

Lecture on

# Schema Design

„Advanced XML Schema“

Walter Kriha

# Goals

- Understand the deficits of Document Type Definitions
- Understand the goals of XML Schema
- Learn basic XML Schema elements
- Learn to design flexible schemas
- Limits of XML Schema, e.g. with respect to JDF modelling

XML Schema pushes the limits on what a validating parser can do with respect to making sure that a certain instance complies to a specific document type. This fits nicely to growing industry efforts to standardize data exchange or production workflows using XML documents. In this context it is vital that all partners agree on how a compliant document looks.

## The deficits of DTDs

- Not in XML syntax, requires special parsing.
- No way to validate content of elements. Little support for validating attributes
- No namespace support if e.g. parts of different DTDs should be combined
- No definition of new types based on existing types

# Example: Deficits of DTDs

<!Element list (.....) > <!-- this is not XML syntax: new APIs and tools are needed to read it -->

<!Element list (entry+) > <!-- what if you need EXACTLY 56 entries? -->

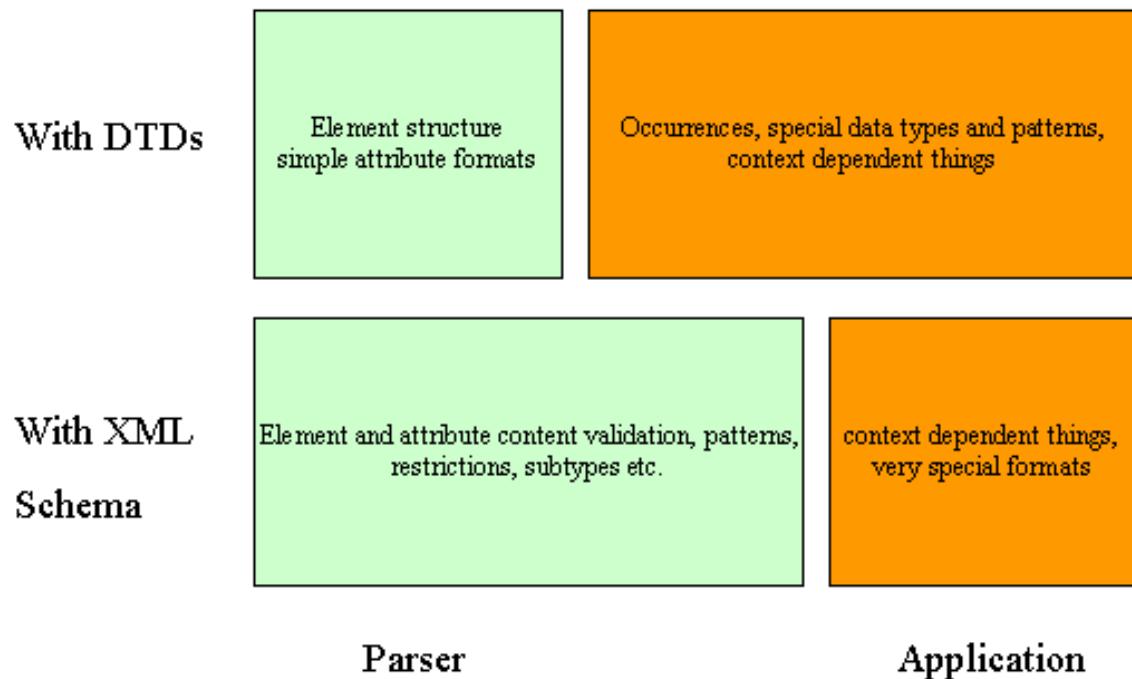
<!Element mylist (entry+, special) > <!-- what if you want a new type of list element which is like „list“ only with something more? Can you „derive“ the new type or do you have to copy the content model? -->

<!Element isbn (#PCDATA) > <!-- You cannot express that the ISBN number is always 10 long, divided into 4 blocks of 1,3,5 and 1 each etc. (0-123-12345-1) -->

<!Element country (#PCDATA) > <!-- would you like to restrict the content to the official country names only (enumeration) -->

To be fair: those demands come mostly from data-centric applications. The advantage of XML-Schema for regular authors is much less clear.

# Validation in Parser vs. Application



XML Schema increases ways for a parser to check conformance of instances due to better data types and a more specific element structure declaration (e.g. how many times element X has to show up in a certain location)

# Element Content Validation

## DTD (classic)

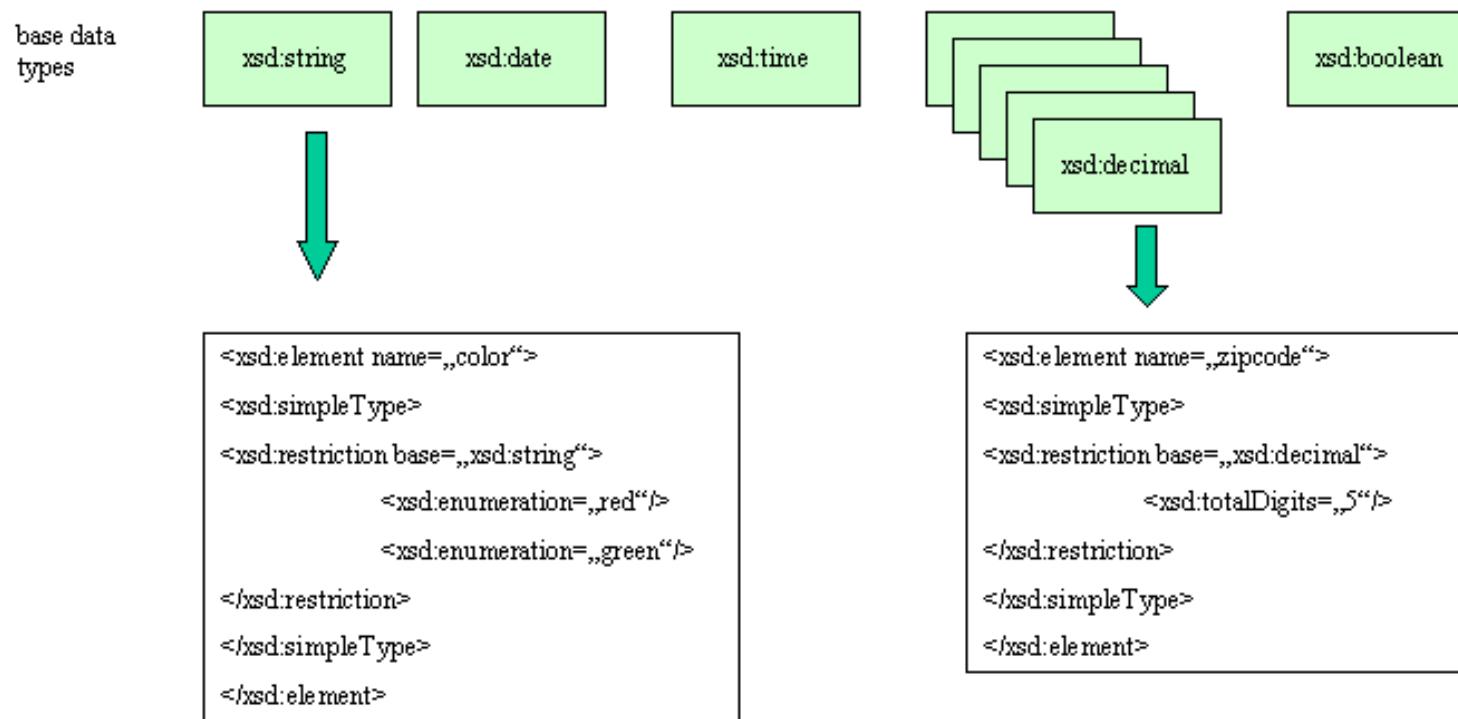
	<Element zipcode (#PCDATA)>
instance	<zipcode> foobar </zipcode>

## XML Schema

	<xsd:element name=„zipcode“ type=„xsd:decimal“/>
instance	<zipcode> 12345 </zipcode>

The classic dtd approach is unable to restrict the content of the element „zipcode“ to decimal numbers only. XML Schema can express that constraint and the parser will check the instance if it conforms to the specification

# Data Types and Restrictions (facets)



XML provides basic data types for numbers, strings, date, time, boolean, urls etc. Users can base their own types on those default or built in types by restricting the possible values those types can take. Possible restrictions depend on the base type.

## Frequently used restrictions

- enumeration (e.g. available colors)
- pattern (some value based on regular expressions: [a-z])
- minInclusive, maxInclusive (a range of possible values)
- minExclusive, maxExclusive (a range of possible values)
- minLength, maxLength, length
- totalDigits, fractionDigits
- whiteSpace (deals with tabs, newlines, CRs etc.)

Please look at the specific base data type to find which restrictions (also called facets) are possible in this case

## Example: enumeration (from JDF)

```
<xsd:simpleType name="eAppOS_>
    <xsd:restriction base="xsd:NMTOKEN">
        <xsd:enumeration value="Unknown"/>
        <xsd:enumeration value="Mac"/>
        <xsd:enumeration value="Windows"/>
        <xsd:enumeration value="Linux"/>
        <xsd:enumeration value="Solaris"/>
        <xsd:enumeration value="IRIX"/>
        <xsd:enumeration value="DG_UX"/>
        <xsd:enumeration value="HP_UX"/>
    </xsd:restriction>
</xsd:simpleType>
```

This type of enumeration occurs very often in industry schemas. JDF is literally riddled with such definitions. The advantage: the parser can easily detect misspellings or new elements. The disadvantage: a new element of the enumeration is a schema change!

# Specify content: string data type

`xsd:string` (content contains a character string)

Restrictions:

- enumeration
- length
- maxLength
- pattern
- whiteSpace

pattern examples: `xs:pattern = „here goes pattern“`

`[abc]` = can contain either a or b or c

`[a-zA-Z] [a-zA-Z]` = can contain 2 lowercase or uppercase letters

`„foo|bar“` = can contain either „foo“ or „bar“

## Specify content: date/time data types

xsd:date (CCYY-MM-DD format is required)

xsd:time (HH-MM-SS)

xsd:dateTime (CCYY-MM-DDT HH-MM-SS) (a good timestamp!)

xsd:duration

Without those data types the handling of international dates and times becomes very tedious and error prone: 11/02/2002 could be November 2<sup>nd</sup> or February 11<sup>th</sup> of 2002. Of course you are always free to build your own date and time elements with possibly month, day, year, century elements etc. But all the convenient restrictions (e.g. starting dates, periods etc.) work only with the standard data types which parsers know.

## Specify content: numeric data types

xsd:decimal

xsd:integer

xsd:byte

xsd:long

xsd:int

xsd:negativeInteger, nonNegativeInteger, positiveInteger etc.

xsd:unsignedInt, Long, Short etc.

Technical schemas have most datatypes available. Most of the data types are themselves derived from xsd:decimal

# Element Structure Validation

## DTD (classic)

```
<Element container      (item*)>
<Element item          (#PCDATA)>

instance
<container><item> foo </item></container>
```

## XML Schema

```
<xsd:complexType name=„container“>
  <xsd:sequence>
    <xsd:element name=„item“ type=„xsd:string“ minOccurs=„2“ />
  </xsd:sequence></xsd:complexType>

instance
<container> <item>foo</item><item>bar</item></container>
```

The classic dtd approach is unable to express the number of occurrences of a child element except through repetition. XML Schema can express that constraint and the parser will check the instance if it contains the required number of „item“ elements.

# XSD Simple Types

```
<xsd:element name=„firstName“      type=„xsd:string“/>
```

An Element without child elements AND without attributes is a so called simple element. It represents a leaf node in the document graph.

# XSD Complex Types

```
<xsd:complexType name=„car“>
  <xsd:sequence>
    <xsd:element name=„wheel“ type=„xsd:string“ minOccurs=„4“ />
    <xsd:element name=„engine“ type=„xsd:string“ minOccurs=„1“ />
    <xsd:element name=„roof“ type=„xsd:string“ />
  </xsd:sequence>
  <xsd:attribute name=„prodID“ type=„xsd:decimal“ />
</xsd:complexType>
```

This car element contains three children which have to appear in exactly this order in the instance. It also has one attribute – a decimal product identification.

# Deriving Complex Types

```
<xsd:complexType name=„car“>
  <xsd:sequence>
    <xsd:element name=„wheel“ type=„xsd:string“ minOccurs=„4“ />
    <xsd:element name=„engine“ type=„xsd:string“ minOccurs=„1“ />
    <xsd:element name=„roof“ type=„xsd:string“ />
  </xsd:sequence>
  <xsd:attribute name=„prodID“ type=„xsd:decimal“/>
</xsd:complexType>
<xsd:complexType name=„truck“>
  <xsd:complexContent>
    <xsd:extension base=„car“>
      <xsd:element name=„trailer“ type=„xsd:string“ minOccurs=„0“ />
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Another complex type can be based on an existing type and extended with further elements and attributes

# XSD attributes

```
<xsd:attribute name=„prodID“ type=„xsd:decimal“ use=„optional“/>  
(default)  
  
<xsd:attribute name=„prodID“ type=„xsd:decimal“ use=„required“/>  
(instance needs to set attribute)  
  
<xsd:attribute name=„prodID“ type=„xsd:decimal“ default=„4711“/>  
(will be used if instance does NOT specify attribute value)  
  
<xsd:attribute name=„prodID“ type=„xsd:decimal“ fixed=„1122“/>  
(instance does not need to set attribute value because it will be set to 1122  
by default.)
```

„Optional“ is the default mode for all attributes. „Fixed“ is very useful to propagate certain attribute values into instances without requiring the author to type them in. „Required“ will create an error if the attribute value is not set.

# Element or Attribute?

- Elements are always extensible (e.g. new children)
- Attributes are always of type simple and cannot acquire a more complicated structure
- Certain applications or processors may expect some data in attributed and others in element content.

Religious wars have been fought over the question whether one should use elements or attributes to keep content. Both are perfectly legal ways but from an extensibility point of view elements are more flexible because they can extend their internal structure while attributes cannot. A good rule of thumb is also: if it is meta-information about the elements and their contents then it is an attribute. Or: if an attribute is deleted and the document loses significant content, then the attribute should probably be an element

# Referring to Types

```
<xsd:element name=„bumperCar“ type=„car“>
<xsd:complexType name=„car“>
    <xsd:sequence>
        <xsd:element name=„wheel“ type=„xsd:string“ minOccurs=„4“ />
        <xsd:element name=„engine“ type=„xsd:string“ minOccurs=„1“ />
        <xsd:element name=„roof“ type=„xsd:string“ />
    </xsd:sequence>
    <xsd:attribute name=„prodID“ type=„xsd:decimal“/>
</xsd:complexType>
```

Here element bumperCar refers to „car“ as its type definition. An element „bumpercar“ needs to contain exactly the same children as specified for type „car“

# Context Dependent Validation

## XML Schema

```
<xsd:complexType name=„container“>
  <xsd:sequence>
    <xsd:element name=„item“ type=„xsd:string“ minOccurs=2>
    </xsd:sequence>
    <xsd:element name=„zipcode“ type=„xsd:decimal“/>
```

Rule: if zipcode == X,  
minOccurs == Y

THIS IS NOT  
POSSIBLE!

## instance

```
<zipcode> 12345 </zipcode>
```

```
<container> <item>foo</item><item>bar</item></container>
```

Even XML Schema cannot validate context dependent values. This was e.g. a problem for JDF. In those cases the processing applications are extended to check such dependencies.

# XML Schema goals

- XML syntax only
- Element Type definitions and reusable type definitions
- Strong data typing with derived types
- Re-use through reference of elements
- Content validation of element and attribute content

XML Schema pushes the limits on what a validating parser can do with respect to making sure that a certain instance complies to a specific document type. This fits nicely to growing industry efforts to standardize data exchange or production workflows using XML documents. In this context it is vital that all partners agree on how a compliant document looks.

# Formal parts of XML Schema

carSchema.xsd:

```
<?xml version=„1.0“ encoding=„UTF-8“?>
<xsd:schema xmlns:xsd=„http://www.w3.org/2001/XMLSchema“>
<xsd:element cars.....>

</xsd:schema>
```

carInstance.xml

```
<?xml version=„1.0“ encoding=„UTF-8“?>
<cars xmlns:xsi=„http://www.w3.org/2001/XMLSchema-instance“
      xsi:SchemaLocation=http://www.cars.com/carSchema>
<...>
</cars>
```

**XMLSchema** is the namespace for schema elements. „xsd:“ refers to this namespace. The prefix can be changed through the `xmlns:xxxx` attribute in case of conflicts.

# XML namespaces: the problem

XML schema ONE:

```
<xsd:element name=„foo“ type=„car“ />
```

XML schema Two:

```
<xsd:element name=„foo“ type=„book“ />
```

XML Instance using BOTH foo elements:

```
<container>
  <foo>
    <wheels>....</wheels>
    <doors>....</doors>
  </foo>
  <foo>
    <ISBN>.....</ISBN>
    <pageCount>...</pageCount>
  </foo>
</container>
```

XML Authors can pick any name they want for their elements. Thus the danger of name collisions exists. How would the parser validate the instance above? It does not know WHICH foo you mean! A mechanism to disambiguate the elements is needed: XML namespaces.

# XML namespaces: the solution

XML schema ONE:

```
<xsd:element name=„foo“ type=„car“ />
```

XML schema Two:

```
<xsd:element name=„foo“ type=„book“ />
```

XML Instance using BOTH foo elements:

```
<container xmlns:car=„http://www.cars.com/carSchema“  
           xmlns:book=„http://www.books.com/bookSchema“>  
  
  <car:foo>  
    <wheels>....</wheels>  
    <doors>....</doors>  
  </car:foo>  
  <book:foo>  
    <ISBN>.....</ISBN>  
    <pageCount>...</pageCount>  
  </book:foo></container>
```

The namespace prefixes are used to distinguish the elements with the same name.

BTW: the namespace for jdf is: [http://www.CIP4.org/JDFSchema\\_1](http://www.CIP4.org/JDFSchema_1)

# JDF instances (examples 1)

This is a simple example of a JDF that describes color conversion for one file.

```
<?xml version='1.0' encoding='utf-8' ?>
<JDF ID="HDM20001101102611" Type="ColorSpaceConversion"
JobID="HDM20001101102611"
Status="waiting" Version="1.0">
<!--(c) Heidelberger Druckmaschinen AG 1999-2000-->
<!--Warning: preliminary format; use at your own risk-->
<NodeInfo/>
<ResourcePool>
<RunList ID="Link0003" Class="Parameter" Status="available">
<Run>
<RunSeparation Pages="0~-1">
<LayoutElement>
<FileSpec FileName="in/colortest.pdf"/>
</LayoutElement>
</RunSeparation>
</Run>
</RunList>
.....
```

## JDF schema (examples 1)

```
<AuditPool>
  <Created Author="Rainer's JDFWriter 0.2000"
   TimeStamp="2000-11-01T10:26:11+01:00"/>
  <Modified Author="EatJDF Complete: task=*&
   TimeStamp="2000-11-01T10:26:57+01:00"/>
  <PhaseTime End="2000-11-01T10:26:57+01:00"
    Start="2000-11-01T10:26:57+01:00"
    Status="setup" TimeStamp="2000-11-01T10:26:57+01:00"/>
  <PhaseTime End="2000-11-01T10:26:57+01:00"
    Start="2000-11-01T10:26:57+01:00"
    Status="in_progress" TimeStamp="2000-11-01T10:26:57+01:00"/>
  <PhaseTime End="2000-11-01T10:26:57+01:00"
    Start="2000-11-01T10:26:57+01:00"
    Status="cleanup" TimeStamp="2000-11-01T10:26:57+01:00"/>
  <ProcessRun End="2000-11-01T10:26:57+01:00"
    Start="2000-11-01T10:26:57+01:00"
    EndStatus="Completed" TimeStamp="2000-11-01T10:26:57+01:00"/>
</AuditPool>
```

Note the use of xsd:dateTime basic data type with timezone (+1)

## JDF schema (examples 2)

```
<xsd:simpleType name="eNodeStatus_>
    <xsd:annotation>
        <xsd:documentation>          JDF::Status, T3.3
        </xsd:documentation>
    </xsd:annotation>
    <xsd:restriction base="xsd:NMTOKEN">
        <xsd:enumeration value="Waiting"/>
        <xsd:enumeration value="Ready"/>
        <xsd:enumeration value="FailedTestRun"/>
        <xsd:enumeration value="Setup"/>
        <xsd:enumeration value="InProgress"/>
        <xsd:enumeration value="Cleanup"/>
        <xsd:enumeration value="Spawned"/>
        <xsd:enumeration value="Stopped"/>
        <xsd:enumeration value="Completed"/>
        <xsd:enumeration value="Aborted"/>
        <xsd:enumeration value="Pool"/>
    </xsd:restriction>
</xsd:simpleType>
```

Compare the status values from the audit records with these enumerations!

## Next Session

- Advanced concepts: extensions, qualifications, namespaces
- how to design a schema
- examples from JDF

Please read the JDF related documentation on [www.cip4.org!](http://www.cip4.org)

## Resources (1)

- Graham Mann, XML Schema for Job Definition Format
- XML Schema Part 0: Primer, [www.w3.org/TR/schema-0](http://www.w3.org/TR/schema-0)
- JDF instance examples from [www.coverpages.org/jdf50-Examples.txt](http://www.coverpages.org/jdf50-Examples.txt)
- xml schema tutorial,  
[www.w3schools.com/schema/default.asp](http://www.w3schools.com/schema/default.asp)
- <http://www.xfront.com/index.html#tutorials> hosts excellent XSD and XSL tutorials from Roger Costello.