Typen! Aus diesem Grund stellt die Verwendung von Identity Constraints einen gewissen Konflikt dar zur in Abschnitt 8 empfohlenen Bevorzugung der typ- vor der namensbasierter Verarbeitung. Insbesondere sind Identity Constraints in Elementdeklarationen enthalten. Werden also die Typen zum bevorzugten

Konstrukt für die Wiederverwendung wie in Abschnitt 7.5 beschrieben, so lässt dies die Identity Constraints ausser Acht. Aus diesem Grund sollte in zur Wiederverwendung bestimmten Typen sehr deutlich dokumentiert werden, wenn Elemente des Typs einen Identity Constraint enthalten sollten.

Die Selektion der für Identity Constraints relevanten Knoten geschieht durch XPaths. Diese müssen die korrekten Namespace Präfixe und Qualifikationen verwenden, ansonsten werden trotz korrekter (lokaler) Namen u.U. keine Knoten durch den XPath selektiert.

9.5.1 Empfehlung

- **SHOULD**: Soll die Eindeutigkeit, Existenz, oder Referenzierung von Namen garantiert werden, so sollten XML Schema Identity Constraints benutzt werden, um diese Constraints zu definieren. Dabei ist darauf zu achten, dass die XPaths so restriktiv wie möglich definiert werden, um keine Konflikte bei späteren Änderungen des Schemas zu riskieren.
- **SHOULD**: Werden Identity Constraints verwendet, so sollten die Strukturen, auf die sie verweisen, eigene Typen für die Namen verwenden. Auf diese Weise können die Typen einfach wiederverwendet werden, wenn die entsprechenden Namen an anderer Stelle des Schemas wiederverwendet werden sollen.
- **SHOULD NOT**: Der DTD Mechanismus der `ID/IDREF` Attribute sollte nicht verwendet werden, da er starken Einschränkungen unterliegt und es zudem nicht erlaubt, die Constraints so genau zu definieren, wie es mit den Identity Constraints möglich ist.
- **MAY**: Sind durch die Applikation Anforderungen gegeben, die sich durch XML Schemas Identity Constraints nicht implementieren lassen (z.B. Eindeutigkeit oder Referenzierung über mehrere Dokumente hinweg), so können diese durch externe Identifikation und externe Constraints definiert werden. Für die durch diese externen Definitionen erfassten Strukturen sollten dann eigene Typen verwendet werden, und im Schema sollte deutlich dokumentiert sein, dass die Werte dieser Typen Constraints unterworfen sind, die ausserhalb des Schemas definiert sind.
- **SHOULD**: Sollen Identity Constraints in zur Wiederverwendung bestimmten Typen vorkommen, so kann dies am besten dadurch dokumentiert werden, dass ein Beispielelement dieses Typs definiert wird, das die Identity Constraints enthält. Im Typ kann dann durch einen Kommentar darauf verwiesen werden, dass bei einer Wiederverwendung des Typs die Identity Constraints aus dem Beispielelement kopiert werden müssen.

9.6 Wertelisten

Wertelisten sind sehr häufig Bestandteil eines Vokabulars, weil für gewisse Teile des Datenmodells nur bestimmte wohldefinierte Werte zugelassen sein sollen. Im Prinzip lassen sich Wertelisten auf zwei Arten definieren:

- Statische Werteliste: In diesem Fall ist die Werteliste als Typ definiert, und der Typ definiert die erlaubten Werte direkt als Aufzählung. Dies passiert üblicherweise durch einen passenden Simple Type und Enumeration Facets, die die erlauben Werte aufzählen.
- Dynamische Werteliste: Sollen die Werte nicht im Schema definiert werden, so wird der Typ als lexikalisch eingeschränkter Typ definiert, der die möglichen Werte in ihrem Wertebereich einschränkt, aber nicht die Werte direkt auflistet. Die aktuellen Werte müssen dann in einer extern geführten Liste definiert werden, und Anwendungen müssen diese Liste kennen und Zugriff auf diese Liste haben, um eine echte Typüberprüfung durchführen zu können.

Einfache und typische Beispiele für diese beiden Ansätze finden sich im XML Schema für XML Schema, das als statischen Werteliste z.B. die folgende Definition enthält:

```
<xs:simpleType name="derivationControl">
  <xs:restriction base="xs:NMTOKEN">
    <xs:enumeration value="substitution"/>
    <xs:enumeration value="extension"/>
    <xs:enumeration value="restriction"/>
    <xs:enumeration value="list"/>
    <xs:enumeration value="union"/>
  </xs:restriction>
</xs:simpleType>
```

In diesem Fall sind die erlauben Werte, die die für die Typableitung in XML Schema erlaubten Werte definieren, direkt im Schema angegeben. Dies ist sinnvoll, da sich diese Werte kaum ändern werden, bzw. im Falle einer Änderung XML Schema ohnehin so grundlegend überarbeitet würde, dass ein komplett neues Schema definiert werden müsste.

Im Gegensatz dazu gibt es im XML Schema für XML Schema auch Beispiele dynamischer Wertelisten, bei denen das Schema auf externe Listen verweist:

```
<xs:simpleType name="language" id="language">
 <xs:restriction base="xs:token">
  <xs:pattern value="[a-zA-Z]{1,8}(-[a-zA-Z0-9]{1,8})*">
   <xs:annotation>
    <xs:documentation source="http://www.ietf.org/rfc/rfc3066.txt">
      pattern specifies the content of section 2.12 of XML 1.0
      and RFC 3066 (Revised version of RFC 1766).
    </xs:documentation>
   </xs:annotation>
  </xs:pattern>
 </xs:restriction>
</xs:simpleType>
```

In diesem Beispiel für den `language` Typ von XML Schema werden die erlaubten Werte nicht direkt im Schema definiert, sondern es wird lediglich ein Pattern definiert, das die Menge der erlaubten Werte lexikalisch einschränkt. Die konkreten Werte für diesen Typen müssen aus einer externen Liste gelesen werden, die durch ein IETF Dokument definiert wird. Der Grund für diese Konstruktion ist hier, dass die Liste der Sprachen zum einen eine in Entwicklung befindliche Liste ist, und dass die Entwickler von XML Schema auf der anderen

Seite keine Aussage darüber machen wollten, welche Sprachen in XML Schema endgültig definiert sind und welche nicht. Formal gesehen lässt das Schema also auch Sprachmarkierungen wie `xx-YY` zu (dies würde erfolgreich validiert werden), und erst die Konsultation von [RFC3066] bzw. der entsprechenden Liste bei der IANA würde zeigen, dass dies keine gültige Sprachmarkierung ist.

9.6.1 Empfehlung

- **SHOULD**: Handelt es sich bei Wertelisten um statische Listen, deren Werte abschliessend bekannt und aller Voraussicht nach stabil sind, so sollten diese Werte im Schema mit einem Simple Type und Enumeration Facets aufgezählt werden. Um eine solche Werteliste besser zu verwalten und bessere Zugriffskontrollen durchführen zu können, ist es im Allgemeinen sinnvoll, sie in ein eigenständiges Schema-Dokument auszulagern (siehe Abschnitt 11.1).
- **SHOULD**: Handelt es sich bei Wertelisten um dynamische Listen, deren Werte nicht abschliessend bekannt sind oder sich verändern können, so sollten sie nur durch eine möglichst exakte lexikalische Einschränkung und den Verweis auf eine externe Liste definiert werden. Auf diese Weise können den Wertelisten neue Werte hinzugefügt werden, ohne dass sich am Schema etwas ändern muss.

9.7 Sprachmarkierung von Inhalten

In Dokumenten mit natürlichsprachlichen Inhalten besteht oftmals die Anforderung, Inhalte mehrsprachig aufführen zu können. Wie genau dies gesteuert wird (z.B. ob genau ein Inhalt in einer anzugebenden Sprache erlaubt ist, oder ob parallele Inhalte in mehreren Sprachen erlaubt sind), ist eine Frage der Applikationsanforderungen.

Unabhängig vom spezifischen Design der Inhalte sollten Sprachmarkierungen jedoch immer mit Hilfe des `xml:lang` Attributes erfolgen, das im XML Standard selber definiert ist. Da dieses Attribut aus dem XML Namespace selber (also dem Namespace, den der XML Namespaces Standard für XML selber reserviert) stammt, muss es per Import (wie in Abschnitt 11.2 besprochen) aus einem eigenen Schema eingebunden werden. Ein XML Schema für das `xml:lang` Attribut ist sehr einfach, das folgende Schema ist ein Beispiel:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           targetNamespace="http://www.w3.org/XML/1998/namespace">
  <xs:attribute name="lang" type="xs:language"/>
</xs:schema>
```

Soll dagegen die Sprachauswahl eingeschränkt werden, so wäre es ebenfalls möglich, wie in Abschnitt 9.6 beschrieben, die Menge der erlaubten Sprachen im `xml:lang` Schema als Aufzählung aufzuführen.

9.7.1 Empfehlung

- **SHOULD**: Sollen Inhalte mit Sprachmarkierungen versehen werden, so sollten diese Markierungen in einem `xml:lang` Attribut angebracht werden, da dieses im XML Standard selber definiert ist und eine allgemeine Konvention für Sprachmarkierungen darstellt. In [eCH-0050] wird ein entsprechendes Schema definiert, das zur Wiederverwendung im Rahmen von eCH bereitgestellt wird.

10 Versionierung von XML Schemas

Schemas ändern sich im Laufe der Zeit und müssen neuen Entwicklungen und Randbedingungen angepasst werden. Dabei stellt sich die Frage, inwieweit bei der Änderung eines Schemas Auswirkungen auf Applikationen zu befürchten sind, die auf älteren Versionen des Schemas basieren. Ist ein Schema offen gestaltet (wie in Abschnitt 13.1 beschrieben), so bleiben Erweiterungen des Schemas ohne Auswirkungen, solange sie die vorgesehenen Erweiterungspunkte benutzen.

10.1 Problemstellung

Bei der Versionierung stellt sich allgemein die Frage, wann bei der Änderung eines Schemas noch von einer neuen Version gesprochen werden kann, und wann die Änderungen an einem Schema dermassen umfassend sind, dass es angebrachter wäre, von einem neuen Schema zu sprechen.

Geht man von einer Versionierung aus, so stellt sich die Frage, welche Änderungen als eine Minor Version gelten sollten, und welche Änderungen als eine Major Version. Als allgemeine Richtlinie können dabei die folgenden Aussagen gelten:

- Ändert sich die Semantik von Konstrukten, so ist in jedem Fall eine neue Major Version zu definieren, das sich dann die Verarbeitung dieser Konstrukte der neuen Semantik anpassen muss. Wenn ein Empfänger eine neue Schema-Version benutzt und damit nicht mehr alte Dokumente validieren könnte, so muss dies ebenfalls zu einer neuen Major Version führen.
- Bei anderen Änderungen, also Änderungen, bei denen alte Instanzen mit dem neuen Schema validiert werden können, ist eine Minor Version ausreichend.

Detaillierter betrachtet, lassen sich zur Frage von Änderungen des Schemas die folgenden Beobachtungen machen:

- Existieren Erweiterungspunkte und werden dort neue Strukturen hinzugefügt, so ist nur eine neue Minor version nötig.
- Werden vorher optionale Teile des Schemas neu als verpflichtend definiert, so ist eine Major Version notwendig, weil Applikationen, die Instanzen dieses Schemas erzeugen, entsprechend angepasst werden müssen.
- Ändert sich die Dokumentation, so ist nur eine Minor Version nötig (jedoch nur dann, wenn die Dokumentation keine neue Semantik beschreibt!).
- Werden Erweiterungen an Stellen eingeführt, an denen keine Erweiterungspunkte vorgesehen waren, so muss eine neue Major Version erzeugt werden.

Wie erwähnt sollte bei massiven Änderungen eines Schemas erwogen werden, ob es sich bei dem Resultat tatsächlich noch um eine neue Version des alten Schemas handelt, oder ob es nicht ein komplett neues Schema ist, das lediglich gewisse inhaltliche Zusammenhänge mit dem Ausgangsschema aufweist.

Bei neuen Versionen von Schemas kann es vorkommen, dass vormals erlaubte Werte, Elemente oder Attribute nicht mehr zugelassen werden sollen. Anstatt diese zu verbieten, kann es sinnvoll sein, diese im neuen Schema nach wie vor zuzulassen, aber deutlich als *deprecated* zu markieren, so dass sie zwar für die Validierung nach wie vor akzeptiert werden, bei der Verarbeitung aber klar ist, dass diese Werte oder Strukturen ignoriert werden sollten. Mit einem solchen Vorgehen lässt sich erreichen, dass Applikationen verschiedener Schema-Versionen besser koexistieren können.

10.2 Empfehlungen

- **SHOULD**: Ändert sich ein Schema in einer Weise, dass bei der Validierung und Interpretation gemäss dem neuen Schema auch Instanzen nach dem alten Schema korrekt verarbeitet werden können, so sollte eine neue Minor Version erzeugt werden.
- **SHOULD**: Ändert sich ein Schema in einer Weise, dass bei der Validierung und Interpretation gemäss dem neuen Schema Instanzen nach dem alten Schema nicht mehr korrekt verarbeitet werden können, so sollte eine neue Major Version erzeugt werden.
- **SHOULD NOT**: Sind vormals erlaubte Werte oder Strukturen in einer neuen Version verboten, so sollten sie nicht in der Schema-Definition verboten werden, sondern auf der Anwendungsebene als deprecated markiert werden, so dass alte Instanzen weiterhin verarbeitet werden können.
- **MUST**: Um die Markierung der Minor Version eines Schemas und dessen Instanzen zu ermöglichen (die Major Version wird über den Namespace Name markiert), wird das `version` Attribut des Schemas verwendet, das nur die Minor Version enthält. Die Minor Version in Instanzen wird durch eine Markierung vorgenommen, die im Schema selbst vorgesehen sein muss, am besten aber durch die Verwendung des in [eCH-0050] vorgesehenen Attributs erfolgen sollte (dieses Attribut ist ebenso durch [eCH-0018] empfohlen).
- **MUST**: Handelt es sich um einen bidirektionalen Datenaustausch, so muss bei einem Austausch die auf beiden Seiten unterstützte Major Version übereinstimmen. Ist dies nicht der Fall, kann die Kommunikation nicht stattfinden.
- **MAY**: Handelt es sich um einen primär unidirektionalen Datenaustausch, so kann bei der Versionierung u.U. darauf Rücksicht genommen werden und es können bei einer Versionierung durch eine Minor Version Änderungen akzeptiert werden, solange diese auf der Empfängerseite nicht zu Kompatibilitätsproblemen führen. In diesem Fall ist eine gute Dokumentation nötig, die das Szenario deutlich beschreibt und damit die Fälle definiert, in denen eine Kommunikation möglich ist.
- **MAY**: Die obigen Empfehlungen sind nur für produktiv eingesetzte und entwickelte Schemas anwendbar. Für die Entwicklungsphase vor dem produktiven Einsatz und für Schemas ausserhalb des produktiven Einsatzes ist ein einfacheres Vorgehen möglich.

11 Strukturierung von Schemas und Schema-Dokumenten

Ein XML Schema ist logisch betrachtet eine Einheit, d.h. alle Komponenten müssen zusammenhängen und es dürfen keine "offenen Referenzen" bestehen. Auf der anderen Seite ist es möglich, Schema-Dokumente zu strukturieren, und Schema-Dokumente sind im Gegensatz zu Schema-Komponenten die konkrete Art, wie man XML Schemas definiert. Diese beiden Aspekte sollen noch einmal hervorgehoben werden:

- Ein *Schema-Dokument* ist ein XML Dokument, dass als Namespace den XML Schema Namespace verwendet (der an den `xs` Präfix gebunden sein sollte), als Document Element ein `xs:schema` Element aus diesen Namespace verwendet, und innerhalb dieses Elementes dann verschiedene Komponenten definiert, z.B. Elemente, Attribute oder Typen.
- *Schema-Komponenten* sind das Datenmodell von XML Schema, es sind die abstrakten Konzepte, die mittels eines Schema-Dokuments definiert werden. XML Schema selber ist anhand dieser Komponenten definiert, und Schema-Dokumente sind eine XML Syntax, mit der man diese Komponenten repräsentieren kann.

Ausgehend von diesen Definitionen ist es wichtig zu erkennen, dass es eine Menge an Aspekten gibt, die zwar im Schema-Dokument erkennbar sind, für das eigentlich wichtige Datenmodell der Schema-Komponenten jedoch irrelevant sind. Für die Autoren und Benutzer von XML Schema können diese Aspekte natürlich trotzdem relevant sein aus Gründen der Organisation, der Zugriffskontrolle oder Übersicht, und aus diesem Grunde soll auf diese Aspekte hier eingegangen werden.

Zu der Frage der Benennung von Schema-Dokumenten und Schemas (der abstrakten Sammlung der Komponenten) lässt sich also folgendes festhalten: Die Benennung eines Schema-Dokuments ist irrelevant und der Name (z.B. die URI), unter dem ein Schema-Dokument abgelegt ist, sollte nicht zur Identifikation des Schemas verwendet werden. Die Benennung des Schemas mit seinem Namespace Name ist die relevante Benennung des Schemas und sollte alleinig zur Identifikation verwendet werden (dies entspricht den in [eCH-0018] festgelegten Richtlinien zur Identifikation des Schemas durch Instanzen). Aus diesem Grund können Schema-Dokumente z.B. aus Effizienzgründen bedenkenlos kopiert werden (z.B. auf lokale Speichermedien), so dass ein effizienterer Zugriff möglich ist. Die Konsistenz solcher Kopien zu garantieren und das Management der Kopien zu organisieren ist dann in der Verantwortung der Anwendung.

11.1 Strukturierung von Schema-Dokumenten

Innerhalb eines Schema-Dokumentes ist die Reihenfolge der globalen Komponenten (also jener, Komponenten, die Kinder des `xs:schema` Elements sind) irrelevant, es ist dort dem Autor überlassen, die Reihenfolge zu wählen, die seinen Bedürfnissen entspricht. Die für XML Schema relevanten Zusammenhänge zwischen diesen Komponenten werden alleine über ihre Namen hergestellt (die Art der möglichen Zusammenhänge ist detailliert in Abschnitt 7 beschrieben), weshalb alle globalen Komponenten einen Namen tragen müssen.

Für grosse Schema-Dokumente kann die Menge der Komponenten rasch dazu führen, dass das Schema sehr gross und damit unübersichtlich wird. In diesem Fall sollte das `xs:include` Element verwendet werden, um die Schema-Dokumente zu modularisieren. Die Verwendung von `xs:include` hat keinerlei Auswirkungen auf die definierten Schema-Komponenten, sie kann jedoch die Organisation, Zugriffskontrolle und Übersicht beim Umgang mit XML Schema deutlich verbessern (zudem können modularisierte Schema-Dokumente, wie in Abschnitt 11.2 beschrieben, in verschiedenen Szenarien wiederverwendet werden).

Das Element `xs:include` in XML Schema funktioniert auf eine sehr einfache Weise, es ist fast genau so, als würde der textuelle Inhalt des referenzierten Schema-Dokumentes an der Stelle des `xs:include` Elementes eingebunden werden (natürlich ohne das umgebende `xs:schema` Element des eingebundenen Schema-Dokumentes zu berücksichtigen). Das stimmt nicht ganz exakt, weil Namespace-Deklarationen vorher aufgelöst werden und es deshalb eine gewisse Interpretation vor dem Einbinden gibt, aber dies sind lediglich Details. `xs:include` funktioniert rekursiv, d.h. das das Einbinden beliebig tief geschachtelt werden kann, und alle diese Abhängigkeiten vor der Interpretation des Schema aufgelöst werden.

XML Schema bietet mit `xs:redefine` einen über `xs:include` hinausgehenden Mechanismus an, weil mit `xs:redefine` referenzierte Schema-Dokumente zwar auch wie bei `xs:include` Bestandteil des einschliessenden Dokuments werden, es aber darüber hinaus erlaubt ist, gezielt Komponenten des eingeschlossenen Schemas zu überschreiben. Auf diese Weise können Schema-Dokumente wiederverwendet, aber an bestimmten Stellen gezielt verändert werden.

Sowohl `xs:include` als auch `xs:redefine` haben deutliche Einschränkungen hinsichtlich des verwendeten Namespaces: Das eingebundene Schema-Dokument muss den gleichen `targetNamespace` haben wie das einbindende Schema-Dokument (allgemeines zur Verwendung von Namespaces in XML Schema findet sich in Abschnitt 5).

Darüber hinaus ist es gestattet, dass ein eingebundenes Schema keinen `targetNamespace` definiert, in diesem Fall nehmen die Komponenten des eingebundenen Schemas den Namespace des einbindenden Schemas an. Weil auf diese Weise Komponenten aus einem Schema-Dokument ohne `targetNamespace` unter verschiedenen Namespace Namen auftauchen können, wird diese Art der Verwendung auch als *Chamäleon Schemas* bezeichnet (die eingebundenen Komponenten nehmen den Namespace der Umgebung an, in der sie verwendet werden).

11.1.1 Problemstellung

Schema-Definitionen können auf sehr viele verschiedene Arten auf Schema-Dokumente aufgeteilt werden. Der Erzeuger eines Schemas sollte zum einen seine Bedürfnisse bei der Schemadefinition im Auge haben, als auch die der Schema-Benutzer. Wie generell bei der Modularisierung ist der ideale Kompromiss zwischen sehr grossen und schlecht handhabbaren monolithischen Definitionen und einer unübersichtlichen Vielzahl an Kleinstmodulen zu suchen.

Chamäleon Schemas stellen ein besonderes Problem dar, da sie zum einen attraktiv sind (einfache Wiederverwendung von Komponenten in verschiedenen Kontexten), zum anderen aber auch Risikopotential bergen. Die beiden grössten potentiellen Probleme bei der Verwendung von Chamäleon Schemas sind Name Clashes, weil wiederverwendete Komponenten nicht sauber über den Namespace Namen getrennt sind, und zum anderen die fehlende Möglichkeit, bestehende Gemeinsamkeiten von Komponenten zu erkennen, da sie jeweils unter einem anderen Namespace Namen auftauchen und ihre gemeinsame Herkunft nicht mehr erkennbar ist.

11.1.2 Empfehlungen

- **SHOULD**: Aus Gründen der Modularisierung sollten Schema-Dokumente, die über eine gewisse Grösse hinausgehen, unter Verwendung von `xs:include` strukturiert werden. Dies erlaubt übersichtlichere Schemas und zudem die Wiederverwendung von Teilen des Schemas (z.B. Wertelisten, wie in Abschnitt 9.6 beschrieben).
- **SHOULD NOT**: Der `xs:redefine` Mechanismus von XML Schema sollte vermieden werden, da er zu komplizierten Abhängigkeiten zwischen verschiedenen Schema-Dokumenten führt und vor allem bei späteren Änderungen von Schema-Dokumenten zu schwierig auffindbaren Fehler führen kann.
- **SHOULD NOT**: Chamäleon Schemas (d.h. Schema-Dokumente ohne `targetNamespace`, die daher den Namespace des Schema-Dokumenten annehmen, in das sie eingebunden werden), sollten vermieden werden, da sie diverse Fehlermöglichkeiten mit sich bringen und bestehende Gemeinsamkeiten nicht offenlegen.
- **MAY**: Arbeiten verschiedene Teile eines Teams an der Entwicklung oder Weiterentwicklung eines Schemas, so kann dies ebenfalls ein Grund für die Modularisierung eines Schemas sein. In diesem Fall wird sich die Modularisierung an der inhaltlichen Zuteilung der verschiedenen Teile des Schemas an verschiedene Teile des Teams orientieren.

11.2 Strukturierung von Schemas

Die in Abschnitt 11.1 erläuterte Strukturierung von Schema-Dokumenten beschreibt, wie die verschiedenen Schema-Dokumente, die ein Schema definieren, strukturiert werden sollten. Über die dort beschriebenen Mechanismen hinaus bietet XML Schema noch den Mechanismus des `xs:import` an, der im Gegensatz zu `xs:include` nicht Schema-Dokumente referenziert, sondern andere Schemas. Mit `xs:import` werden also bestehende Schemas in einem Schema referenziert, und die Trennung findet über den Namespace statt, d.h. das importierte Schema muss einen anderen Namespace haben als das importierende Schema, und auf diese Weise können Komponenten der beiden Schemas voneinander unterschieden werden.

Bei der Definition von Schema-Dokumenten sollte bedacht werden, dass sie sowohl per `xs:include` als auch per `xs:import` verwendet werden können. So könnte eine in einem Projekt definierte Werteliste (siehe Abschnitt 9.6), die dort in einem eigenen Schema-

Dokument definiert und per `xs:include` verwendet wurde, zu einem späteren Zeitpunkt von einem anderen Schema als eigenes Schema mittels `xs:import` importiert werden, so dass der Typ, der die Werteliste definiert, in diesem neuen Schema als Typ des importierten Schemas wiederverwendbar ist.

11.2.1 Problemstellung

Bei der Definition von Schema-Dokumenten und Schemas ist nicht vollständig absehbar, in welcher Form ein Schema oder Teile eines Schemas wiederverwendet werden könnten. Es ist daher darauf zu achten (wie bereits in Abschnitt 11.1.1 beschrieben), dass bei der Definition von Schemas und Schema-Dokumenten modular vorgegangen wird, so dass die Möglichkeiten späterer Wiederverwendung möglichst offengehalten werden.

11.2.2 Empfehlungen

- **SHOULD**: Soweit möglich, sollten in einem neuen Schema existierende Lösungen für Teilaspekte des Schemas durch das Importieren der betreffenden Schemas wiederverwendet werden. Dies erleichtert die Entwicklung von Schemas, und ist auch für die Anwender eines Schemas eine Erleichterung, weil diese bereits bekannte Schemas benutzen und auf diese Weise bestehendes Know-how und bestehenden Code wiederverwenden können.
- **SHOULD**: Handelt es sich bei der Entwicklung von Schemas um unterschiedliche Teams oder Projekte, so sollten diese nicht ein gemeinsames Schema entwickeln, sondern verschiedene Schemas. Auf diese Weise ist die Entwicklung besser entkoppelt, und mögliche Probleme durch verschieden schnellen Fortschritt in den Teams oder Projekten werden umgangen.

12 Implementierung referentieller Beziehungen

In Datenmodellen treten oft Abhängigkeiten auf, die in Implementierungen des Datenmodells repräsentiert werden müssen. Aufgrund seiner hierarchischen Natur kennt XML zwei verschiedene Arten, wie Abhängigkeiten implementiert werden können:

- *Hierarchien*: Diese Art von Beziehungen ist in XML sehr einfach und natürlich zu verwenden, da XML Dokumente immer Bäume sind, hierarchische Strukturen also Teil des Grundmodells von XML sind. Da es sich aber um Bäume handelt, müssen auch die damit einhergehenden Einschränkungen in Betracht gezogen werden, nämlich die Einschränkung, dass nur 1:n Beziehungen auf diese Weise abgebildet werden können. Sollen n:m Beziehungen abgebildet werden, so kann dies nicht ohne weiteres auf Hierarchien abgebildet werden. Zudem ergibt sich die Einschränkung, dass nur eine solche 1:n Beziehung abgebildet werden kann, da ein Element nur Inhalt eines anderen sein kann.
- *Referenzen*: Um die oben erwähnten Einschränkungen zu umgehen, muss in XML eine Implementierung gewählt werden, die nicht Hierarchien benutzt, sondern Referenzen. In diesem Fall werden die zueinander in Beziehung zu setzenden Strukturen jeweils identifiziert, und die Beziehung wird darüber hergestellt, dass die Referenzen zueinander in Beziehung gesetzt werden. Dies ist sehr ähnlich dem Ansatz der Schlüssel und Fremdschlüssel in Datenbanken. Um Aussagen über die erlaubten Beziehungen machen zu können, gibt es in XML ergänzende Mechanismen, in DTDs sind dies die `ID/IDREF` Attributtypen, in XML Schema sind es die in Abschnitt 9.5 beschriebenen Identity Constraints.

Aus diesen Alternativen lässt sich erkennen, dass bei komplexen referentiellen Beziehungen (n:m oder mehr als eine 1:n Beziehung) auf Referenzen zurückgegriffen werden muss, während für einfache 1:n Beziehungen auch eine Abbildung auf Hierarchien möglich ist.

XML selber und die Features der verschiedenen XML Schemasprachen konzentrieren sich eher darauf, die erlaubten hierarchischen Benziehungen zu definieren. Es gibt zwar Mechanismen zur Definition von Randbedingungen für Referenzen (die oben erwähnten `ID/IDREF` Attributtypen der DTDs und die in Abschnitt 9.5 beschriebenen Identity Constraints von XML Schema), jedoch bieten diese keine besonders weitgehenden Möglichkeiten, Randbedingungen für Referenzen zu definieren. Werden Referenzen verwendet, so können viele aus dem fachlichen Modell resultierenden Randbedingungen also nicht im XML Schema definiert werden, sondern nur dokumentiert oder mit einem der in Abschnitt 14 beschriebenen ergänzenden Mechanismen definiert.

Bei der Verwendung hierarchischer oder referentieller Beziehungen in XML Schema sollten die folgenden Überlegungen in Betracht gezogen werden:

- *Pro Hierarchie, Contra Referenzen:* Sollen auf Dokumenten vorwiegend XML-Technologien eingesetzt werden (z.B. XPath bzw. die darauf basierenden XSLT und XQuery), so können diese wesentlich besser auf hierarchisch strukturierten Daten arbeiten, der Umgang mit flachen und konsequent nicht-hierarchischen Dokumenten ist mit diesen Technologien nicht optimal.

- *Pro Referenzen, Contra Hierarchie:* Hierarchische Beziehungen sind wenig robust gegenüber strukturellen Änderungen in Instanzen, geringfügige Änderungen in den Daten können u.U. grössere Änderungen an den Hierarchien zur Folge haben, da u.U. ganze "Unterbäume" innerhalb des Dokuments bewegt werden müssen.

Wichtig ist also neben prinzipiellen Abwägungen zwischen hierarchischen und referentiellen Strukturen auch die Frage, wie die Anwendungen aufgebaut sind, die mit den Strukturen arbeiten. Es sollte jedoch immer bedacht werden, dass die Lebensdauer von Schemas und der damit beschriebenen Daten die der Anwendungen oft übersteigt und zudem später häufig neue Anwendungen hinzukommen, die nicht unbedingt die gleichen Randbedingungen haben werden wie die gerade aktuellen Anwendungen. Aus diesem Grund ist eine Modellierung, die zur sehr von der Sicht der gerade aktuellen Anwendungen geprägt ist, zumindest mit Vorsicht vorzunehmen.

12.1 Empfehlungen

- **MAY**: Ist es aus Sicht von Implementierungsüberlegungen angebracht, auf hierarchische Darstellungen zu verzichten (weil beispielsweise Datenproduktion und -konsum immer auf der Basis eines RDBMS stattfinden), so ist es erlaubt, auf die hierarchische Darstellung von Daten zu verzichten. Dies sollte jedoch nur mit Vorsicht und Bedacht geschehen, weil sich die durch die Implementierungsüberlegungen gegebenen Randbedingungen ändern können (z.B. bei der Umstellung auf XML-Datenbanken), und das Schema dann nicht mehr zur geänderten Umgebung passt.
- **SHOULD**: Werden Referenzen verwendet, so sollten die Randbedingungen der Referenzen so gut wie möglich mit Hilfe von Identity Constraints beschrieben werden (mit den in Abschnitt 9.5 beschrieben Vorbehalten), weitergehende Randbedingungen sollten dokumentiert werden.
- **SHOULD**: Sind Schemas hauptsächlich auf Endbenutzer ausgerichtet, z.B. im Fall von XML-Dokumenten, die von Menschen verwendet (betrachtet oder editiert) werden, z.B. Konfigurationsdateien oder Web-orientierte Dokumente, so sollte die Modellierung an den entsprechenden Stellen eher hierarchisch sein. Hierarchische Strukturen sind für Menschen einfacher zu verstehen als durch viele Referenzen miteinander vernetzte Strukturen.
- **SHOULD**: Sind Schemas hauptsächlich für die maschinelle Weiterverarbeitung bestimmt, so kann das Modell eher flacher definiert werden als für den hauptsächlich auf Endbenutzer ausgerichteten Fall. Allerdings sollte an den Stellen, wo es eine fachlich begründete, dem Basismodell innewohnende hierarchische Struktur gibt, auch eine solche Struktur im XML definiert werden.

13 Offene und erweiterbare Schemas

Um Schemas sinnvoll erweitern zu können, müssen zwei an sich getrennte Aspekte betrachtet werden: Zum einen muss das Design des Ausgangsschemas schon so gestaltet sein, dass es sich überhaupt erweitern lässt. Die dafür wichtigen Aspekte wurden bereits 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 bisher nicht erwähnte 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 gemeinsam betrachtet werden müssen.

13.1 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. 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 Ver-

sion 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 Ideal 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.

13.1.1 Empfehlungen

- **MAY**: Ist ein offenes Schema erwünscht, so dass in Instanzen des Schemas Inhalte erscheinen dürfen, die nicht detailliert im Schema definiert sind, so kann die über Wildcards für Elemente und/oder Attribute erreicht werden. Diese Wildcards können einerseits dazu dienen, Offenheit gegenüber nicht bekannten Inhalten zu bewahren, andererseits aber auch dazu, Offenheit gegenüber vorhergesehenen zukünftigen Erweiterungen zu erreichen.

13.2 Erweiterbare XML Schemas

Nach diesen eher theoretischen Ausführungen soll nun untersucht werden, wie sich die Erweiterbarkeit in einem XML Schema unterstützen lässt. Dafür gehen wir davon aus, dass z.B. 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 in Abschnitt 11 näher betrachtet. Hier soll in einem einfachen Ansatz angenommen werden, 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 hier das Wichtigste, die Namespaces erhalten. Das 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 sollen Erweiterungen zugelassen werden, das Schema soll also offen gestaltet werden. 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 die erlaubten Namespaces der Element Wildcard mit `##other` auf andere Namespaces als den `targetNamespace` eingeschränkt wurden. Wären Erweiterungen im `targetNamespace` des Schemas zugelassen worden, und anschliessend hätte 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 Erweiterbarkeit 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.

13.2.1 Richtlinien für erweiterbare Schemas

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 sich die folgenden Ausführungen beziehen. Die Erweiterung durch Type Substitution ist damit reservieren 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 wichtig, denn ein Ziel ist, 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 als Instanz 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 ratsam, in einem Umfeld mit erweiterbaren und offenen Schemas ein gut definiertes Verarbeitungsmodell zu haben, das 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.

13.2.2 Empfehlungen

Welche der hier beschriebenen Mechanismen in welcher Weise eingesetzt werden, hängt vom Modell ab, das mit XML Schema umgesetzt werden soll, sowie von der Planung, wie sich dieses Modell und das damit zusammenhängende Schema entwickeln kann oder wird. Aus diesem Grund kann für diesen Bereich keine Empfehlung abgegeben werden.

Leider gibt es für XML Schema noch keine etablierte Methodik zur Schema-Evolution oder für wohldefinierte Wege, wie sich ein Schema ändernden Bedürfnissen anpassen lässt. Aus diesem Grund ist eine Planung der eigenen Strategie unerlässlich, um im Verlauf der Versionierung eines Schemas eine kohärente Strategie zu haben, wie mit neuen Anforderungen umgegangen wird.

- **SHOULD**: Bei der Definition eines Vokabulars sollten Möglichkeiten der Offenheit, Erweiterung und Versionierung bereits berücksichtigt werden. Die beabsichtigte Strategie, wie mit diesen Aspekten umgegangen werden soll, sollte Teil der Dokumentation des Schemas sein, nicht nur Teil des Design des Schemas. Auf diese Weise ist es für Anwender des Schemas einfacher, das Design des Schemas zu verstehen, das Schema weiterzuverwenden, und Applikationen zu schreiben, die Instanzen des Schemas robust verarbeiten können.

14 Ergänzende Mechanismen

XML Schema ist für die absehbare Zukunft die wichtigste Schemasprache für XML. Natürlich hat aber auch XML Schema Schwächen und kann einige teilweise recht einfache Anforderungen an ein Schema nicht implementieren, z.B. die folgenden Fälle:

- *Alternativen aus Element und Attribut*: Oftmals wäre es nützlich, wenn sich ausdrücken liesse, dass ein bestimmtes Element benutzt werden darf oder aber ein Attribut, nicht aber beides. Da XML Schema die Inhaltsmodelle von Elementen strikt trennt von den Attributlisten, ist eine solche Wechselbeziehung nicht ausdrückbar mit XML Schema.
- *Beziehungen zwischen Attributen*: Während für Elemente eine recht ausdruckstarke Sprache zur Verfügung steht, um ihr Vorkommen zu definieren, werden Attributen in Listen aufgeführt, so dass keine Beziehungen zwischen Attributen definiert werden können, z.B. dass zwei optionale Attribute immer gemeinsam vorkommen müssen.
- *Beziehungen zwischen Werten*: Die Simple Types erlauben zwar die Definition des Wertebereichs von Elementen oder Attributen, aber es können keine Beziehungen zwischen Werten definiert werden. So kann z.B. nicht definiert werden, dass ein Attribut immer einen Wert haben soll der grösser ist als der eines anderen.

Finden sich viele solcher oder ähnlicher Bedingungen im fachlichen Modell, die sich nicht auf XML Schema abbilden lassen, so sollten sie dokumentiert werden. Darüber hinaus ist es möglich, andere Schemasprachen wie RELAX NG [ISO19757-2] oder Schematron [ISO19757-3] einzusetzen, die XML Schema ergänzen. Diese ergänzenden Schemasprachen sind ausserhalb des Anwendungsbereiches des vorliegenden Dokumentes (das sich auf XML Schema beschränkt), jedoch sollte auf die Möglichkeit ihrer Verwendung Rücksicht genommen werden, da sich dadurch u.U. eine bessere Implementierung des fachlichen Modells in XML erreichen lässt.

15 Sicherheitsüberlegungen

Keine.

16 Haftungsausschluss/Hinweise auf Rechte Dritter

eCH-Standards, welche der Verein **eCH** dem Benutzer zur unentgeltlichen Nutzung zur Verfügung stellt, oder welche **eCH** referenziert, haben nur den Status von Empfehlungen. Der Verein **eCH** haftet in keinem Fall für Entscheidungen oder Massnahmen, welche der Benutzer auf Grund dieser Dokumente trifft und / oder ergreift. Der Benutzer ist verpflichtet, die Dokumente vor deren Nutzung selbst zu überprüfen und sich gegebenenfalls beraten zu lassen. **eCH**-Standards können und sollen die technische, organisatorische oder juristische Beratung im konkreten Einzelfall nicht ersetzen.

In **eCH**-Standards referenzierte Dokumente, Verfahren, Methoden, Produkte und Standards sind unter Umständen markenrechtlich, urheberrechtlich oder patentrechtlich geschützt. Es liegt in der ausschliesslichen Verantwortlichkeit des Benutzers, sich die allenfalls erforderlichen Rechte bei den jeweils berechtigten Personen und/oder Organisationen zu beschaffen.

Obwohl der Verein **eCH** all seine Sorgfalt darauf verwendet, die **eCH**-Standards sorgfältig auszuarbeiten, kann keine Zusicherung oder Garantie auf Aktualität, Vollständigkeit, Richtigkeit bzw. Fehlerfreiheit der zur Verfügung gestellten Informationen und Dokumente gegeben werden. Der Inhalt von **eCH**-Standards kann jederzeit und ohne Ankündigung geändert werden.

Jede Haftung für Schäden, welche dem Benutzer aus dem Gebrauch der **eCH**-Standards entstehen ist, soweit gesetzlich zulässig, wegbedungen.

17 Urheberrechte

Wer **eCH**-Standards erarbeitet, behält das geistige Eigentum an diesen. Allerdings verpflichtet sich der Erarbeitende sein betreffendes geistiges Eigentum oder seine Rechte an geistigem Eigentum anderer, sofern möglich, den jeweiligen Fachgruppen und dem Verein **eCH** kostenlos zur uneingeschränkten Nutzung und Weiterentwicklung im Rahmen des Vereinszweckes zur Verfügung zu stellen.

Die von den Fachgruppen erarbeiteten Standards können unter Nennung der jeweiligen Urheber von **eCH** unentgeltlich und uneingeschränkt genutzt, weiterverbreitet und weiterentwickelt werden.

eCH-Standards sind vollständig dokumentiert und frei von lizenz- und/oder patentrechtlichen Einschränkungen. Die dazugehörige Dokumentation kann unentgeltlich bezogen werden.

Diese Bestimmungen gelten ausschliesslich für die von **eCH** erarbeiteten Standards, nicht jedoch für Standards oder Produkte Dritter, auf welche in den **eCH**-Standards Bezug genommen wird. Die Standards enthalten die entsprechenden Hinweise auf die Rechte Dritter.

Anhang A – Referenzen & Bibliographie

[eCH-0018] Erik Wilde, Hanspeter Salvisberg, Alexander Pina, *XML Best Practices*, eCH, Berne, Switzerland, eCH-0018, August 2005

[eCH-0033] Erik Wilde, *Beschreibung von XML Namespaces*, eCH, Berne, Switzerland, eCH-0033, 2006.

[eCH-0050] Erik Wilde, *Hilfskomponenten zur Konstruktion von XML Schemas*, eCH, Berne, Switzerland, eCH-0050, 2006.

[ISO19757-2] International Organization for Standardization, *Information Technology — Document Schema Definition Languages (DSDL) — Part 2: Grammar-based Validation — RELAX NG*, ISO/IEC 19757-2, November 2003.

[ISO19757-3] International Organization for Standardization, *Information Technology — Document Schema Definition Languages (DSDL) — Part 3: Rule-based Validation — Schematron*, ISO/IEC 19757-3, June 2005.

[RFC2119] Scott O. Bradner, *Key Words for use in RFCs to Indicate Requirement Levels*, Internet RFC 2119, March 1997. http://www.ietf.org/rfc/rfc2119.txt

[RFC3066] Harald Tveit Alvestrand, *Tags for the Identification of Languages*, Internet RFC 3066, January 2001. http://www.ietf.org/rfc/rfc3066.txt

[RFC3986] Tim Berners-Lee, Roy Fielding, Larry Masinter, *Uniform Resource Identifier (URI): Generic Syntax*, Internet RFC 3986, January 2005. http://www.ietf.org/rfc/rfc3986.txt

[xml10third] Tim Bray, Jean Paoli, C. Michael Sperberg-McQueen, Eve Maler, François Yergeau, *Extensible Markup Language (XML) 1.0 (Third Edition)*, World Wide Web Consortium, Recommendation REC-xml-20040204, February 2004. http://www.w3.org/TR/2004/REC-xml-20040204

[xml11] Tim Bray, Jean Paoli, C. Michael Sperberg-McQueen, Eve Maler, François Yergeau, John Cowan, *Extensible Markup Language (XML) 1.1*, World Wide Web Consortium, Recommendation REC-xml11-20040204, February 2004. http://www.w3.org/TR/2004/REC-xml11-20040204

[xml11schema10] Henry S. Thompson, *Processing XML 1.1 Documents with XML Schema 1.0 Processors*, World Wide Web Consortium, Note NOTE-xml11schema10-20050511, May 2005. http://www.w3.org/TR/2005/NOTE-xml11schema10-20050511

[xmlns] Tim Bray, Dave Hollander, Andrew Layman, *Namespaces in XML*, World Wide Web Consortium, Recommendation REC-xml-names-19990114, January 1999. http://www.w3.org/TR/1999/REC-xml-names-

19990114

[xmlns11] Tim Bray, Dave Hollander, Andrew Layman, Richard Tobin, *Namespaces in XML 1.1*, World Wide Web Consortium, Recommendation REC-xml-names11-20040204, February 2004. http://www.w3.org/TR/2004/REC-xml-names11-20040204

[xmlschema1sec] Henry S. Thompson, David Beech, Murray Maloney, Noah Mendelsohn, *XML Schema Part 1: Structures Second Edition*, World Wide Web Consortium, Recommendation REC-xmlschema-1-20041028, October 2004. http://www.w3.org/TR/2004/REC-xmlschema-1-20041028

[xmlschema2sec] Paul V. Biron, Ashok Malhotra, *XML Schema Part 2: Datatypes Second Edition*, World Wide Web Consortium, Recommendation REC-xmlschema-2-20041028, October 2004. http://www.w3.org/TR/2004/REC-xmlschema-2-20041028

Anhang B – Mitarbeit & Überprüfung

Hans-Ulrich Bucher, Avataris AG

Remo Dick, ISC EJPD

Claude Eisenhut, Eisenhut Informatik

Urs Gähler, VRSG

Jürg Hotz, Kanton Thurgau

Adrian K. Keller, SAG Software Systems AG

Willy Müller, ISB

Hubert Münst, Data Factory

Patrick Ostertag, Etat de Fribourg

Alexander Pina, Unisys (Schweiz) AG

Fabian Probst, Fachhochschule Solothurn Nordwestschweiz

Hanspeter Salvisberg, Unisys (Schweiz) AG

Verena Sieber, T-Systems

Hans Ulrich Wiedmer, KOGIS - LT

Hansruedi Vock, BIT

Erik Wilde, ETH Zürich

Anhang C – Abkürzungen & Glossar

Ein kommentiertes, mit weiterführenden Links versehenes und wesentlich ausführlicheres Abkürzungsverzeichnis und Glossar findet sich auf dem Web unter http://dret.net/glossary/.

DOM	Document Object Model
DSDL	Document Schema Definition Languages
DTD	Document Type Definition
IANA	Internet Assigned Numbers Authority
IETF	Internet Engineering Task Force
RDBMS	Relational Database Management System
RFC	Request for Comments
URI	Universal Resource Identifier
XML	Extensible Markup Language
XSD	XML Schema Definition Language, häufig verwendete (aber nicht offizielle) Abkürzung für XML Schema

Versionen

Version	*Datum*	*Name*	*Bemerkungen (geändert, geprüft, genehmigt) *(geplant)*
0.10	2005-04-05	Erik Wilde	erster Entwurf auf Grundlage eCH-Sitzung 25.11.04
0.20	2005-05-30	Erik Wilde	Einarbeitung der Kommentare vom 8.4.05 (eCH-Sitzung)
0.30	2005-09-07	Erik Wilde	Einarbeitung der Kommentare vom 3.6.05 (eCH-Sitzung)
0.40	2005-11-16	Erik Wilde	Einarbeitung der Kommentare vom 16.9.05 (eCH-Sitzung)
0.90	2006-01-09	Erik Wilde	Einarbeitung der Kommentare vom 2.12.05 (eCH-Sitzung)
0.91	2006-02-09	Erik Wilde	Einarbeitung der Kommentare vom 2.2.06 (eCH-Sitzung)
0.92	2006-02-16	Willy Müller	Formatierungen
1.0	2006-11-27	Erik Wilde	Definitive Version