

Angewandte Softwareentwicklung

XML

WS 2014/2015



Markus Berg

Hochschule Wismar

Fakultät für Ingenieurwissenschaften

Bereich Elektrotechnik und Informatik

markus.berg@hs-wismar.de

<http://mmberg.net>

Inhalt

- **Teil I**
 - Was ist XML?
 - Syntax / Aufbau
 - Strukturierung: XML Schema
 - Praktikum 3, Teil I
- **Teil II**
 - Suchen in XML: XPath
 - Transformationen mit XSLT
 - Parsing
 - Demos
 - Praktikum 3, Teil II

Teil I

XML & XSD



XML

- eXtensible Markup Language
- W3C Standard
- Textbasiert
- Streng genommen keine Auszeichnungssprache, sondern Sprache zur Definition von Auszeichnungssprachen (z.B. XHTML) bzw. Austauschformaten (z.B. SOAP)
 - „Metabeschreibungssprache“
 - Somit ist es selbst kein bestimmtes Dokumentenformat
- Gibt lediglich eine Syntax für alle auf ihr basierenden Sprachen/Dokumenten vor (Wohlgeformtheit)
- XML beschreibt Daten, nicht deren Verarbeitung oder Visualisierung

Markup

- „Auszeichnung“
 - Weist einem Inhalt eine Beschreibung zu, die die weitergehende Verarbeitung ermöglicht
 - 22.08.2014
 - `<datum>22.08.2014</datum>`
 - `<Person>`
 - `<name>Lena</name>`
 - `<gebdat>22.08.2014</gebdat>`
 - `</Person>`
 - Syntaktische bzw. semantische Auszeichnung
 - i.d.R. Semantik/Bedeutung
 - Aus einem „Datum“ (i.S.v. data) wird eine „Information“
 - Der 22.08.2014 ist ein Geburtsdatum bzw. das GebDat von Lena
 - Syntax über Datentyp spezifiziert

Beispiele

- XHTML
- VoiceXML
- XÖV
 - XWaffe
 - XAusländer
 - XMeld
- WSDL

Syntaxvorschriften von XML selbst

- Hierarchie verschiedener Elemente
 - Bestehen aus Tag und Inhalt
 - Tagname in spitzen Klammern (<tag>)
 - Müssen geöffnet (<tag>) und geschlossen werden (</tag>)
 - Inhalt steht dazwischen (<tag>inhalt</tag>)
 - Alles zusammen: Element

```
<vorlesung>  
  <name>Angewandte Softwareentwicklung</name>  
  <credits>5</credits>  
  <dozent>Berg</dozent>  
</vorlesung>
```

Element

- Elemente können Attribute haben

```
<vorlesung id="1234">  
  ...  
</vorlesung>
```

Syntaxvorschriften

- Nicht geschlossene Elemente nicht erlaubt
 - Was ist mit Elementen ohne Inhalt?
 - z.B. HTML `
` oder `<hr>`?
 - Kurzschreibweise für leere Elemente
 - `
</br>` → `
`
- Attributwerte werden in Anführungszeichen geschrieben
 - Einfach oder doppelt
 - Nicht gemischt

```
<buch id="3">...</buch>
```

```
<buch id='3'>...</buch>
```

Syntaxvorschriften

- Es muss immer ein Root-Element geben, das den gesamten Inhalt umschließt

```
<vorlesung>  
  <name>Angewandte Softwareentwicklung</name>  
  <credits>5</credits>  
  <dozent>Berg</dozent>  
</vorlesung>
```



```
<name>Angewandte Softwareentwicklung</name>  
<credits>5</credits>  
<dozent>Berg</dozent>
```

Syntaxvorschriften

- Elemente können statt Text auch Unterelemente enthalten
- Korrekte Schachtelung einhalten
 - D.h. zuletzt geöffnetes Tag wird als erstes geschlossen

```
<vorlesungen>
  <vorlesung>
    <name>Angewandte Softwareentwicklung</name>
    <credits>5</credits>
    <dozent>Berg</dozent>
  </vorlesung>
  <vorlesung>
    <name>Datenbanken</name>
    <credits>5</credits>
    <dozent>Düsterhöft</dozent>
  </vorlesung>
</vorlesungen>
```

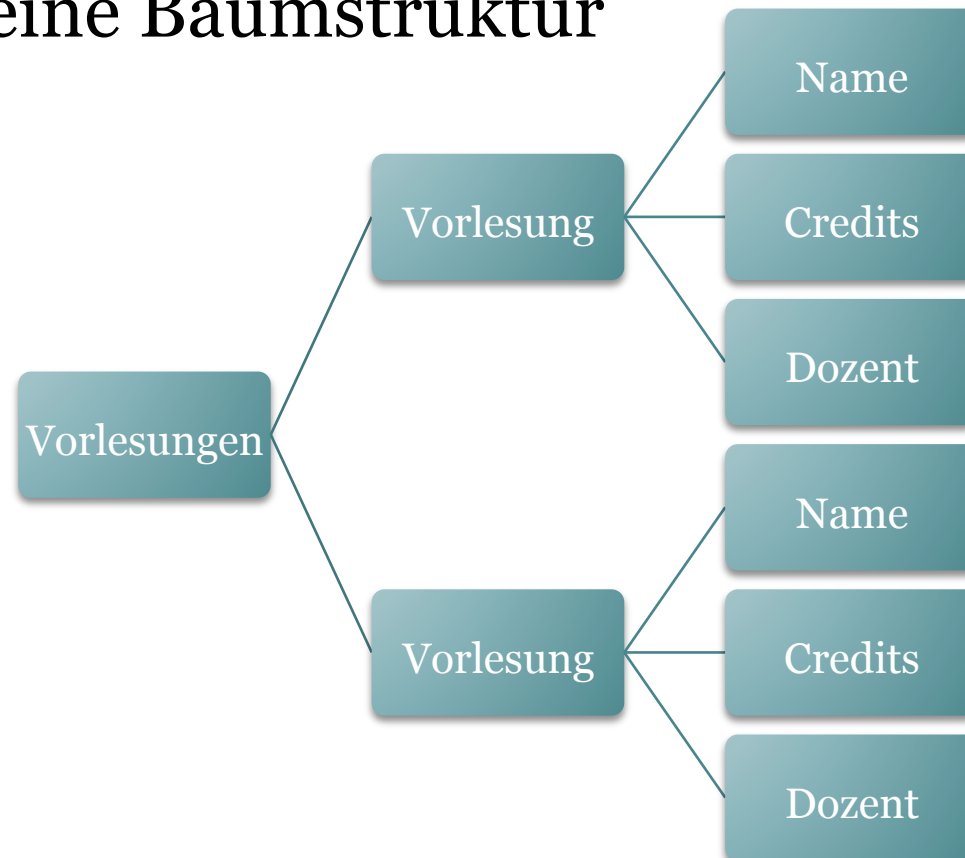


```
<a><b></a></b>
```

```
<vorlesungen>
  <vorlesung>
    <name>Angewandte Softwareentwicklung</name>
    <credits>5</credits>
    <dozent>Berg</dozent>
  <vorlesung>
    <name>Datenbanken</name>
    <credits>5</credits>
    <dozent>Düsterhöft</dozent>
  </vorlesung>
</vorlesungen>
```

Syntaxvorschriften

- XML ist eine Baumstruktur



Attribut oder Tag?

- Prinzipiell gleichwertig, aber:
 - Attribute können nicht weiter strukturiert werden
 - Attribute können nur einmal vorkommen



```
<vater kind="hans meier" kind="susi meier">  
  <vorname>Ulrich</vorname>  
  <nachname>Meier</nachname>  
</vater>
```

```
<vater>  
  <vorname>Ulrich</vorname>  
  <nachname>Meier</nachname>  
  <kinder>  
    <kind>  
      <vorname>Hans</vorname>  
      <nachname>Meier</nachname>  
    </kind>  
    <kind>  
      <vorname>Susi</vorname>  
      <nachname>Meier</nachname>  
    </kind>  
  </kinder>  
</vater>
```


Syntaxvorschriften

- Jedes Dokument beginnt mit Prolog

```
<?xml version="1.0"?>
```

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

```
<?xml version="1.0" encoding="utf-8"?>
```

- Sogenannte Processing Instruction (PI) für Interpreter
 - z.B. XML-Version
 - Zeichencodierung UTF-8
- Beginnen mit „<?“ Statt „<“
- Enden mit „?>“

Escapen von reservierten Zeichen

- ... Durch HTML-Entities
 - < → <
 - > → > (muss nicht escaped werden)
 - & → &
 - ‘ → '
 - “ → "
- ... oder CDATA (character data)
 - Alle Sonderzeichen erlaubt, wird vom Interpreter nicht als XML interpretiert

```
<text>  
<![CDATA[ Hier kann alles stehen & keinen “störts” ]]>  
</text>
```

Kommentare

- Wie in HTML

```
<!-- dies ist ein Kommentar -->
```

- Dürfen nicht innerhalb von Tags stehen

Namespaces & Prefixes

- Namensräume erlauben die Referenzierung von Elementen aus verschiedenen „Sprachen“ bzw. Schemas
 - Ähnlich der Qualifizierung über Packages in Java
 - `net.mmberg.Person.Name`
 - `net.mmberg.Vorlesung.Name`
 - Eindeutiger Identifier um Namenskonflikte zu vermeiden
 - In XML werden hierfür URLs genutzt
 - Müssen nicht real existieren
 - z.B. http://mmberg.net/schemas/Vorlesung_V1
 - Abgekürzt über Prefix (hier: „vorl“)

```
<myElement xmlns:vorl="http://mmberg.net/schemas/Vorlesung_V1">
```

Namespaces

- Deklarieren von Elementen eines Namespaces
 - Ansprechen über Präfix

```
<vorl:Vorlesung xmlns:vorl="http://mberg.net/schemas/Vorlesung_V1">
  <vorl:Name>ASE</vorl:Name>
  <vorl:Dozent>Berg</vorl:Dozent>
</vorl:Vorlesung>
```

- Kombinieren von Namensräumen

```
<Studiengang
xmlns:vorl="http://mberg.net/schemas/Vorlesung_V1"
xmlns:mit="http://mberg.net/schemas/Mitarbeiter_V1">
  <vorl:Vorlesung>
    <vorl:Name>ASE</vorl:Name>
    <vorl:Dozent>
      <mit:Name>Litschke</mit:Name>
      <mit:Titel>Prof.Dr.</mit:Titel>
    </vorl:Dozent>
  </vorl:Vorlesung>
</Studiengang>
```

Namespaces

- Standardnamensraum
 - Gilt implizit, wenn kein separates Präfix angegeben wird

```
<Studiengang
xmlns:="http://mberg.net/schemas/Studiengang"
xmlns:vorl="http://mberg.net/schemas/Vorlesung_V1"
xmlns:mit="http://mberg.net/schemas/Mitarbeiter_V1">
<name>Angewandte Informatik</name>
<vorl:Vorlesung>
  <vorl:Name>ASE</vorl:Name>
  <vorl:Dozent>
    <mit:Name>Berg</mit:Name>
    <mit:Bereich>EuI</mit:Bereich>
  </vorl:Dozent>
</vorl:Vorlesung>
</Studiengang>
```

Wohlgeformtheit

- Genau ein Wurzelement
- Korrekte Schachtelung (ergibt eine Baumstruktur)
- Elemente müssen immer geschlossen werden
- Reservierte Zeichen escapen bzw. CDATA-Blöcke benutzen
- Tags und Attribute bestehen aus
 - Buchstaben (Groß-/Kleinschreibung wird unterschieden)
 - Zahlen
 - Unterstriche
 - Punkte
 - Bindestriche
- Müssen mit Buchstabe oder Unterstrich beginnen
- Keine Leerzeichen!



Ist das Dokument wohlgeformt?

```
<?xml version="1.0" encoding="utf-8"?>
<fakultaet name="fiw">
  <dekan>Müller</dekan>
</fakultaet>
<servicepoint>
  <oeffnet>09:00</oeffnet>
  <schliesst>15:00</schliesst>
</servicepoint>
```

A**Ja****B****Nein**



Ist das Dokument wohlgeformt?

```
<?xml version="1.0" encoding="utf-8"?>  
<fakultaet name="fiw">  
  <dekan>Müller</DEKAN>  
</fakultaet>
```

A**Ja****B****Nein**

Definition der Syntax für Markupssprachen

- Bis jetzt nur Wohlgeformtheit
- Keine Datentypen
 - Ein Geburtsdatum ist vom Typ Datum
- Keine Strukturvorschrift (inhaltlich)
 - Eine Person besitzt genau einen Namen und ein Geburtsdatum
 - Eine Person kann mehrere Adressen haben

XML Schema (XSD)

- Formvorschrift
 - D.h. welche Elemente dürfen in welcher Reihenfolge und Anzahl vorkommen und welche Datentypen besitzen sie?
 - D.h. Syntax der Sprache definieren
 - Ermöglicht Validierung
- Vorgänger: DTD (Document Type Definition)
 - Limitierte Datentypen
 - Nicht XML-konforme Syntax
- Definiert in separater Datei (umgs. „Schema“)
- Schema ist ebenfalls XML-konform
- Dateiendung meist .xsd

XSD Aufbau

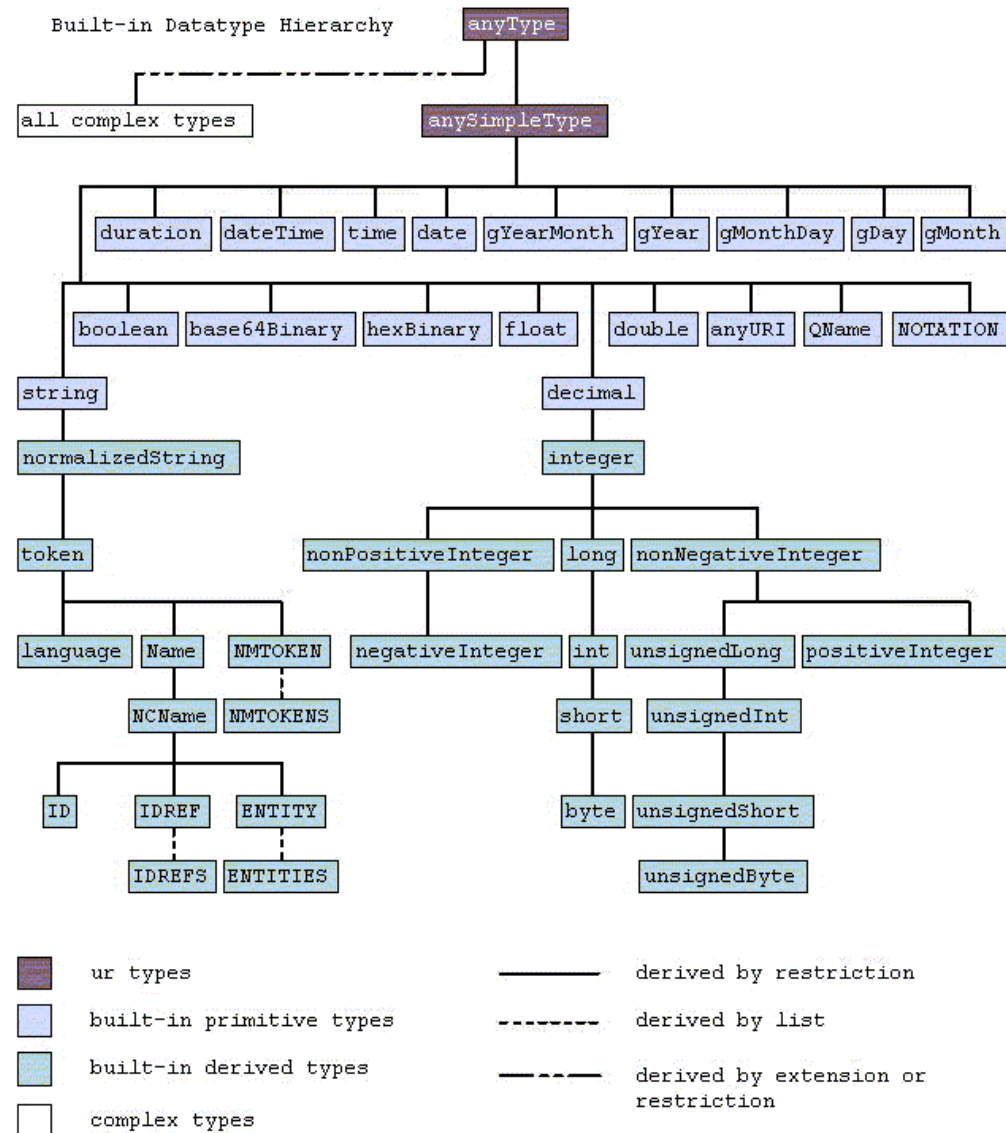
- Rahmen

```
<?xml version='1.0'?>  
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">  
<!-- hier kommen die Definitionen -->  
</xsd:schema>
```

- Alle XSD-Elemente kommen aus dem Namespace „http://www.w3.org/2001/XMLSchema“
- **targetNamespace**
 - Gibt an zu welchem Namespace die Elemente, die in diesem Schema definiert werden, gehören
 - D.h. alle Elemente einer Instanz des Schemas existieren in diesem Namespace

XSD Datentypen

- Simple (Standarddatentypen)
 - Haben keine Elemente oder Attribute
 - string
 - integer
 - boolean
 - date
 - ...
- Complex
 - Selbst definierte Typen, die sich aus Elementen/Attributen zusammensetzen
 - Die Elemente können dabei komplex oder einfach sein



Typen vs. Elemente

- Elemente haben einen Typ
 - Wie in der Programmierung: Variablen haben einen Typ
- Elemente sind von außen sichtbar bzw. werden im XML-Dokument genutzt
 - Elemente: Tags in einem XML-Dokument
- Typen sind für ein XML-Dokument nicht sichtbar (nur schemaintern)
- Typen und Elemente können in anderen Schemas benutzt und somit wiederverwendet werden

Definition eigener einfacher Typen I

- Simple Types
 - Das Element *vorname* ist vom Typ String
 - Keine Definition nötig, da built-in Datentyp

```
<xsd:element name="vorname" type="xsd:string"/>
```

- Erstellung neuer einfacher Typen durch Einschränkung
 - Sgn. Facetten
 - z.B.
 - minInclusive
 - minExclusive
 - maxInclusive
 - maxExclusive

```
<xsd:simpleType name="weekday">  
  <xsd:restriction base="xsd:integer">  
    <xsd:minInclusive value="1" />  
    <xsd:maxInclusive value="7" />  
  </xsd:restriction>  
</xsd:simpleType>
```

Definition eigener einfacher Typen II

- Liste von erlaubten Werten

```
<xsd:simpleType name="weekday">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="Montag"/>
    <xsd:enumeration value="Dienstag"/>
    <xsd:enumeration value="Mittwoch"/>
    <xsd:enumeration value="Donnerstag"/>
    <xsd:enumeration value="Freitag"/>
    <xsd:enumeration value="Samstag"/>
    <xsd:enumeration value="Sonntag"/>
  </xsd:restriction>
</xsd:simpleType>
```

- Weitere Facetten
 - minLength (Anzahl der Zeichen bei Strings)
 - maxLength
 - Pattern (Regulärer Ausdruck)
 - ...

Definition eigener komplexer Typen I

- Complex Types
 - Das Element `person` ist vom komplexen Typ `Person`
 - Eigene Definition des Typs (definiert den Aufbau)
 - Komplexe Typen enthalten Elemente mit einfachen oder wiederum komplexen Typen

```
<xsd:element name="person" type="Person"/>  
  
<xsd:complexType name="Person">  
  <xsd:sequence>  
    <xsd:element name="vorname" type="xsd:string"/>  
    <xsd:element name="nachname" type="xsd:string"/>  
  </xsd:sequence>  
</xsd:complexType>
```

```
<person>  
  <vorname>Peter</vorname>  
  <nachname>Müller</nachname>  
</person>
```

Definition eigener komplexer Typen II

- Sobald ein Element ein Attribut besitzt, ist es komplex
 - `<mitarbeiter akadGrad="Dr">Berg</mitarbeiter>`
 - Daher gibt es komplexe Typen mit einfachem Inhalt
- Komplexe Typen mit komplexem Inhalt werden innerhalb von folgenden Konstrukten definiert:
 - **sequence**
 - Eine Sequence nimmt ein oder mehrere Elemente auf
 - Reihenfolge muss eingehalten werden
 - **choice**
 - Choice darf Teil einer Sequence sein
 - Auswahl

Definition von Attributen

- Attribute dürfen nur einfache Datentypen haben
- Sie treten einmal oder gar nicht auf
- Kardinalitäten
 - use: required, optional, prohibited
 - default (Standardwert falls Attribut nicht angegeben, nur wenn use=optional)
 - fixed (Attribut hat immer diesen Wert)

```
<xsd:complexType name="person">  
  <xsd:attribute name="gender" type="xsd:string"  
    use="required"/>  
</xsd:complexType
```

Benennung

- Elemente und Typen dürfen den gleichen Namen haben
- Wenn Elemente aus verschiedenen Namensräumen stammen, dürfen sie den gleichen Namen haben
- Zwei Elemente in verschiedenen Typdefinitionen dürfen den gleichen Namen haben

Definition von Typen durch Erweiterung

- Existierender Typ „Person“ wird um ein Attribut erweitert:

```
<xsd:complexType name="Mitarbeiter">
  <xsd:complexContent>
    <xsd:extension base="Person">
      <xsd:attribute name="mid"
                    type="xsd:positiveInteger"
                    use="required" />
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

- ... bzw. um ein Element erweitert:

```
<xsd:complexType name="Mitarbeiter">
  <xsd:complexContent>
    <xsd:extension base="Person">
      <xsd:sequence>
        <xsd:element name="abteilung" type="xsd:string"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Kardinalitäten

- Wie oft muss ein Element mindestens vorkommen und wie oft darf es maximal vorkommen?

- `minOccurs`
- `maxOccurs`

```
<xsd:element name="vorname" type="xsd:string" minOccurs="0"
maxOccurs="unbounded"/>
```

- Optionale Elemente
 - `minOccurs=1`
- Listen
 - `maxOccurs=unbounded`
 - `maxOccurs=3`

Leeres Element vs. kein Element

```
<xsd:complexType name="Person">  
  <xsd:sequence>  
    <xsd:element name="name" type="xsd:string"/>  
    <xsd:element name="telefon" type="xsd:string" minOccurs="0"/>  
  </xsd:sequence>  
</xsd:complexType>
```

```
<Person>  
  <name>Berg</name>  
  <telefon/>  
</Person>
```

```
<Person>  
  <name>Berg</name>  
</Person>
```

- Stilfrage
- Je nach Schema ist rechte Variante nicht erlaubt (wenn kein minOccurs=0)
- Leeres Element kann nullPointer-Exceptions verhindern, wenn der Zugriffsversuch auf ein erwartetes Element erfolgt, das nicht existiert

Referenzen

- Innerhalb von Typdefinitionen auf andere Elemente verweisen

```
<xsd:element ref="Person" />
```

- Es kann nur auf Elemente, nicht auf Typen verwiesen werden, d.h. es muss zunächst ein Element vom entsprechenden Typ angelegt werden, bevor darauf referenziert werden kann.
- Wo ist der Unterschied?
 - Bei einer Referenz entsteht kein neuer Name, d.h. es wird der Name des Elementes genommen auf das verwiesen wird
 - Eine Referenz setzt ein (zusätzliches) Element voraus
- Ziel
 - Elemente, die häufig genutzt werden, wiederverwenden
 - Statt

```
<xsd:element name="vorname" type="xsd:string"/>
```

 immer wieder zu definieren, einmal definieren und über Referenzen wiederverwenden
 - Vereinfachung wird deutlicher bei komplexen Elementen

Referenz vs. Typ

```
<xsd:complexType name="Person">  
  ...  
</complexType>  
  
<xsd:element name="person" type="Person">  
  
<xsd:element name="individuum">  
  <xsd:sequence>
```

```
<xsd:element ref="person" />
```

```
<xsd:element name="mensch"  
  type="Person" />
```

```
</xsd:sequence>  
</xsd:element>
```

```
<individuum>  
  <person>  
    <name>Berg</name>  
  </person>  
</individuum>
```

```
<individuum>  
  <mensch>  
    <name>Berg</name>  
  </mensch>  
</individuum>
```

Schema einbinden

- Ziel: ein XML-Dokument erzeugen, das auf den Regeln des Schemas basiert
- D.h. Instanz eines Schemas erzeugen
 - bzw. Schema in Instanz referenzieren
- Einbindung durch Angabe im Root-Element (nach PI)
 - Kennzeichnung als Schemainstanz
 - Referenzieren des Schemas anhand von
 - Namespace (entspricht dem targetNamespace des Schemas)
 - Pfadangabe (relativ zum XML-Dokument)

```
<?xml version="1.0"?>
<meinRoot xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:my="http://meinNamespace"
xsi:schemaLocation="http://meinNamespace meinSchema.xsd">
```

Imports / Includes

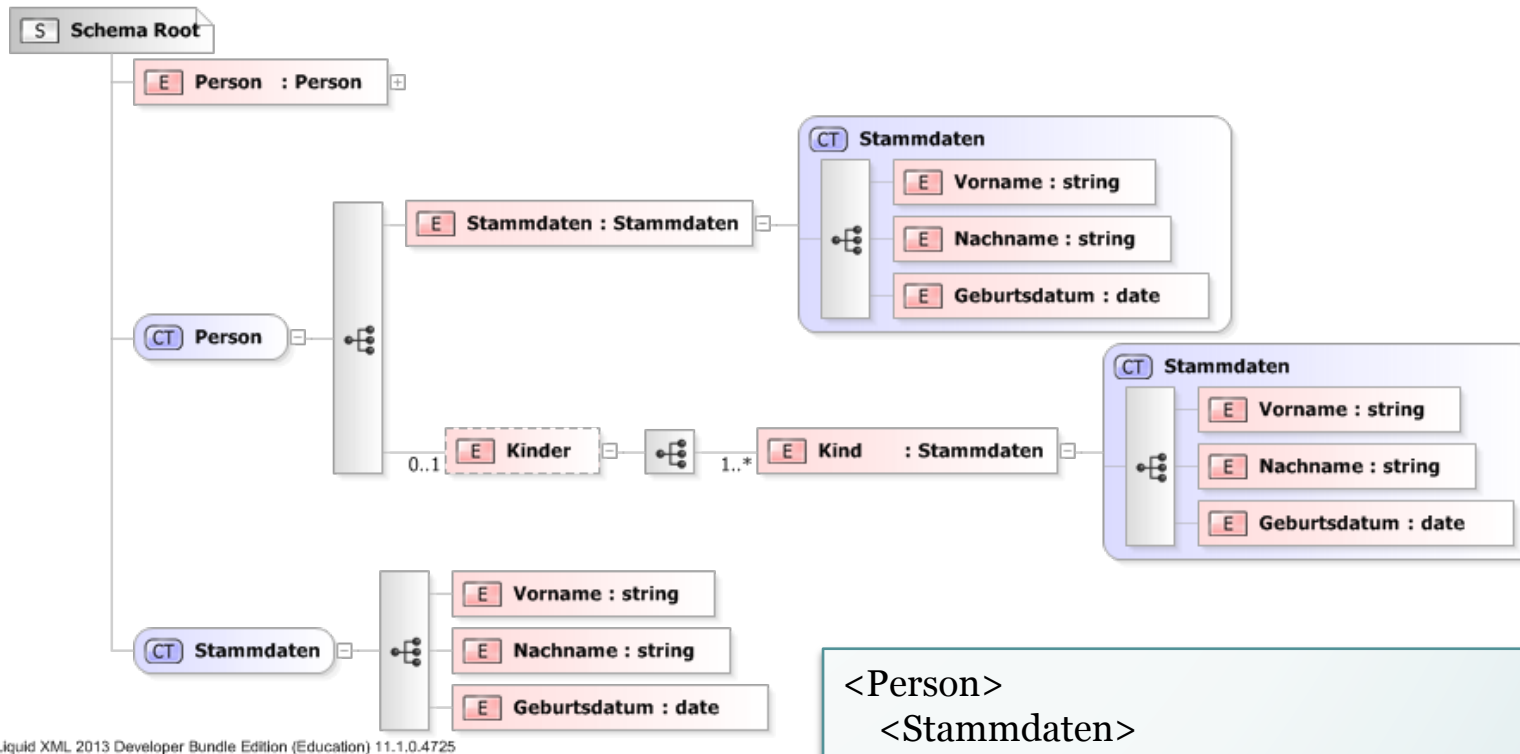
- Wenn im Schema Elemente eines anderen Schemas genutzt werden sollen, muss dieses vorher bekannt gemacht werden
 - **Include:** Elemente aus gleichem Namespace
 - **Import:** Elemente aus anderem Namespace
- **Bsp.:**

```
<xs:include schemaLocation="anderesSchema.xsd"/>
```

```
<xs:import namespace="http://markusberg.de/schema"  
schemaLocation="anderesSchema.xsd"/>
```

Validität (Gültigkeit)

- Wohlgeformtheit und Gültigkeit: Prüfung der Syntax
- Wohlgeformtheit: XML-Syntaxregeln (lexikalische Korrektheit) eingehalten, z.B.
 - Keine verbotenen Zeichen
 - Tags korrekt benutzt
 - Korrekte Schachtelung der Elemente
- Validität prüft die Gültigkeit des Dokumentes, d.h. ob es den im Schema aufgestellten Regeln entspricht, d.h. es werden Syntaxregeln der definierten Sprache (strukturelle Korrektheit) kontrolliert
 - Typen bekannt
 - Werte von Elementen entsprechen dem definierten Typ
 - Reihenfolge der Elemente
- Jedes valide Dokument ist wohlgeformt, aber nicht jedes wohlgeformte Dokument ist valide



Liquid XML 2013 Developer Bundle Edition (Education) 11.1.0.4725

- Wohlgeformt
- Nicht gültig
 - Fehlendes Element „Vorname“
 - Falscher Inhalt „Geburtsdatum“

```

<Person>
  <Stammdaten>
    <Vorname>Peter</Vorname>
    <Nachname>Müller</Nachname>
    <Geburtsdatum>1971-08-15</Geburtsdatum>
  </Stammdaten>
  <Kinder>
    <Kind>
      <Nachname>Müller</Nachname>
      <Geburtsdatum>2002-09</Geburtsdatum>
    </Kind>
  </Kinder>
</Person>

```

Vorteile von XML

- Mit jedem Texteditor bearbeitbar
- Lesbar für Menschen
- Automatisiert verarbeitbar
 - Parsing
- Prüfung auf Einhaltung der Strukturregeln möglich
 - Validierung
- Textbasiert und somit plattform-,
programmiersprachen- und betriebssystemunabhängig

Nachteile von XML

- Verglichen mit binären Dateiformaten mehr Speicherplatz nötig
- Auch visuell platzverschwendend und aufwändig zu schreiben
- Alternative: JSON
 - kompakter & direkt ausführbar in JavaScript, da es sich um JavaScript-Syntax (Arrays etc.) handelt
 - Ebenso leicht lesbar
 - Aber: keine Schemas, nicht erweiterbar, Ziel ist nicht Erstellung von Sprachen sondern Datenserialisierung

```
<employees>
  <employee>
    <firstName>John</firstName>
    <lastName>Doe</lastName>
  </employee>
  <employee>
    <firstName>Anna</firstName>
    <lastName>Smith</lastName>
  </employee>
  <employee>
    <firstName>Peter</firstName>
    <lastName>Jones</lastName>
  </employee>
</employees>
```

```
{"employees": [
  {"firstName": "John", "lastName": "Doe"},
  {"firstName": "Anna", "lastName": "Smith"},
  {"firstName": "Peter", "lastName": "Jones"}
]}
```

Editoren

- „Echte“ XML-IDEs
 - Altova XML Spy
 - Liquid XML Studio
(bis 2009 Free Community Edition)
 - Oxygen XML Editor
- Plugins für Eclipse, Netbeans, ...
 - Meist veraltet
- Texteditoren
 - Notepad++

Quiz

A decorative graphic consisting of a solid teal horizontal bar that spans the width of the page. Below this bar, on the right side, there are several horizontal lines of varying lengths and colors, including teal and white, creating a layered, modern look.



Welcher Begriff beschreibt das Einhalten der XML-Syntaxregeln?

A**XSD (XML Schema)****B****Wohlgeformtheit****C****Validität****D****DTD (Document Type Definition)****E****Parsing**



**Kann eine Liste von Personen als
Attribut abgebildet werden?**

A

Ja

B

Nein



Wodurch wird eine beliebige Menge
an Elementen angebeben
(maxOccurence)

A

max

B

C

unbounded



Ist HTML XML-konform?

A

Ja

B

Nein



**Ist ein wohlgeformtes Dokument
immer valide?**

A

Ja

B

Nein



Wozu dienen Namespaces?

A**Zum Definieren von Präfixen****B****Zum eindeutigen Identifizieren von Elementen aus verschiedenen Schemas****C****Um anzugeben, ob Leerzeichen in Elementnamen erlaubt sind**

Teil II

XPath

XSLT

Parsing

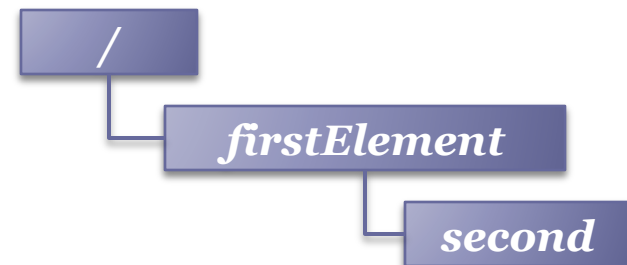
Suche per XPath

- **Problem:**
 - Komplexes XML-Dokument
- **Aufgabe:**
 - Ermitteln von Daten, die bestimmten Bedingungen genügen
 - z.B. die Adressen aller Mitarbeiter, die seit 10 Jahren hier arbeiten, sollen für das Verschicken einer Grußkarte ermittelt werden
 - Ähnl. SQL für Datenbanken
- **Lösung:**
 - XPath

XPath: Knoten

- Erinnerung: XML ist eine Baumstruktur
- Der Baum besteht aus Knoten (nodes)
 - Elemente, Attribute, Text
 - (Namespace, Kommentar, Processing Instruction, Dokument)
- Die XPath-Wurzel ist jedoch das **Dokument** selbst und nicht das erste XML-**Element**
- Das erste Element des XML-Dokumentes liegt also unterhalb des Dokumentknotens
- Wurzel wird über „/“ referenziert
- Somit muss zum Zugriff auf das Wurzelement des XML-Dokumentes trotzdem der Name explizit unterhalb von „/“ angegeben werden

```
<firstElement>  
  <second>hello</second>  
</firstElement>
```



Xpath: Pfadangaben

```
<firstElement>  
  <second>hello</second>  
</firstElement>
```

```
/firstElement
```

liefert

```
<firstElement>  
  <second>hello</second>  
</firstElement>
```

```
/firstElement/second
```

liefert

```
<second>hello</second>
```

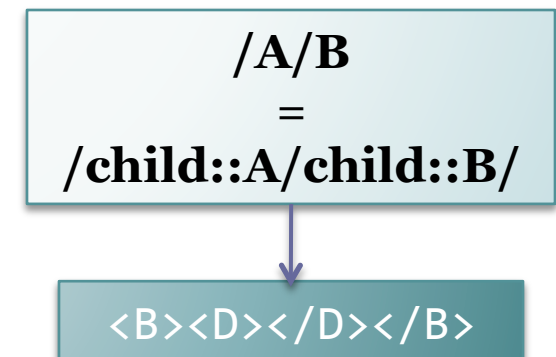
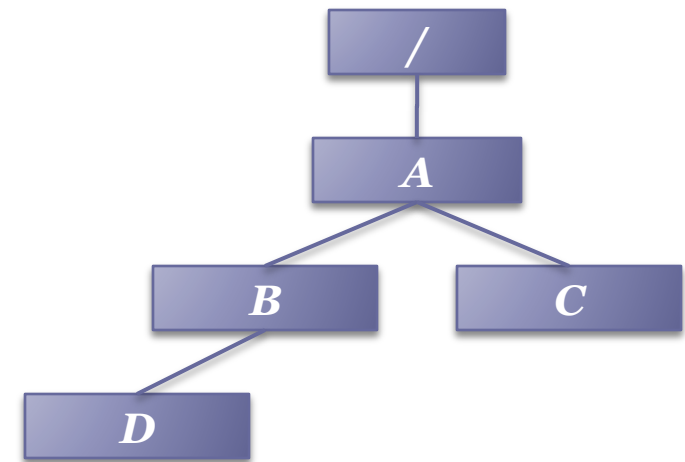
- Wir können erkennen: XPath Angaben sehen aus wie Pfadangaben oder URLs
 - Wir navigieren uns schrittweise durch den Baum
- Über die Pfadausdrücke werden Teile des Baumes selektiert, d.h. Knoten oder Knotenmengen (node, node sets)

Pfadangaben

- Es gibt relative und absolute Pfadausdrücke
- Wenn der Ausdruck mit dem Wurzelement beginnt, handelt es sich um einen absoluten Pfad
 - z.B. `/firstElement/second`
- Relative Pfade beginnen mit „//“
 - Es werden alle entsprechenden Elemente unabhängig von ihrer Position (bzw. ihren Vorgängern) im Dokument gefunden
 - z.B. `//second`
- „.“ bezeichnet den aktuellen Knoten

XPath: Achsen

- Achsen selektieren Knoten relativ zum jeweiligen Kontextknoten
 - Beschreiben somit die Relation zwischen den Knoten:
 - Kontextknoten = vorheriger Teil des Pfades, z.B. /
 - Aktueller Lokalisierungsschritt, z.B. A
 - Achse: z.B. child
 - /child::A
 - Kontextknoten /
 - A in der Rolle Kind in Bezug auf den Kontextknoten
 - D.h. alle Elemente unterhalb vom Kontextknoten / in der Rolle Kind und mit dem Namen „A“
 - /A/child:B
 - Kontextknoten A
 - alle Kinder von A mit dem Namen B
 - Keine Angabe, dann implizit „child“



Weitere Achsen

- following-sibling

```
//B/following-sibling::*
```

```
<D></D><E></E>
```

- descendant

- Tiefere Hierarchie

- ancestor

- Höhere Hierarchie

```
//C/ancestor::B
```

```
<B><C id=„2“></C></B>
```

- parent

- Eltern (d.h. direkter Ancestor)

- Attribute

- abgekürzt über „@“

```
/A/B/C/attribute::id
```

```
/A/B/C/@id
```

```
2
```

- ...

```
<A>
  <B>
    <C id=„2“></C>
  </B>
  <D></D>
  <E></E>
</A>
```

Pfadangaben

- Bestehen aus einem oder mehreren Lokalisierungsschritten (= Pfadelementen, getrennt durch „/“)

```
/child::A/B[@id=„2“]/C
```

- Das Ergebnis eines jeden Schrittes ist der Kontextknoten für den nächsten Schritt
- Pfade bestehen aus
 - Achse
 - Knotentest
 - Prädikate

```
achse::Knoten[Prädikate]
```

```
child::A[@id=„3“]
```

Prädikate

- Schränken das durch Knotenname und Achse beschriebene Nodeset weiter ein
 - Suchanfrage wird konkretisiert
 - z.B. nur Knoten mit bestimmten Attributwerten
 - z.B. Knoten an bestimmter Position
 - Index startet mit 1 (statt 0)
- Prädikate werden in eckigen Klammern angegeben

```
<A>  
  <B cat=„u“>hallo</B>  
  <B cat=„w“>wie</B>  
  <B cat=„u“>geht's</B>  
</A>
```

```
/A/B[@cat=„u“]
```

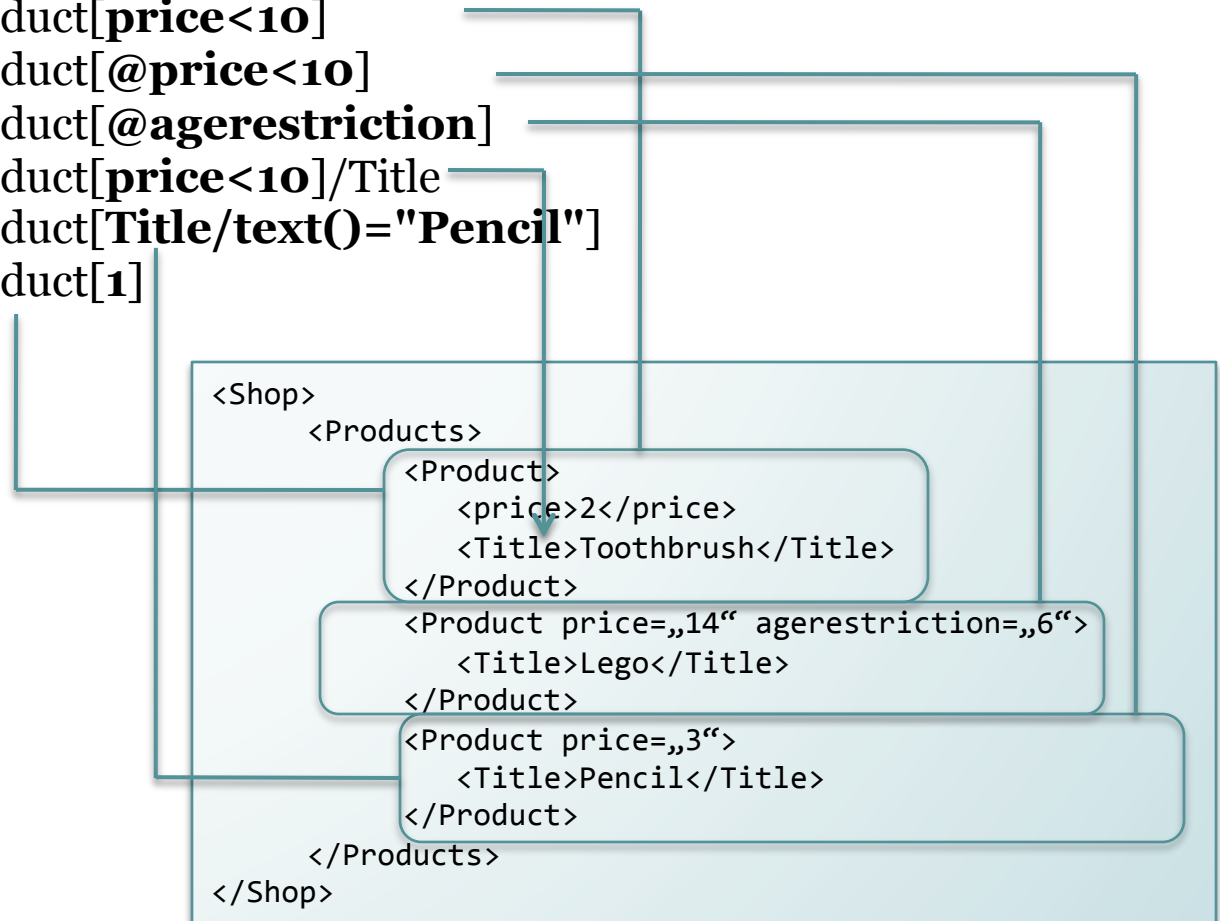
```
<B cat=„u“>hallo</B>  
<B cat=„u“>geht's</B>
```

```
/A/B[2]
```

```
<B cat=„w“>wie</B>
```


Prädikate

- /Shop/Products/Product[**price<10**]
- /Shop/Products/Product[**@price<10**]
- /Shop/Products/Product[**@agerestriction**]
- /Shop/Products/Product[**price<10**]/Title
- /Shop/Products/Product[**Title/text()="Pencil"**]
- /Shop/Products/Product[**1**]



Prädikate kombinieren

- Es können über bool'sche Operatoren auch mehrere Prädikate miteinander kombiniert werden
 - `and`
 - `or`
- z.B.

```
/Shop/Products/Product[@price<=10 and  
Title/text()=„Pencil“]
```

Namespaces berücksichtigen!

- Knoten müssen immer voll qualifiziert werden, d.h. Namespaces müssen mit angegeben werden

```
/Shop/mb:Products/mb:Product
```

```
/Shop/child::mb:Products/child::mb:Product[@price>11]
```

```
<Shop xmlns:mb=„http://mberg.net/example“>
  <mb:Products>
    <mb:Product>
      <price>2</price>
      <Title>Toothbrush</Title>
    </mb:Product>
    <mb:Product price=„14“ agerestriction=„6“>
      <Title>Lego</Title>
    </mb:Product>
    <mb:Product price=„3“>
      <Title>Pencil</Title>
    </mb:Product>
  </mb:Products>
</Shop>
```

Funktionen

- Können in Prädikaten verwendet werden
- Eine Funktion bereits kennen gelernt
 - `text()` → Textinhalt eines Knotens ermitteln
- Numeric
 - `round(number)`
 - `abs(number)`
- String
 - `concat(string1, string2, ...)`
 - `substring(string, start)` bzw. `substring(string, start, length)`
 - `string-length(string)`
 - `upper-case(string)`
 - `starts-with(string)`
- Sonstiges
 - `local-name(node)` → Name ohne Namespace
 - `count()` → Anzahl der Elemente
 - `boolean()` → gibt true oder false zurück, z.B. ob ein Knoten existiert
 - `not()`
 - `position()` → Position des aktuell verarbeiteten Knotens in einer Liste
 - `last()` → wählt das letzte Element in einer Liste aus

Tipps

- „*“ bezeichnet einen beliebigen Knoten
 - Kann über Prädikate eingeschränkt werden

```
/Shop/Products/*[@price<=10]
```

- Funktionen können Literale, Pfadausdrücke oder auch eine Referenz auf den aktuellen Knoten entgegennehmen

```
string-length(„hallo“)  
string-length(/Shop/Products/Product[1]/Title)  
string-length(.)
```

Übung: XPath-Aufgaben

- Zeit, das Erlernte anzuwenden und zu testen...
- <http://learn.onion.net/language=de/taps=9075/2902>
- Sie müssen nicht alle Aufgaben lösen
 - Die ersten 10 sind aber unbedingt empfohlen
- Fangen Sie vorne an, die Schwierigkeit steigt mit den Aufgaben

XSLT

- **Verschiedene Anwendungen**
 - strukturieren Daten unterschiedlich
 - setzen auf unterschiedliche Sprachen
- **Problem:**
 - Daten auf Grundlage verschiedener Schemas
- **Ziel:**
 - Konvertieren von Daten
 - XML → XML
 - XML → CSV
 - u.a. Visualisieren von Daten (z.B. XML → HTML)

Beispiel: XML zu XML

```
<Person mitarbeiterID=„007“>  
  <Vorname>James</Vorname>  
  <Nachname>Bond</Nachname>  
</Person>
```

?

```
<Mitarbeiter>  
  <ID>007</ID>  
  <Namen>  
    <Vorname>James</Vorname>  
    <Nachname>Bond</Nachname>  
  </Namen>  
</Mitarbeiter>
```


Beispiel: XML zu HTML

```
<Person mitarbeiterID=„007“>  
  <Vorname>James</Vorname>  
  <Nachname>Bond</Nachname>  
</Person>
```

?

```
<html>  
  <body>  
    <table>  
      <tr>  
        <td>Bond</td>  
        <td>James</td>  
        <td>007</td>  
      </tr>  
    </table>  
  </body>  
</html>
```

Lösung

- XSLT: XSL Transformation
 - Teil der XSL eXtensible Stylesheet Language
 - Dokumente somit ebenfalls XML
 - Dateiendung meist .xsl
 - Namespace: <http://www.w3.org/1999/XSL/>
- Umwandeln von XML-Dokumenten anhand von Regeln, die in XSLT definiert werden
 - Ziel meist XML
 - Aber auch andere textbasierte Formate möglich
- Lokalisierung von Elementen über XPath („Pattern“)

Prinzip

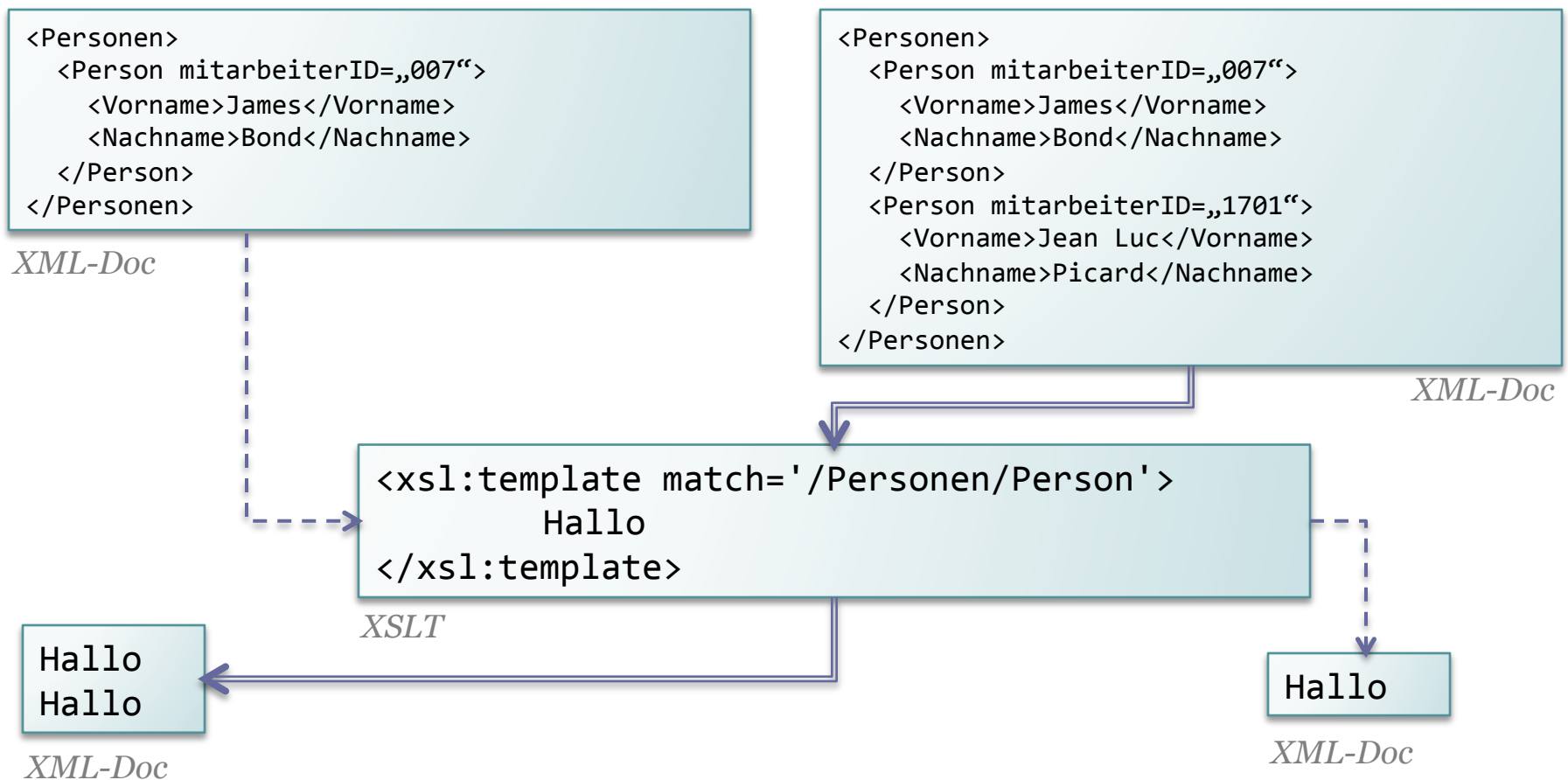
- Ein XSLT Dokument besteht aus selbstdefinierten Transformationsregeln auf Grundlage von Templates bzw. Matchern
 - „Wenn das spezifizierte Element gefunden wird, erzeuge folgende Ausgabe“
- Matcher wird über XPath-Ausdrücke definiert
 - z.B. /Person
 - Wird einem Template als Attribut übergeben

```
<xsl:template match='/Person' >
```

- Anschließend wird angegeben welchen Text das Template produzieren soll, falls ein dem XPath entsprechendes Element gefunden wird

```
<xsl:template match='/Person' >  
    Hallo  
</xsl:template>
```

Mini-Beispiel



Templates

- Die Ausgabe von statischem Text ist für eine Transformation nicht ausreichend
- Oft soll der Inhalt beibehalten werden und nur die Struktur des Dokumentes ändert sich
- Zugriff auf originale Werte erforderlich: `value-of`

```
<xsl:template match='/Personen/Person' >  
  <xsl:value-of select='./Vorname' >  
</xsl:template>
```

*Bezieht sich auf
„gematchten“ Knoten
(Kontextknoten)*

→ `/Personen/Person/Vorname`

Mini-Beispiel II

```
<Personen>
  <Person mitarbeiterID=„007“>
    <Vorname>James</Vorname>
    <Nachname>Bond</Nachname>
  </Person>
  <Person mitarbeiterID=„1701“>
    <Vorname>Jean Luc</Vorname>
    <Nachname>Picard</Nachname>
  </Person>
</Personen>
```

XML-Doc

```
<xsl:template match='/Personen/Person'>
  <xsl:value-of select='./Vorname'>
</xsl:template>
```

XSLT

```
James
Jean Luc
```

XML-Doc

Templates

- Bis jetzt: Text
- Ziel: XML
 - Alles im Template wird ausgegeben
 - Einfach XML-Tags hinzufügen

```
<xsl:template match='/Personen/Person'>  
  <firstName><xsl:value-of select='./Vorname'></firstName>  
</xsl:template>
```

```
<firstName>James</firstName>  
<firstName>Jean Luc</firstName>
```

Templates

- Jedes XSLT muss einen **Root-Matcher** besitzen (wenn nicht, greift eine Standardregel, die lediglich den Text aller Nodes konkateniert)

```
<xsl:template match='/'>
```

- Dies ist der Einstiegspunkt für den XSLT-Parser
- Darüber hinaus kann es weitere Templates mit anderen Matchern geben
- Das Suchen nach weiteren Templates, muss (z.B. aus dem Root-Template) aktiv aufgerufen werden

```
<xsl:template match='/'>  
  <xsl:apply-templates/>  
</xsl:template>
```

- Wenn mehrere Templates „matchen“, wird das spezifischste genutzt

Templates


- Matcher müssen keine absoluten Pfadangaben benutzen
- Als Kontextknoten wird immer derjenige genommen, der das `apply-templates` ausgelöst hat, also der gefundene Knoten des vorherigen Templates

```
<xsl:template match='/'>
  <Result><xsl:apply-templates/></Result>
</xsl:template>

<xsl:template match='Person'>
  <Mitarbeiter>
    <Name><xsl:value-of select='Nachname' /></Name>
    <xsl:apply-templates/>
  </Mitarbeiter>
</xsl:template>

<xsl:template match='Adresse'>
  <Stadt><xsl:value-of select='stadt' /></Stadt>
</xsl:template>
```

```
<Personen>
  <Person>
    <Nachname>Bond</Nachname>
    <Vorname>Jamea</Vorname>
    <Adresse>
      <Stadt>London</Stadt>
      <Land>England</Land>
    </Adresse>
  </Person>
</Personen>
```



```
<Result>
  <Mitarbeiter>
    <Name>Bond</Name>
    <Stadt>London</Stadt>
  </Mitarbeiter>
</Result>
```

Benannte Templates

- Werden nicht über einen Matcher sondern über einen Namen aufgerufen

```
<xsl:template name='meinTemplate'>  
...  
</xsl:template>
```

- Aufruf über:

```
<xsl:call-template name='meinTemplate' />
```

Parametrisierte Templates

- Benannte Templates können parametrisiert werden
 - Zugriff auf Parameter über \$

```
<xsl:template name='meinTemplate'>
  <xsl:param name='meinParameter'>Defaultwert</xsl:param>
  ...
  <meinElement>
    <xsl:value-of select="$meinParameter"/>
  </meinElement>
</xsl:template>
```

- Aufruf:

```
<xsl:call-template name='meinTemplate'>
  <xsl:with-param name='meinParameter' select='/meinPfad' />
</xsl:call-template>
```

select="'meinWert'"

Variablen

- Definieren

```
<xsl:variable name="anzahlPersonen" select="count(/Personen/Person) " />
```

- Auslesen

- wie bei Parametern über \$

```
<xsl:value-of select='$anzahlPersonen' />
```

Fallunterscheidungen

- If
 - Testet ob eine über einen XPath-Ausdruck mit booleschem Ergebnis definierte Bedingung erfüllt ist und führt alle Befehle in den Kindelementen aus

```
<xsl:if test='boolean-expression'>  
  <!-- ausführen wenn true-->  
</xsl:if>
```

- Keine „else“ und keine Möglichkeit mehrere Fälle anzugeben
- Choose
 - when (mehrfach)
 - otherwise

```
<xsl:choose>  
  <xsl:when test='boolean-expression'>  
    <!-- ausführen wenn true-->  
  </xsl:when>  
  <xsl:when test='boolean-expression'>  
    <!-- ausführen wenn true-->  
  </xsl:when>  
  <xsl:otherwise>  
    <!-- wenn keine Bedingung zutrifft -->  
  </xsl:otherwise>  
</xsl:choose>
```

Schleifen

- For-each
 - Jedes Element einer Liste als Kontextknoten verwenden und folgende Anweisungen ausführen
 - Angabe eines XPath-Ausdrucks, der mehrere gleiche Knoten selektiert, z.B. mehrere person-Elemente in einer personen-Sequence

```
<xsl:for-each select='/personen/person'>  
  <xsl:value-of select='name' />  
</xsl:for-each>
```

/personen/person/name
bzw.
./name

Copy & Copy-of

- Kopieren von Elementen an die Stelle, wo die Copy-Anweisung steht
- **Copy:**
 - Kopiert den aktuellen Knoten (ohne Attribute und Kinder, d.h. ohne Textknoten, d.h. ohne Inhalt)

```
<xsl:copy/>
```

- Aktueller Knoten definiert z.B. über Matcher oder Schleife
- Inhalt des Elements kann angegeben werden

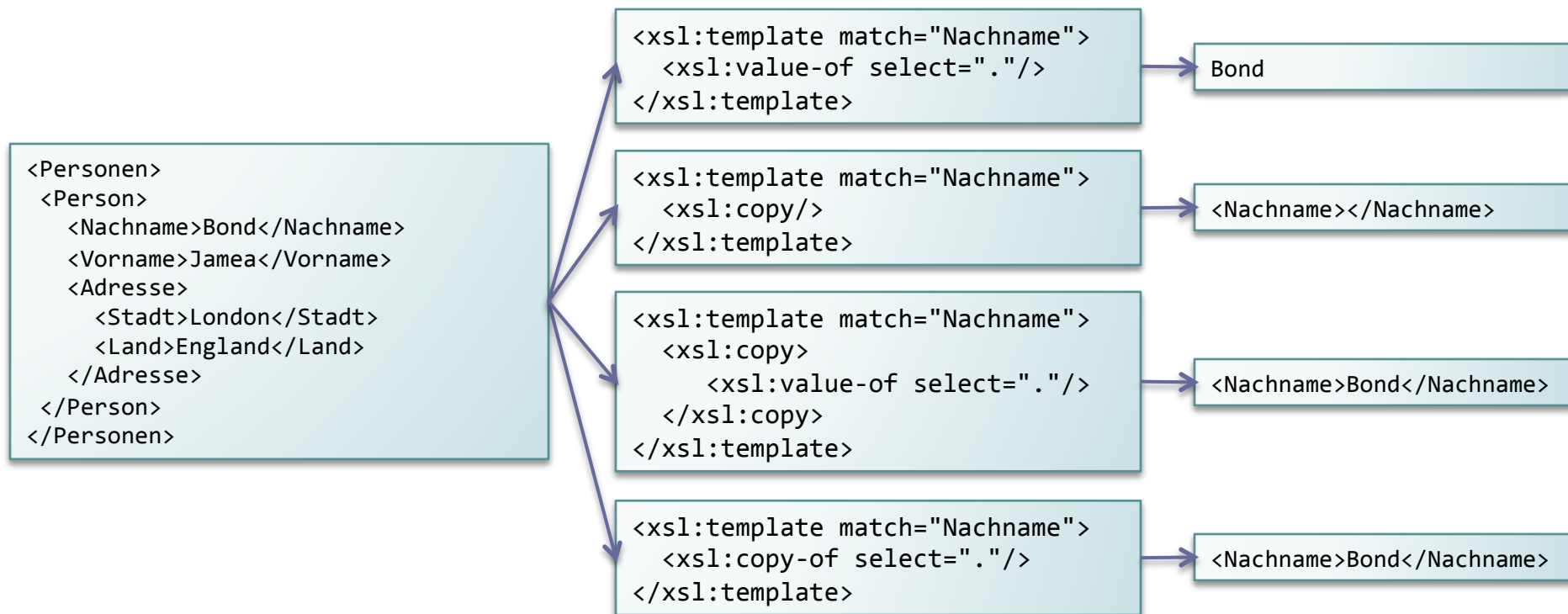
```
<xsl:template match="person">  
  <xsl:copy>  
    <xsl:value-of select="."/>  
  </xsl:copy>  
</xsl:template>
```

- **Copy-of:**
 - Zu kopierende Elemente (inkl. Kindern und Attributen) werden über select ausgewählt

```
<xsl:copy-of select="/Personen/person"/>
```

Copy-of vs. value-of

- Value-of kopiert den Inhalt eines Knotens
- Copy kopiert den Knoten selbst (ohne Inhalt)
- Copy-of kopiert den Knoten mit Inhalt



Einstellungen

- Ausgabemethode
 - `method = xml | html | text`
- Encoding
 - `encoding`
 - z.B. `utf-8`
- Einrückung
 - `indent = yes | no`
- MIME-Typ
 - `media-type`

```
<xsl:output method="xml" indent="yes"/>
```

XSL Stylesheet

- Bzw. XML zu HTML mit XSLT
- Einbinden über PI: teilt dem Interpreter mit, dass die XML-Daten mit der angegebenen XSL-Datei transformiert werden sollen, um eine entsprechend formatierte Ausgabe zu erhalten

```
<?xml-stylesheet type="text/xsl" href="filename.xsl"?>
```

- Öffnen einer XML-Datei mit Browser
 - Wenn XSL angegeben ist, wird das XML-Dokument automatisch transformiert und das Ergebnis angezeigt

XSL-FO (Formatting Objects)

- Teil von XSL (ebenso wie XSLT)
 - Somit ebenfalls zur Transformation von XML-Dokumenten
- Beschreibt das Layout von Dokumenten (Text, Linien, Bilder,...)
- Anwendung: Erstellung von Dokumenten (z.B. PDF) aus XML-Dokumenten mit Hilfe einer intermediären Sprache, die das Aussehen beschreibt und in verschiedene Zielformate überführt werden kann
- Ablauf
 - XML-Quelldaten in XSL-FO-Beschreibung überführen
 - XML zu XSL-FO (XSLT)
 - XSL-FO-Beschreibung in PDF wandeln
 - FO zu PDF (FO Prozessor, z.B. von Apache)



XSL-FO: Beispiel

- Struktur
 - Page
 - Flow
 - Block

```
<?xml version="1.0" encoding="UTF-8"?>

<fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">

  <fo:layout-master-set>
    <fo:simple-page-master master-name="myMaster">
      <fo:region-body region-name="xsl-region-body"/>
    </fo:simple-page-master>
  </fo:layout-master-set>

  <fo:page-sequence master-reference="myMaster">
    <fo:flow flow-name="xsl-region-body">
      <fo:block>Moin Welt</fo:block>
    </fo:flow>
  </fo:page-sequence>

</fo:root>
```

Bzw. region-before
für Kopfzeilen,
region-after für
Fußzeilen etc.

Parsen von XML

- Das Verarbeiten von XML erfolgt durch einen Parser
- Zwei Varianten
 - **DOM**
 - Document Object Model
 - **SAX**
 - Simple API for XML

Parsing mit DOM

- Gesamtes XML-Dokument wird zunächst eingelesen und als Baumstruktur vorgehalten
- Dokument liegt komplett im Speicher
- Vor dem Zugriff auf die Elemente wird sichergestellt, dass das Dokument wohlgeformt bzw. valide ist
- Es kann im Baum navigiert werden
- Änderungen an Elementen möglich

Parsing mit SAX

- Eventbasiert
- Während des Einlesens wird gemeldet auf welche Elemente der Parser stößt
 - Dokument wird einmal Schritt für Schritt eingelesen
 - Dabei werden Ereignisse gemeldet
 - Ich habe ein öffnendes Tag „Nachname“ gefunden
 - Ich habe Text „Bond“ gefunden
 - Ich habe ein schließendes Tag „Nachname“ gefunden
 - Danach kein Zugriff mehr möglich
- Größe des Arbeitsspeichers spielt keine Rolle, da das Dokument nicht komplett geladen wird
- Schnell und speicherschonend
- Fehler (z.B. nicht wohlgeformtes XML) fallen erst auf, wenn der Parser die fehlerhafte Stelle erreicht hat
- Hierarchieinformationen (Struktur) gehen verloren
 - Endanwendung muss diese Infos selbst verwalten
- Keine Änderungen am Dokument möglich (d.h. keine Schreiboperationen)

„XML is like violence. If it doesn't solve your problem, you're not using enough of it.“

- Autor unbekannt

Quellen und weiterführende Literatur

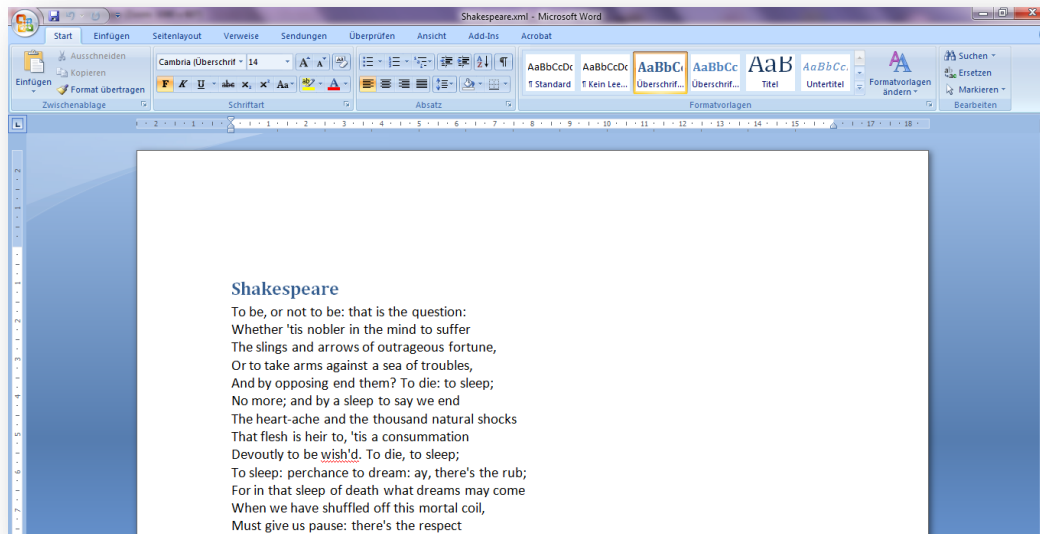
- <http://openbook.galileocomputing.de/kit/itkomp15000.htm>
- <https://www.db.informatik.uni-kassel.de/Lehre/WS0910/XML/XML2009.pdf>
- <http://www.w3schools.com/xpath/>
- <http://www.w3.org/TR/xpath>
- <http://www.w3.org/TR/xslt>
- <http://www.w3schools.com/xslfo>

Demos



Demo: .docx - Transformation

- Word-Dateien sind bereits XML (WordML)
 - Bzw. Zip-Archiv (mit mehreren XML-Dateien)
 - Speichern als eine einzige XML-Datei möglich



- Es kann direkt eine XSLT angewandt werden

XSLT: Inhalt von Überschrift1 ersetzen

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <xsl:stylesheet version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
3      xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns="http://www.w3.org/1999/xhtml"
4      xmlns:w="http://schemas.openxmlformats.org/wordprocessingml/2006/main">
5  <xsl:template match="/">
6      <xsl:apply-templates/>
7  </xsl:template>
8
9  <xsl:template match="w:t">
10     <xsl:choose>
11     <xsl:when test="parent::w:r/parent::w:p/w:pPr/w:pStyle/@w:val='berschrift1'">
12         <w:t>hallo</w:t>
13     </xsl:when>
14     <xsl:otherwise>
15         <xsl:copy-of select="."/>
16     </xsl:otherwise>
17     </xsl:choose>
18
19 </xsl:template>
20
21 <xsl:template match="node()|@*">
22     <xsl:copy>
23         <xsl:apply-templates select="node()|@*" />
24     </xsl:copy>
25 </xsl:template>
26 </xsl:stylesheet>
```

Ergebnis

