namics Developer Days – 21./22.6.2002

XML Schema: Validieren statt Programmieren

Erik Wilde
ETH Zürich
http://dret.net/netdret/

Übersicht

- Motivation
- XML Schema from 10.000 ft
- Alternativen und Ergänzungen zu XML Schema
- Probleme und Stand der Technik

Motivation

- Programmieren und Validieren
  - B2B Applikationen brauchen valide Daten
    - viele Bedingungen sind nicht im Schema
  - mehr Validieren heisst weniger Programmieren
    - deklarative vs. prozedurale Vorgehensweise
- bessere Verträge in B2B Szenarien
  - *Information Producer* vs. *Information Consumer*
- XML und Schemas
  - DTDs und ihre Nachteile
  - XML Schema als Weiterentwicklung von DTDs

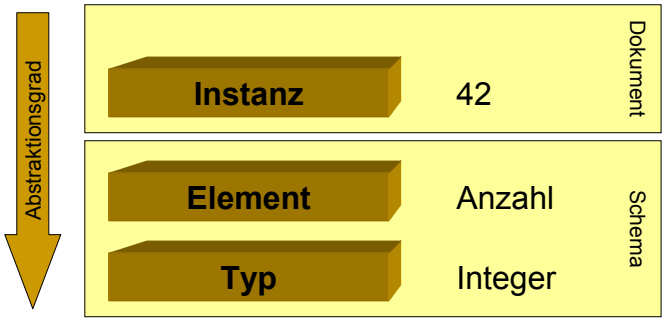
Was ist schlecht an DTDs?

- eine ganze Menge...
- keine Datentypen
  - Elemente haben `#PCDATA`
  - Attribute haben einige wenige Typen (`CDATA`, `ID`, ...)
- keine Unterstützung für Modellierungsmethoden
  - Element- und Attributtypen sind isoliert
  - moderne Modellierung benutzt Typhierarchien
- keine XML Dokument Syntax
  - braucht spezielle Parser, keine XML Tools
- diverse wenig benutzte obskure Features

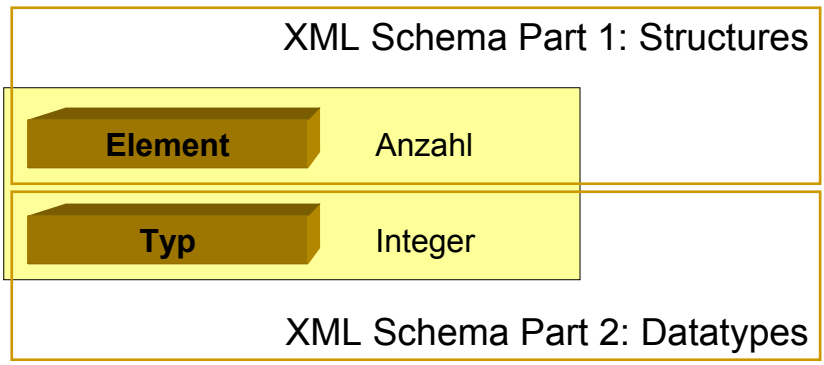
XML Schema from 10.000 ft

- XML Schema Simple Types
- XML Schema Complex Types
- Type Derivation
- Identity Constraints
  - Konsistenzbedingungen
- Advanced Features
  - *Reusable Groups* und *Substitution Groups*

Instanz – Element – Typ



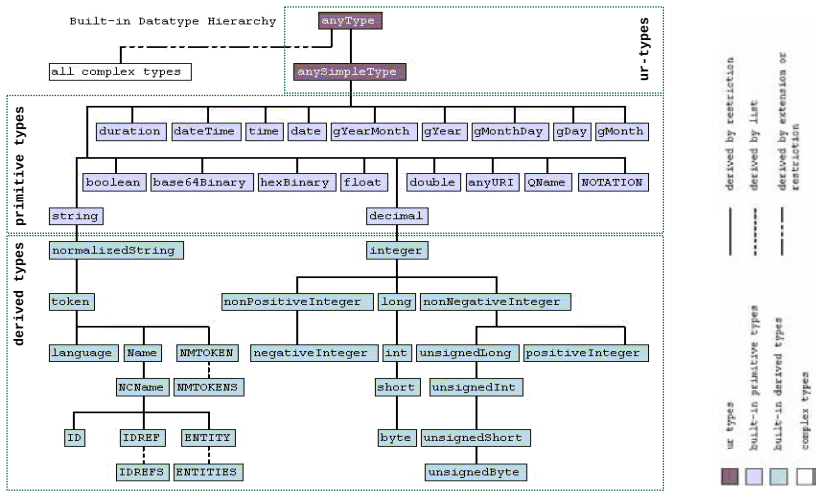
XML Schema Standards



Was sind "Simple Types"?

- die Grundbausteine von
  - XML Schema
  - und damit auch XML Dokumenten
- Inhalt von Elementen oder Attributen
  - Elemente: `<isbn>0130655678</isbn>`
  - Attribute: `<buch isbn="0130655678">`
- drei Varianten von Simple Types
  - Atomic Types (kleinste Einheit, z.B. Zahlen)
  - List Types (mit Space getrennt, z.B. `"3 5 7"`)
  - Union Types (Vereinigung anderer Simple Types)

Hierarchie der Simple Types



Schema der Built-In Types

```
<xs:simpleType name="integer">
    <xs:restriction base="xs:decimal">
        <xs:fractionDigits value="0" fixed="true"/>
    </xs:restriction>
</xs:simpleType>

<xs:simpleType name="nonNegativeInteger">
    <xs:restriction base="xs:integer">
        <xs:minInclusive value="0"/>
    </xs:restriction>
</xs:simpleType>

<xs:simpleType name="positiveInteger">
    <xs:restriction base="xs:nonNegativeInteger">
        <xs:minInclusive value="1"/>
    </xs:restriction>
</xs:simpleType>
```

Named oder Anonymous?

- Simple Types kommen in zwei Varianten
  - Named Simple Types
  - Anonymous Simple Types
- eine Frage der Wiederverwendung
  - Named Types haben einen Typ-Namen
    - können (und sollten) wiederverwendet werden
    - werden immer global deklariert (im gesamten Schema)
  - Anonymous Types werden ohne Name verwendet
    - definiert an der Stelle wo sie verwendet werden
    - damit keine Wiederverwendung möglich
- eine wichtige Modellierungsfrage!

Named Simple Types

- deklariert durch `simpleType`
  - ```
    <xsd:simpleType name="DressSizeType">
        <xsd:restriction base="xsd:integer">
            <xsd:minInclusive value="2"/>
            <xsd:maxInclusive value="18"/>
        </xsd:restriction>
    </xsd:simpleType>
    ```
  - Inhalt `restriction`, `list` oder `union`
- definiert, ob weiter abgeleitet werden darf
  - `<xsd:simpleType name="DressSizeType" final="#all">`
  - kann spezifisch die Art von Ableitung verbieten
    - `extension`, `restriction`, `list`, `union` (und `#all`)
    ```

Anonymous Simple Types

- gleiche Struktur wie Named Types
  - `simpleType` Element
    - aber ohne `name` (und u.U. `final`) Attribut
  - immer innerhalb von anderen Typ-Definitionen
    - `element`, `attribute`, `restriction`, `list`, `union`
- die Möglichkeiten sind genau die gleichen

```
<xsd:attribute name="size">
    <xsd:simpleType>
        <xsd:restriction base="xsd:integer">
            <xsd:minInclusive value="2"/>
            <xsd:maxInclusive value="18"/>
        </xsd:restriction>
    </xsd:simpleType>
</xsd:attribute>
```


Simple Type Restrictions

- Simple Types können durch *Restriction* von anderen Simple Types abgeleitet werden
  - der *Base Type* ist immer ebenfalls ein Simple Type
  - Wurzel dieser Hierarchie ist der `anySimpleType`
  - Erweiterungen sind nur möglich, indem der Simple Type zum Complex Type erweitert wird
- Restrictions enthalten *Facets*
  - Facets sind durch vorgegebene Elemente definiert
  - jede Restriction enthält 0-n Facets
    - Facets können auch wiederholt werden
  - wichtiges Werkzeug zur exakten Typ-Definition

Facets

- definieren Einschränkungen von Wertebereichen von Simple Types
- Einteilung in zwei Klassen von Facets
  - fundamental Facets
    - grundlegende Eigenschaften
  - constraining (or non-fundamental) Facets
    - einschränkende Eigenschaften
- Facets haben einen Wert
- die meisten Facets können fixiert werden

Constraining Facets

- die praktisch anwendbaren Facets
  - definieren Einschränkungen von Wertebereichen
  - können bei Typableitungen verschärft werden
- es gibt 12 Typen von Constraining Facets
  - `length, minLength, maxLength, pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minExclusive, minInclusive, totalDigits, fractionDigits`
  - nicht alle Facets sind für alle Typen sinnvoll
  - Verfügbarkeit richtet sich nach Primitive Type

Primitive Types Facets (I)

string	length, minLength, maxLength, pattern, enumeration, whiteSpace
boolean	pattern, whiteSpace
float	pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive
double	pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive
decimal	totalDigits, fractionDigits, pattern, whiteSpace, enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive
duration	pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive
dateTime	pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive
time	pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive
date	pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive

Primitive Types Facets (II)

gYearMonth	pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive
gYear	pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive
gMonthDay	pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive
gDay	pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive
gMonth	pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive
hexBinary	length, minLength, maxLength, pattern, enumeration, whiteSpace
base64Binary	length, minLength, maxLength, pattern, enumeration, whiteSpace
anyURI	length, minLength, maxLength, pattern, enumeration, whiteSpace
QName	length, minLength, maxLength, pattern, enumeration, whiteSpace
NOTATION	length, minLength, maxLength, pattern, enumeration, whiteSpace

Verwendung von Facets

- Facets können fixiert werden
  - keine weitere Änderung in Subtypen erlaubt
  - wird mit dem `fixed` Attribut angegeben
  - nicht möglich für `pattern` und `enumeration`
- Facets werden vererbt
  - entlang der gesamten Typenhierachie
  - wiederholte Facets müssen restriktiver sein
    - keine Erweiterung der Einschränkungen erlaubt
    - gilt auch für `pattern` und `enumeration`

Was man mit Facets nicht kann...

- Facets schränken einen Simple Type ein
  - in verschiedenen Dimensionen
  - in u.U. mehreren Schritten (Typableitung)
- können sich nicht auf andere Typen beziehen
  - z.B. End-Datum muss nach dem Anfang liegen
- können keine Auswertungen vornehmen
  - Summenfeld muss Summe bestimmter Felder sein
- d.h. konkrete Instanzen sind unberücksichtigt
- falls notwendig: zusätzliche Mechanismen
  - z.B. Schematron oder programmgesteuert

Patterns (Regular Expressions)

- Einschränkung von Simple Types
  - Restrictions mit dem **`pattern`** Element
- Beschränkung der lexikalischen Werte
- einfacher Aufbau der Ausdrücke
  - recht ähnlich den RegEx aus Perl, vi, grep, ...
  - zusätzlich Einbeziehung von Unicode Klassen
- sollten fast immer verwendet werden
  - z.B. alle Texte auf ISO 8859-1 beschränken
  - erleichtert das Leben der Applikationssoftware

Simple Types in XML Schema

- bisher haben wir Atomic Types betrachtet
  - Datentypen gemäss der Typhierarchie
  - Atomic Types erlauben genau einen Wert gemäss genau eines Typs (Typen können erweitert werden)
- Erweiterung des "genau einen Wert": Lists
  - Listen von Werten sind erlaubt
  - es dürfen nur Simple Types verwendet werden
  - Trennung mit XML Whitespace
- Erweiterung des "genau eines Typs": Unions
  - Mengen von Typen sind erlaubt
  - es dürfen nur Simple Types verwendet werden

Simple Type Derivations

		Derivation		
		Restriction	**List**	**Union**
Base Type	**Atomic**	Atomic	List	Union
	List	List		Union
	Union	Union	List	Union

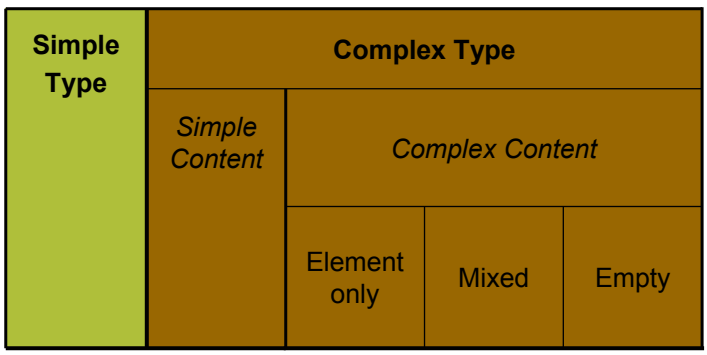
Was sind "Complex Types"?

- Elemente mit komplexen Typ haben entweder Child Elements oder Attribute
- ein komplexer Typ hat (meist) entweder ein Attribut oder ein Element
- Attribute haben auch einen Typ, dieser kann aber nie ein *Complex Type* sein.
- Complex Types haben entweder einen Namen (global definiert) oder sind anonym (lokal definiert)
- Elemente in einem *Complex Type* können nur dann den gleichen Namen haben, wenn sie auch vom gleichen Typ sind (andere Eigenschaften dürfen aber verschieden sein, z.B. `minOccurs`)

Content in Complex Types

- Content eines Elementes sind die Daten (Zeichenketten) und Child Elements zwischen den beiden Tags des Elementes
- Arten von Content für komplexe Typen:
  - Simple Content
  - Complex Content:
    - Element-only Content
    - Mixed Content
    - Empty Content

Complex Types + Content Types



Wieso Type Derivation?

- "vom Allgemeinen zum Speziellen"
- Erweiterung von bestehenden Typ-Definitionen (Original bleibt erhalten)
- abgeleitete Typen können zur Verfeinerung eines Schemas benutzt werden
- oder erlauben die Weiterverwendung von bereits definierten Typen (statt mit Copy/Paste Typ-Deklaration kopieren und anpassen)
- objekt-orientierter Ansatz

Type Extension und Restriction

- *Type Extension* (Erweiterung):
  - bestehenden Typ mit Attributen und/oder Child Elements erweitern
  - Werte des Basis-Typs sind nicht unbedingt auch gültige Werte des erweiterten Typs
- *Type Restriction* (Einschränkung):
  - bestehender Typ wird eingeschränkt
  - Wertebereich eines eingeschränkten Typs ist Untermenge des Basis-Typs
  - alle Werte des eingeschränkten Typs gehören auch in den Wertebereich des Basis-Typs
- Complex Types können nicht gleichzeitig erweitert und eingeschränkt werden (nur in 2 Schritten)

Erlaubte Typ-Erweiterungen

				Base Type				
				Simple Type	Complex Type			
					Simple Content	Complex Content		
						Element only	Mixed	Empty
Derived Type	Simple Type							
	Complex Type	Simple Content		Ja	Ja			
		Complex Content	Element-only			Ja		Ja
			Mixed				Ja	Ja
			Empty					Ja

Erlaubte Typ-Restriktionen

* Falls alle Child Elements optional sind				Base Type				
				Simple Type	Complex Type			
					Simple Content	Complex Content		
						Element only	Mixed	Empty
Derived Type	Simple Type			Ja				
	Complex Type	Simple Content			Ja		Ja*	
		Complex Content	Element-only			Ja	Ja	
			Mixed				Ja	
			Empty			Ja*	Ja*	Ja

Identity Constraints

- Erweiterung des `ID/IDREF(S)` Konzepts
- bieten eine ganze Reihe von Vorteilen
  - belassen den Typ der Keys
    - ermöglicht eine Typ-Validierung der Keys
    - ermöglicht verschiedene Key Typen, z.B. Integers/Strings
  - ermöglichen Keys über mehrere Felder
  - ermöglichen scoped Keys (d.h. kontext-abhängig)
  - beruhen auf Wertegleichheit (z.B. `+002 = 2`)
  - können auch auf Elemente angewendet werden
- sind etwas komplizierter zu benutzen

`ID/IDREF(S)` in XML DTDs

- Referenzen von Elementen zu Elementen
- `ID`s werden als Attribute vergeben
  - sie müssen gültige XML Namen sein
  - sie müssen im XML Dokument eindeutig sein
- `IDREF(S)` werden als Attribute vergeben
  - sie müssen existierende `ID`s referenzieren
- keine Möglichkeit der eigenen Typisierung
  - Identifikation einzig über den Attributtyp `ID`
    - keine Integers als Keys möglich (nur XML Namen)
  - unabhängig vom Elementtyp

Identity Constraints Eigenschaften

- Identity Contraints sind Einschränkungen
  - genauere Spezifikation des Inhalts
  - Keys/Referenzen nur eine mögliche Anwendung
- in vielen Fällen anwendbar
  - genauere Spezifikation des Schemas
  - prüfen deklarativ auf semantische Probleme
- nicht alles kann angegeben werden
  - keine kausalen Abhängigkeiten
  - programmierte Tests sind immer noch notwendig
- leider keine Verbindung mit dem Typ-Konzept

Reusable Groups

- wiederverwendbare Gruppen
  - Elemente (Named Model Groups)
  - Attribute (Attribute Groups)
- Deklaration von Gruppen
  - immer gemeinsam verwendete Elemente/Attribute
  - andere Art der Beziehung als bei Type Derivation
    - lockerer Zusammenhang statt Spezialisierung
- Referenzierung über Gruppennamen
- entspricht Parameter Entities in DTDs
  - semantisch wohldefiniert und damit sauberer

Substitution Groups

- Ersetzen von Elementen durch andere Elemente
  - es müssen abgeleitete Typen sein
- Beispiel:
  - Element Kurs könnte in einer Liste von Kursen ersetzt werden durch Elemente wie Kochkurs, Javakurs etc
- ermöglichen heterogene Aufzählungen
  - Liste von Elementen unterschiedlicher Typen in einer Schema-Instanz
- Substitution Groups machen solche Listen lesbarer

XML Schema Processor

- Software für die Validierung von Infosets
  - oftmals XML Parser mit Schema Unterstützung
    - unterstützen neben DTDs auch XML Schema
  - viele Implementierungen sind noch neu
- XML Schema lässt Spielraum
  - verschiedene Parser machen verschiedene Dinge
  - nicht alle Features müssen unterstützt werden
    - Lokalisierung von Schema Dokumenten
    - Zugriff auf das Resultat der Validierung
  - Requirements definieren und abklären

XML Schema Validation

- XML Schema arbeitet auf dem Infoset
  - Eingabe der Validierung ist ein XML Infoset
  - das Infoset wird validiert
    - je nach `processContents` Anweisungen
    - d.h. u.U. werden Teile nicht validiert
  - das Infoset wird um das Resultat ergänzt
- Post Schema Validation Infoset Contributions
  - XML kennt ähnliches (Default oder Fixed Attribute)
  - sehr viel detaillierter als eine ja/nein Entscheidung
  - zusätzliche Informationen im Infoset
  - können von Programmen ausgewertet werden

Effekte der Schema-Validierung

- das Infoset verändert sich
  - PSVI Contributions
  - aber auch "normale" Infoset Informationen
    - z.B. werden Attributdefaults ergänzt
- sehr wichtig für die Weiterverarbeitung
  - ein Schema sollte von Processing Guidelines begleitet werden
  - u.U. Minimalbedingungen an einen Parser
- XML Schema ist keine komplette Lösung
  - das Umfeld muss ebenso betrachtet werden

Probleme und Stand der Technik

- XML Schema ist aufwendig
  - deutlich mehr Aufwand als DTDs
- XML Schema ist fehlerhaft
  - der Standard enthält Fehler im Typsystem
- XML Schema Software ist fehlerhaft
  - oftmals nicht der volle Standard implementiert
  - nicht auf eine Software verlassen
    - falsch validierte Dokumente sind ein Problem
    - falsche XML Schemas sind ein grosses Problem
- Test-Suite ist in Entwicklung
  - u.U. Entwicklung eines offiziellen Conformance Test

Und trotzdem: XML Schema

- andere Software ist auch fehlerhaft
  - DTD-Validierung ist auch nicht perfekt
  - programmierte Tests sind es auch nicht
  - XMl Schema Validierung wird besser werden
- XML Schema verbessert die Modellierung
  - man muss mehr nachdenken über seine Daten
  - man kann Modellierungstools benutzen
- XML Schema als B2B Basis
  - Contract auf Basis XML Schema statt B2B
  - bessere formale Übereinkunft
  - weniger Interpretationsspielraum auf beiden Seiten

"Validieren" von XML Schemas

- Validation definiert XML Dokument Validierung
  - Ausgangspunkt ist ein korrektes XML Schema
  - wie kommt man zu einem korrekten Schema?
    - XML Schema selber sagt nichts dazu (z.B. lazy vs. eager)
- Apache Xerces XML Schema Parser
  - `http://xml.apache.org/xerces2-j/`
- Schema Quality Checker von IBM Alphaworks
  - `http://www.alphaworks.ibm.com/tech/xmlsqc`
  - validiert keine Dokumente, sondern prüft Schemas
  - Achtung: funktioniert nicht mit Java 2 v 1.4.x

Und so kann man vorgehen...

1. beginnen mit einem einfachen XML Schema
   - erstellt mit einem graphischen Tool wie XML Spy
   - konvertiert aus einer bestehenden DTD
2. "validieren" des XML Schema mit Tools
3. validieren von Beispiel- oder Livedaten
   - Toolunterstützung zur Erstellung von Beispieldaten
4. inkrementelle Verfeinerung des XML Schema
   - z.B. Simple Type Facets, insbesondere Patterns
5. verbleibende Lücken der Validierung feststellen
   - Lücken füllen mit Schematron o.ä. oder Code

Alternativen & Ergänzungen

- XML Schema kann nicht alles
  - keine Auswertung von Instanzen
    - z.B. ein Summenfeld in einer Rechnung
  - keine Abhängigkeiten zwischen Instanzen
    - ein Integer muss grösser sein als ein anderer
- Schematron validiert Instanzen
  - *Tree Patterns* statt Grammatiken
    - finden von *Context Nodes*, Überprüfen von Bedingungen
    - beliebige XPath Ausdrücke sind erlaubt (Effizienz!)
  - kann eine Grammatik ergänzen oder ersetzen
  - Schematron Schema wird nach XSLT übersetzt
    - erfordert keine spezielle Infrastruktur oder Software

Zusammenfassung

- XML Schema als DTD++
  - Weiterentwicklung aufgrund von realem Bedarf
  - Verbesserung in zwei Teilbereichen
    - Datentypen
    - Datenstrukturen
- XML Schema Simple Types
  - einfachen Typenkonzept
  - anwendungsorientierte Datentypen
  - Ableitung von Typen möglich
    - Einschränkungen von Wertebereichen (Facets)
    - Verwendung von List oder Union Types