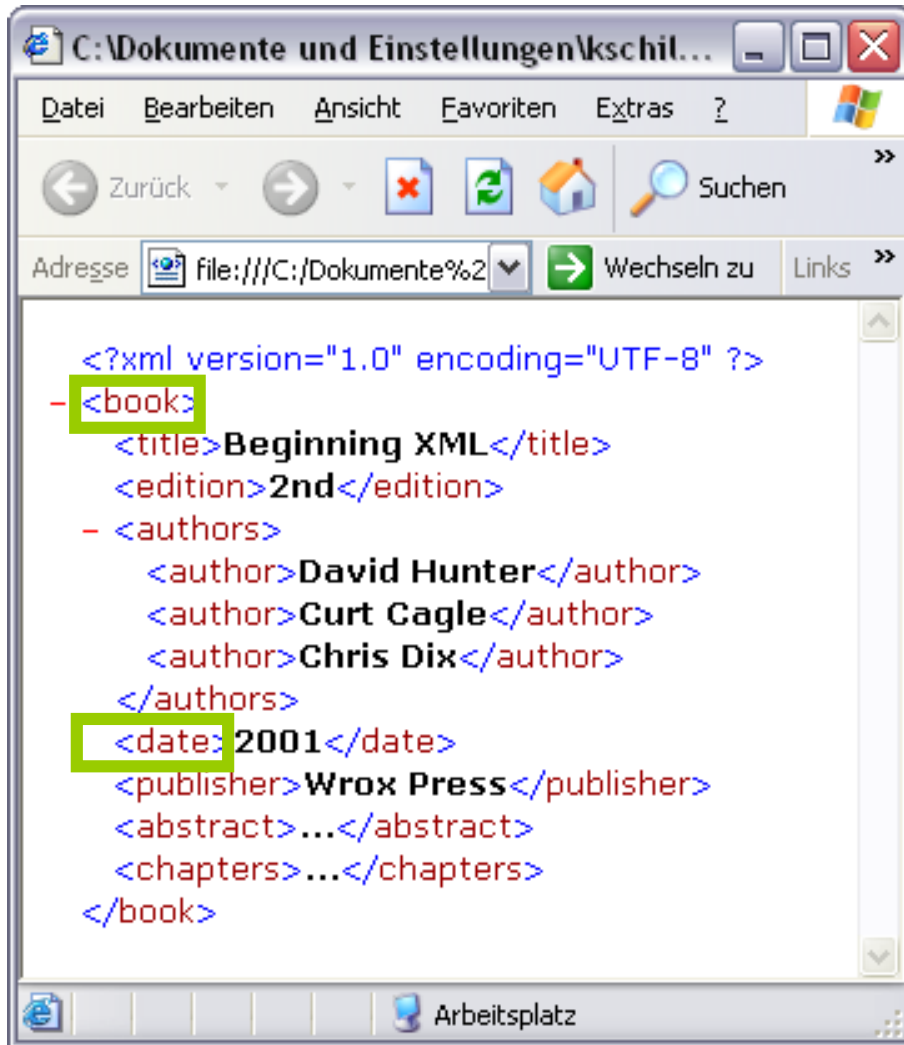




## **Vorlesung XML-Technologien – SoSe 2014 Rückblick und Klausurvorbereitung**

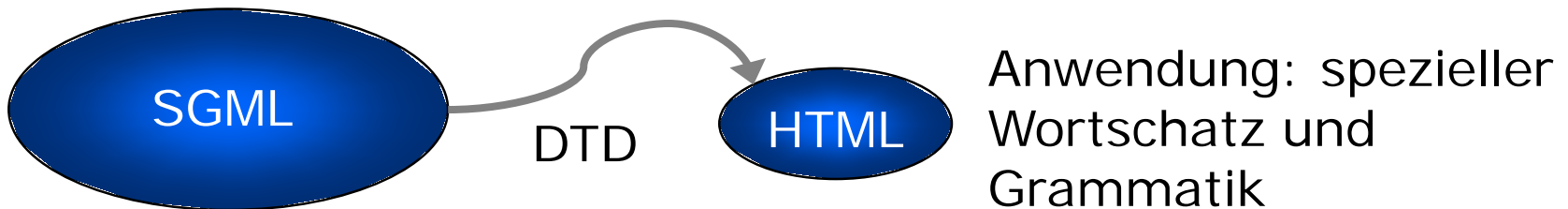
Marko Harasic  
Freie Universität Berlin  
Institut für Informatik  
Netzbasierte Informationssysteme  
[harasic@inf.fu-berlin.de](mailto:harasic@inf.fu-berlin.de)



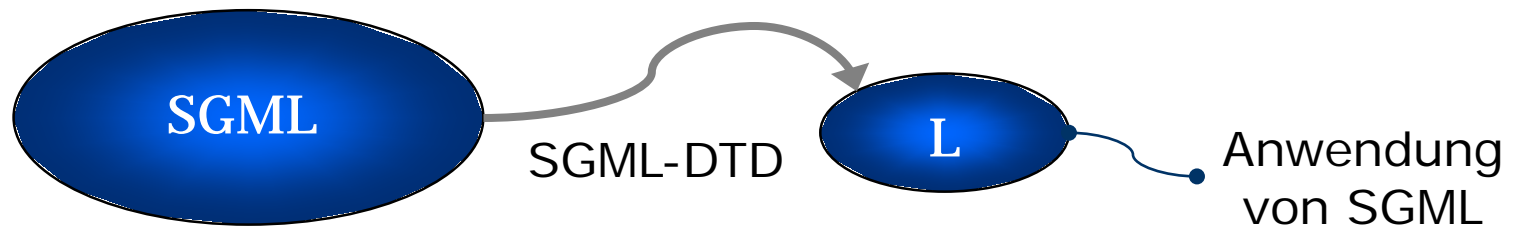
```
<?xml version="1.0" encoding="UTF-8" ?>
- <book>
  <title>Beginning XML</title>
  <edition>2nd</edition>
  - <authors>
    <author>David Hunter</author>
    <author>Curt Cagle</author>
    <author>Chris Dix</author>
  </authors>
  <date>2001</date>
  <publisher>Wrox Press</publisher>
  <abstract>...</abstract>
  <chapters>...</chapters>
</book>
```

- Extensible Markup Language
- erlaubt Strukturieren von Inhalten
- Unterschiede zu HTML:
  - Medienneutral
- Tag-Namen  
<name>...</name> beliebig
- generische  
Auszeichnungssprache

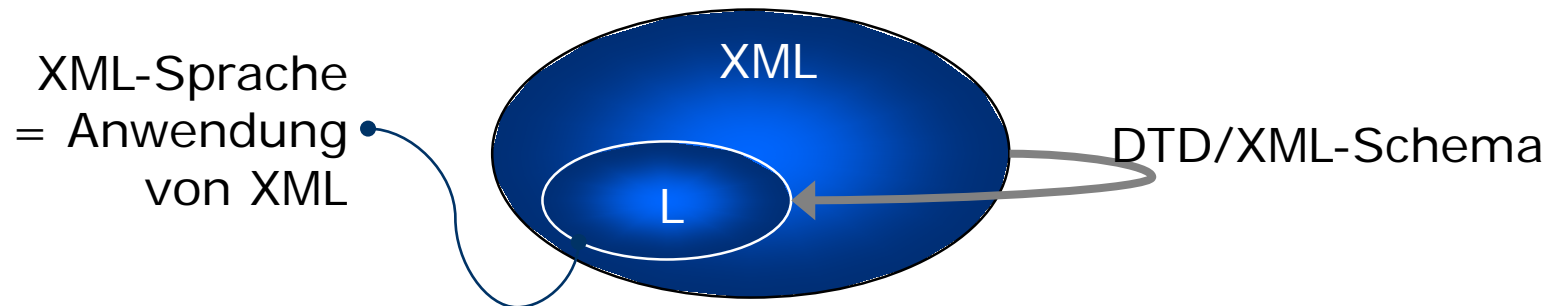
- gibt zwar keine konkreten Tags vor
- Mit Document Type Definitions (DTDs) können aber spezielle Auszeichnungssprachen mit konkreten Tags definiert werden:
  - werden Anwendungen von SGML genannt
  - bekannteste Anwendung von SGML: HTML



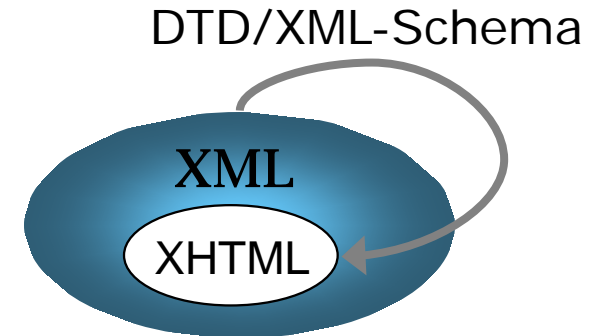
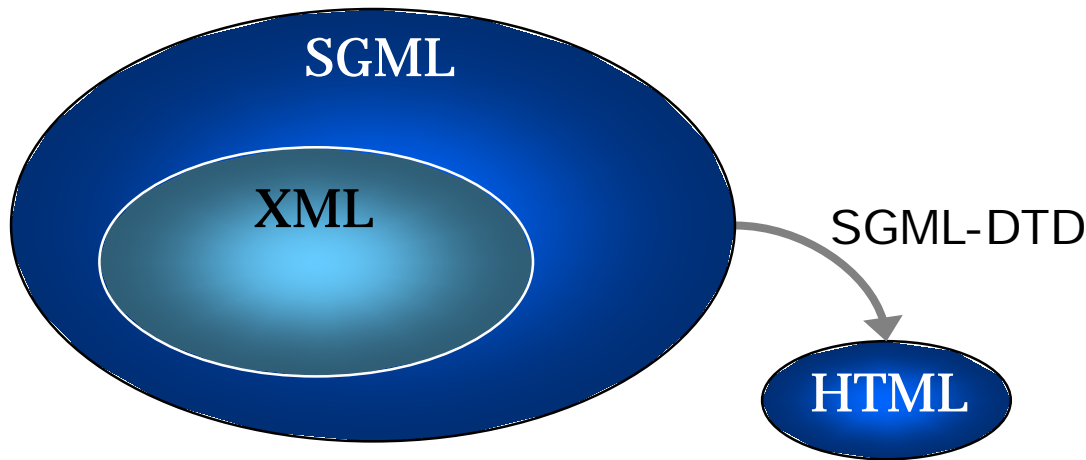
- Anwendung selbst kann keine Anwendung definieren



- L muss *nicht* Teilsprache von SGML sein.
- L kann *keine* neue Sprache definieren.
- Beispiel: HTML



- L immer Teilsprache von XML
- L kann *keine* neue Sprache definieren.
- Beispiel: XHTML



## HTML

- Anwendung von SGML

## XML

- Teilsprache von SGML

## XHTML

- XML-Sprache = Anwendung von XML
- alle XHTML-Dokumente immer wohlgeformte XML-Dokumente

# 1. Unstrukturierter Inhalt

- Beispiel: `<first>John</first>`
- einfacher Text ohne Kind-Elemente  
Kind-Element: Element, das im Inhalt eines Elementes vorkommt
- unstrukturierter Inhalt → *Parsed Character Data (PCDATA)*:
  - **character data**: einfache Zeichenkette
  - **parsed**: Zeichenkette wird vom Parser analysiert, um Ende-Tag zu identifizieren
  - Normalisierung: u.a. Zeilenumbruch (CR+LF) → `&#xA;`

Anmerkung: Auf den Folien schreibe ich der besseren Lesbarkeit wegen *Kind-Element* statt *Kindelement* !

## 2. Strukturierter Inhalt

- Beispiel:

```
<name>  
  <first>John</first>  
  <last>Doe</last>  
</name>
```

- Sequenz von  $> 0$  Kind-Elementen:
  - hier: `<first>John</first>` und `<last>Doe</last>`
- kein Text vor, nach oder zwischen den Kind-Elementen
- Kind-Elemente immer geordnet:  
Reihenfolge, so wie sie im XML-Dokument erscheinen
- Elemente können beliebig tief geschachtelt werden.

### 3. Gemischter Inhalt (mixed content)

- enthält Text mit mindestens einem Kind-Element
- Beispiel:

```
<section>  
Text  
<subsection> ... </subsection>  
Text  
</section>
```



- Beispiel:

```
<name>
  <first>John</first>
  <middle></middle>
  <last>Doe</last>
</name>
```

- weder Text noch Kind-Element
- `<middle></middle>` auch **leeres Element** genannt
- Abkürzung: **selbstschießendes Tag** (engl.: *self-closing tag*) `<middle/>` :

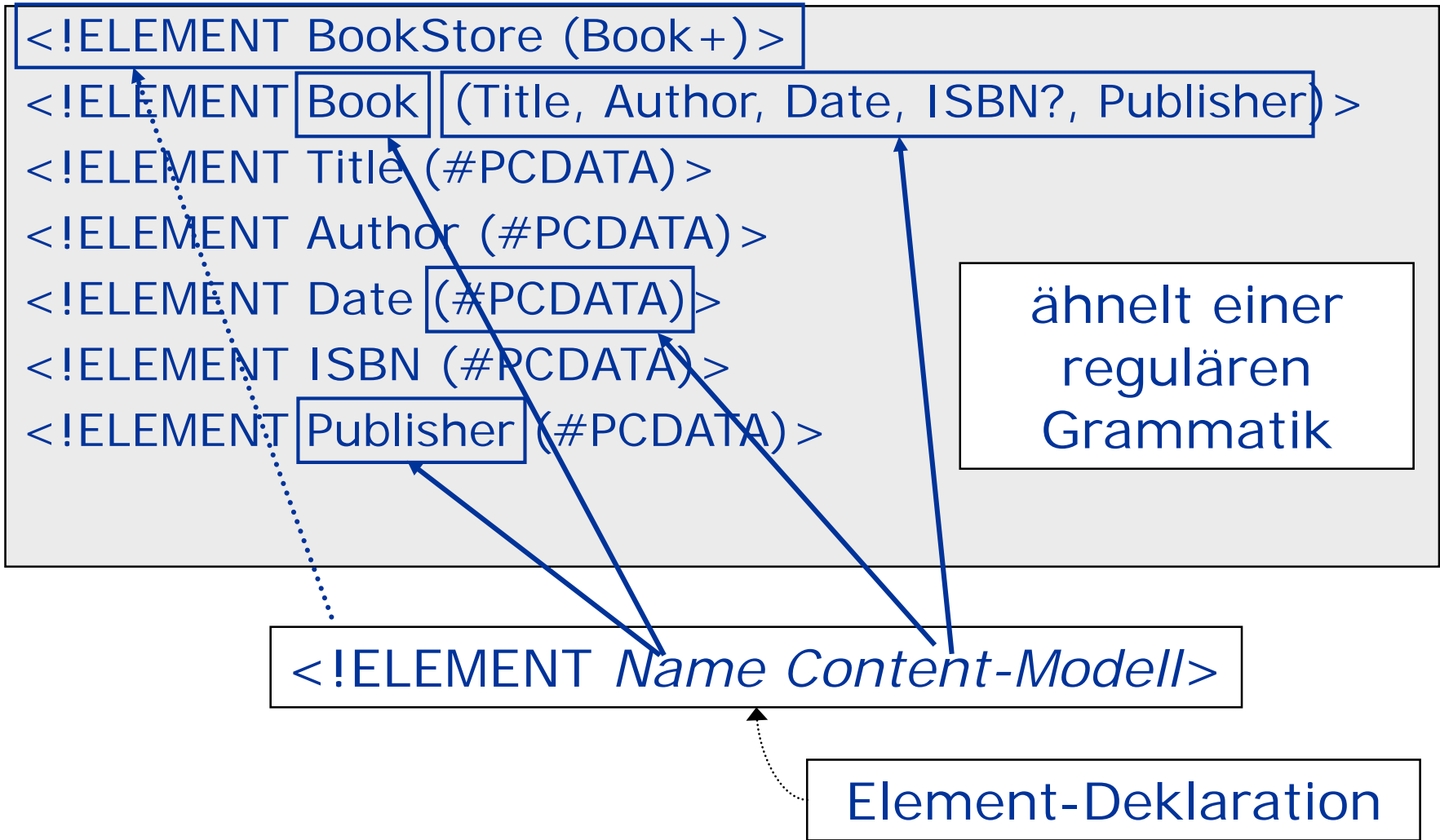
```
<name>
  <first>John</first>
  <middle/>
  <last>Doe</last>
</name>
```

```
<name id="1232345" nickname="Shiny John">  
  <first>John</first>  
  <last>Doe</last>  
</name>
```

- **Attribut: Name-Wert-Paar**
  - name="wert" oder name='wert' aber ~~name="wert'~~
- **Attribut-Wert:**
  - immer PCDATA: keine Kind-Elemente, kein < und &
  - "we"rt" und 'we'rt' ebenfalls nicht erlaubt
  - Normalisierung: u.a. Zeilenumbruch → &#xA;
- Beachte: Reihenfolge der Attribute belanglos

1. Jedes Anfangs-Tag muss ein zugehöriges Ende-Tag haben.
2. Elemente dürfen sich nicht überlappen.
3. XML-Dokumente haben genau ein Wurzel-Element.
4. Element-Namen müssen bestimmten Namenskonventionen entsprechen.
5. XML beachtet grundsätzlich Groß- und Kleinschreibung.
6. XML belässt White Space im Text.
7. Ein Element darf niemals zwei Attribute mit dem selben Namen haben.

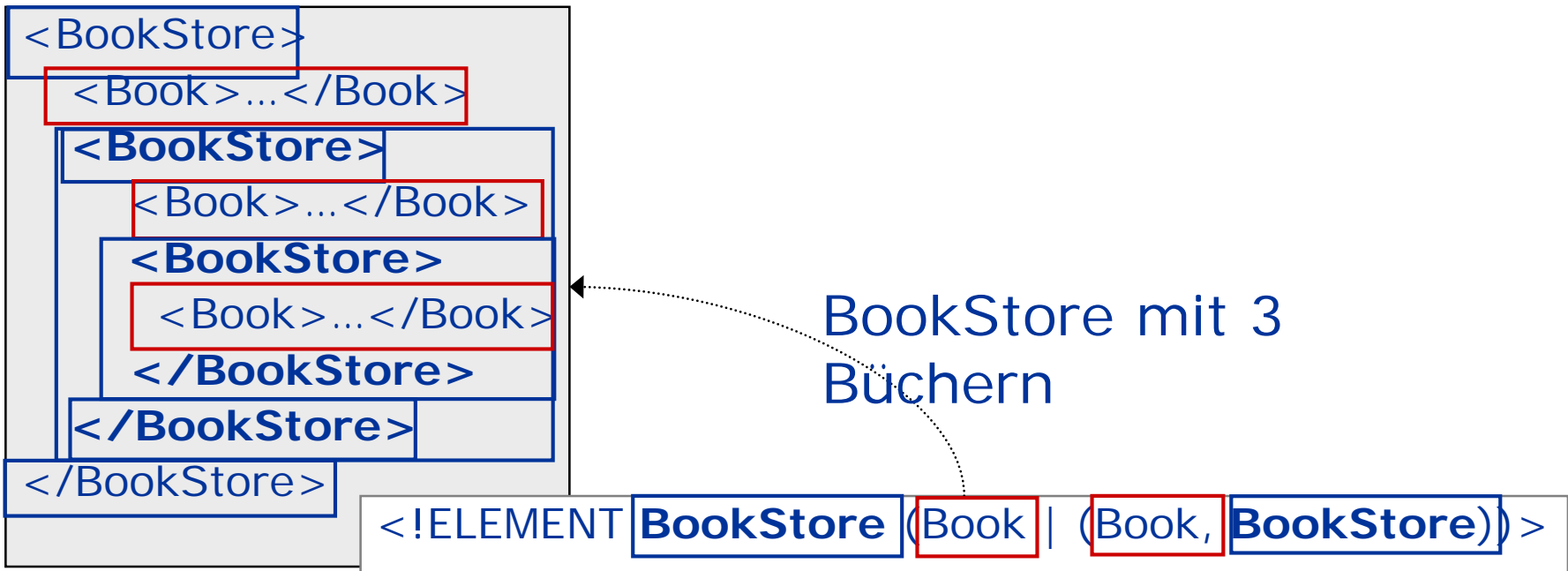
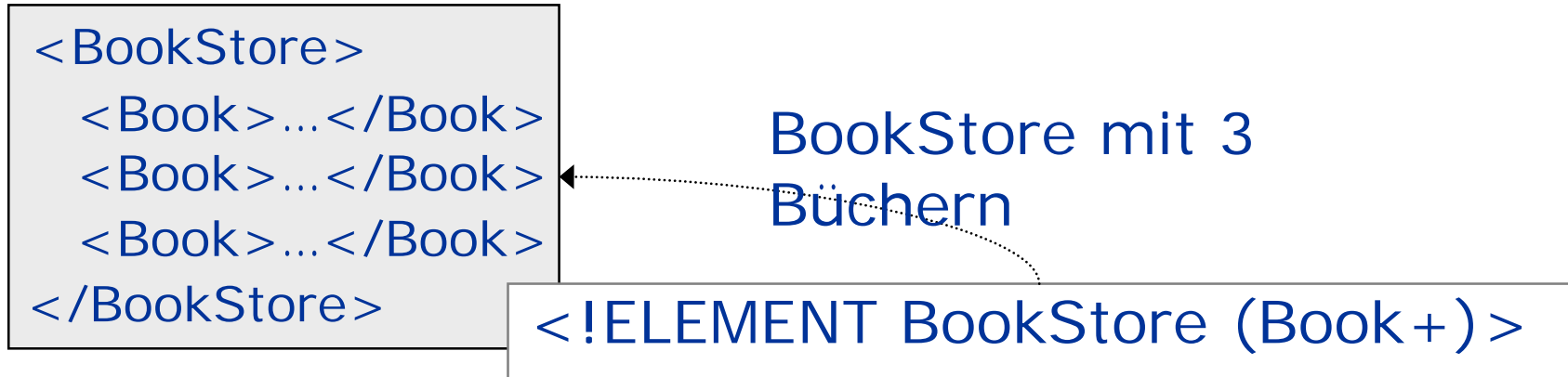
# Beispiel DTD



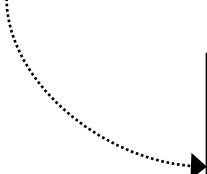
# Elementdeklaration: Datentypen für Inhalte

- **Element:**  
Ausdruck über Elemente mit Symbolen , + \* | ?
- **#PCDATA:**  
unstrukturierter Inhalt ohne reservierte Symbole (<, &)  
<!ELEMENT Title (#PCDATA)>
- **EMPTY:**  
leerer Inhalt, Element kann Attribute haben  
<!ELEMENT hr EMPTY>: <hr height="3"/>
- **ANY:**  
beliebiger Inhalt (strukturiert, unstrukturiert, gemischt oder leer)  
<!ELEMENT Absatz ANY>
- Keine gewohnten Datentypen wie INTEGER oder FLOAT

# Rekursive vs. iterative Deklaration



```
<!ATTLIST BookStore  
    version CDATA #IMPLIED>
```



```
<BookStore version="1.0">  
    ...  
</BookStore>
```

- BookStore hat Attribut version
- Außer version hat BookStore keine weiteren Attribute
- **CDATA**: Attribut-Wert = String ohne <, &, ', "
- Beachte: nicht verwechseln mit <![CDATA[ ... ]]>
- daher Entity References für <, & und ' bzw. " verwenden

```
<!ATTLIST Author  
          gender (male | female) "female" >
```

- hier statt CDATA **Aufzählungstyp**:
- Attribut gender hat entweder Wert male oder female.
- "female" ist Standard-Wert von gender.



```
<!ATTLIST BookStore version CDATA #FIXED "1.0" >
```

- **#FIXED**: Attribut hat immer den gleichen Wert
- **#IMPLIED**: Attribut optional
- **#REQUIRED**: Attribut obligatorisch

- wohlgeformt (well formed):

XML-Dokument entspricht Syntaxregeln von XML

- zulässig (valid) bzgl. einer DTD:
  1. Wurzel-Element des XML-Dokumentes in DTD deklariert
  2. Wurzel-Element hat die in der DTD festgelegte Struktur

- XML-Sprache statt eigener Syntax
- Vielzahl von vordefinierten Datentypen
- eigene Datentypen ableitbar
- Namensraumunterstützung
- Reihenfolgeunabhängige Strukturen

## **einfache Datentypen** (simple types)

- beschreiben unstrukturierten Inhalt ohne Elemente oder Attribute (PCDATA)

## **komplexe Datentypen** (complex types)

- beschreiben strukturierten XML-Inhalt mit Elementen oder Attributen
- natürlich auch gemischten Inhalt

	( primitive )	abgeleitete
einfache	<p>xsd:string                      xsd:language                      xsd:integer                      ...</p>	<pre>&lt;xsd:simpleType name="longitudeType"&gt;   &lt;xsd:restriction base="xsd:integer"&gt;     &lt;xsd:minInclusive value="-180"/&gt;     &lt;xsd:maxInclusive value="180"/&gt;   &lt;/xsd:restriction&gt; &lt;/xsd:simpleType&gt;</pre>
komplexe	<pre>&lt;xsd:complexType&gt;   &lt;xsd:sequence&gt;     ...   &lt;/xsd:sequence&gt; &lt;/xsd:complexType&gt;</pre>	<pre>&lt;xsd:complexType name="BookTypeWithID"&gt;   &lt;xsd:complexContent&gt;     &lt;xsd:extension base="BookType"&gt;       &lt;xsd:attribute name="ID"         type="xsd:token"/&gt;     &lt;/xsd:extension&gt;   &lt;/xsd:complexContent&gt; &lt;/xsd:complexType&gt;</pre>

```
<course>
  <title>Semantic Web</title>
  <lecturers>
    <name>
      <title>Prof. Dr.-Ing.</title>
      <first>Robert</first>
      <last>Tolksdorf</last>
    </name>
  </lecturers>
  <date>12/11/2004</date>
  <abstract>...</abstract>
</course>
```

- **Namenskonflikt:** gleicher Name, aber unterschiedliche Bedeutung
- z.B. Titel einer Veranstaltung vs. Titel einer Person
- in einem Dokument unterschiedliche Vokabularien

# Auflösung durch Präfixe

```
<course:course>
  <course:title> Semantic Web </course:title>
  <course:lecturers>
    <pers:name>
      <pers:title> Prof. Dr.-Ing. </pers:title>
      <pers:first> Robert </pers:first>
      <pers:last> Tolksdorf </pers:last>
    </pers:name>
  </course:lecturers>
  <course:date> 12/11/2004 </course:date>
  <course:abstract> ... </course:abstract>
</course:course>
```

- Präfixe geben Kontext an:  
Aus welchem Bereich stammt der Name
  - z.B. pers:title vs. course:title

# Namensräume in XML

- WWW: Namensräume müssen global eindeutig sein.
- In XML wird Namensraum mit URI identifiziert.
- Zuerst wird Präfix bestimmter Namensraum zugeordnet, z.B.:



- Anschließend kann der Namensraum-Präfix einem Namen vorangestellt werden: z.B. pers:title
- Beachte: Wahl des Präfixes egal!



- **xmlns="URI"** statt xmlns:prefix="URI"
- Namensraum-Präfix kann weggelassen werden.
- Standard-Namensraum gilt für das Element, wo er definiert ist.
- Kind-Elemente erben Standard-Namensraum von ihrem Eltern-Element.
- Ausnahme: Standard-Namensraum wird überschrieben
- Beachte: Standardnamensräume gelten nicht für Attribute

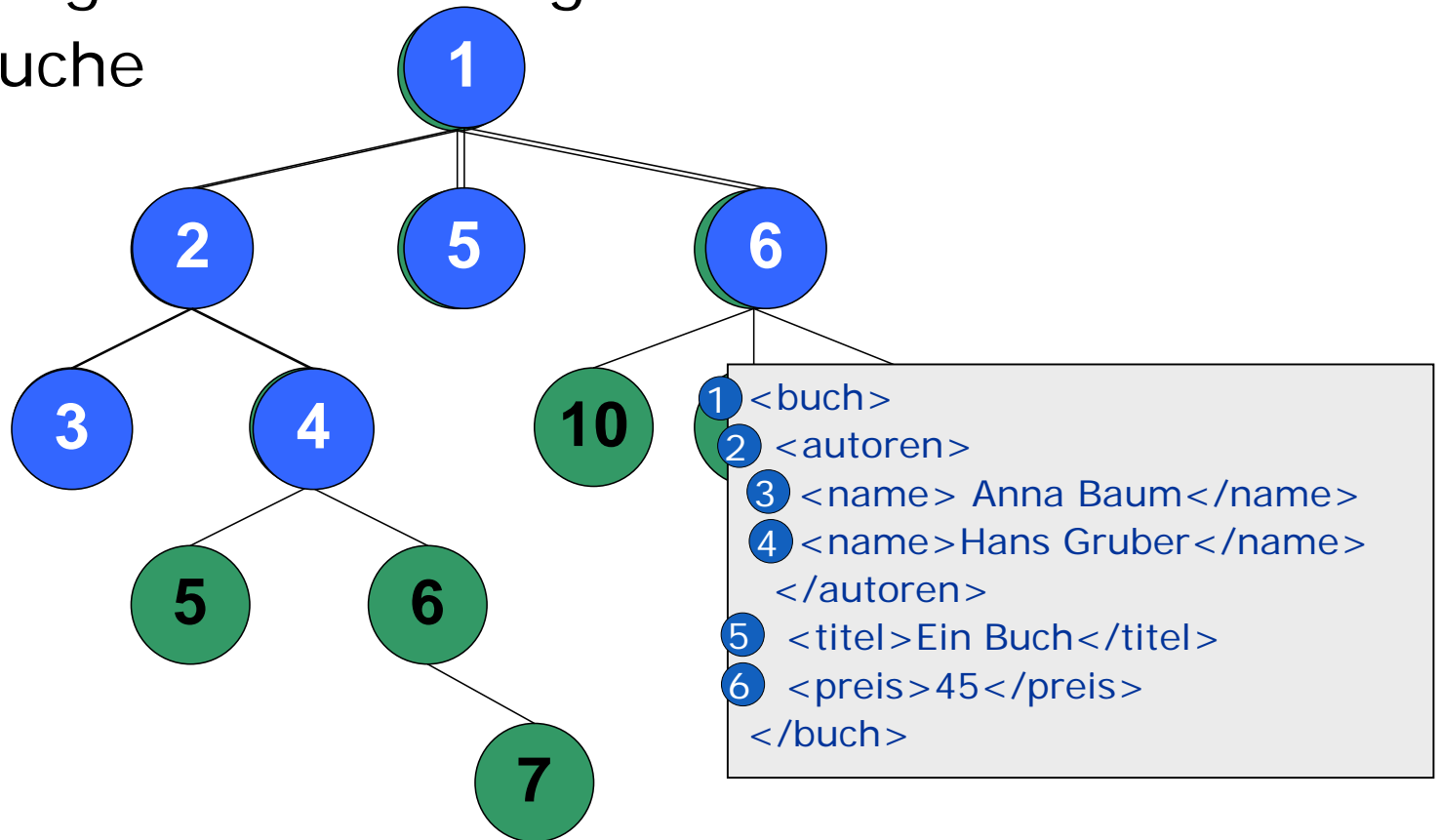
## Qualified vs. Unqualified

- Element- oder Attribut-Name heißt **namensraumeingeschränkt (qualified)**, wenn er einem Namensraum zugeordnet ist.
- Einschränkung vom Element-Namensraum:
  1. Standard-Namensraum festlegen
  2. Namensraum-Präfix voranstellen
- Einschränkung vom Attribut-Namensraum:
  1. Namensraum-Präfix voranstellen

- ähnliches Modell wie in DOM
  - XML-Dokument als Baum mit Elementen, Attributen und PCDATA als Knoten
- virtuelle Dokument-Wurzel (Wurzelknoten):  
durch "/" repräsentiert (links von "/" steht nichts)
  - ⇒ Wurzel-Element immer Kind von "/":  
z.B. /root

# Dokumentenreihenfolge

- Baummodell als Basis
- feste Dokumentreihenfolge (document order) = Reihenfolge der Start-Tags im Dokument
- Tiefensuche



- Elemente werden einfach **über ihren Namen** identifiziert:  
z.B. **order** oder **order/item**
- Attribute werden mit "**@name**" identifiziert:  
z.B. **@id** oder **order/@id**

```
<?xml version="..." encoding="..."?>  
<order id="O56">  
  <item item-id="E16-2">  
    <name>buch</name>  
  </item>  
</order>
```

# Was ist XSLT?

- in XML beschriebene Sprache zur Transformation von XML-Dokumenten
- eine beschreibende Sprache
- XSLT-Programme (stylesheets) haben XML-Syntax
  - plattformunabhängig
- erlaubt XML-Dokumente in beliebige Textformate zu transformieren:
  - XML → XML/HTML/XHTML/WML/RTF/ASCII ...
- W3C-Standard seit 1999

- a)  $K :=$  Dokument-Wurzel ("/") des Ursprungsdocumentes
  
- b) Identifiziere alle Templates, die auf  $K$  anwendbar sind.
  1. Ist genau ein Template anwendbar
    - wende es an
    - Fertig.
  2. Sind mehre Templates anwendbar, dann
    - wende das speziellste an
      - z.B. ist "/order" spezieller als "/\*".
    - Fertig.
  3. Ist kein Template anwendbar
    - wiederhole b) für alle Kinder  $K'$  mit  $K := K'$ .

# Template-Konflikte

- mehrere Templates auf den gleichen Knoten anwendbar
- Lösung → Prioritätsregeln:
  1. Eine spezifische Information hat Vorrang vor einer Regel für allgemeinere Information  
Beispiel: `match="/buch/authors/autor"`  
`match="//autor"`
  2. Suchmuster mit Wildcards (\* oder @\*) sind allgemeiner als entsprechende Muster ohne Wildcards
  3. Nur wenn 1. & 2. nicht zutreffen → Reihenfolge der Templates entscheidend
  4. Priorität der Templates durch Attribut `priority` bestimmbar
    - Standard = 0
    - niedrigere Priorität < 0 < höhere Priorität



# Transformations-Beispiel

## Stylesheet

```
<xsl:template match="A">  
  <xsl:value-of select="@id"/>  
</xsl:template>
```

```
<xsl:template match="B">  
  <xsl:value-of select="@id"/>  
</xsl:template>
```

```
<xsl:template match="C">  
  <xsl:value-of select="@id"/>  
</xsl:template>
```

```
<xsl:template match="D">  
  <xsl:value-of select="@id"/>  
</xsl:template>
```

## Dokument

```
<source>  
  <A id="a1">  
    <B id="b1"/>  
    <B id="b2"/>  
  </A>  
  <A id="a2">  
    <B id="b3"/>  
    <B id="b4"/>  
    <C id="c1">  
      <D id="d1"/>  
    </C>  
    <B id="b5">  
      <C id="c2"/>  
    </B>  
  </A>  
</source>
```

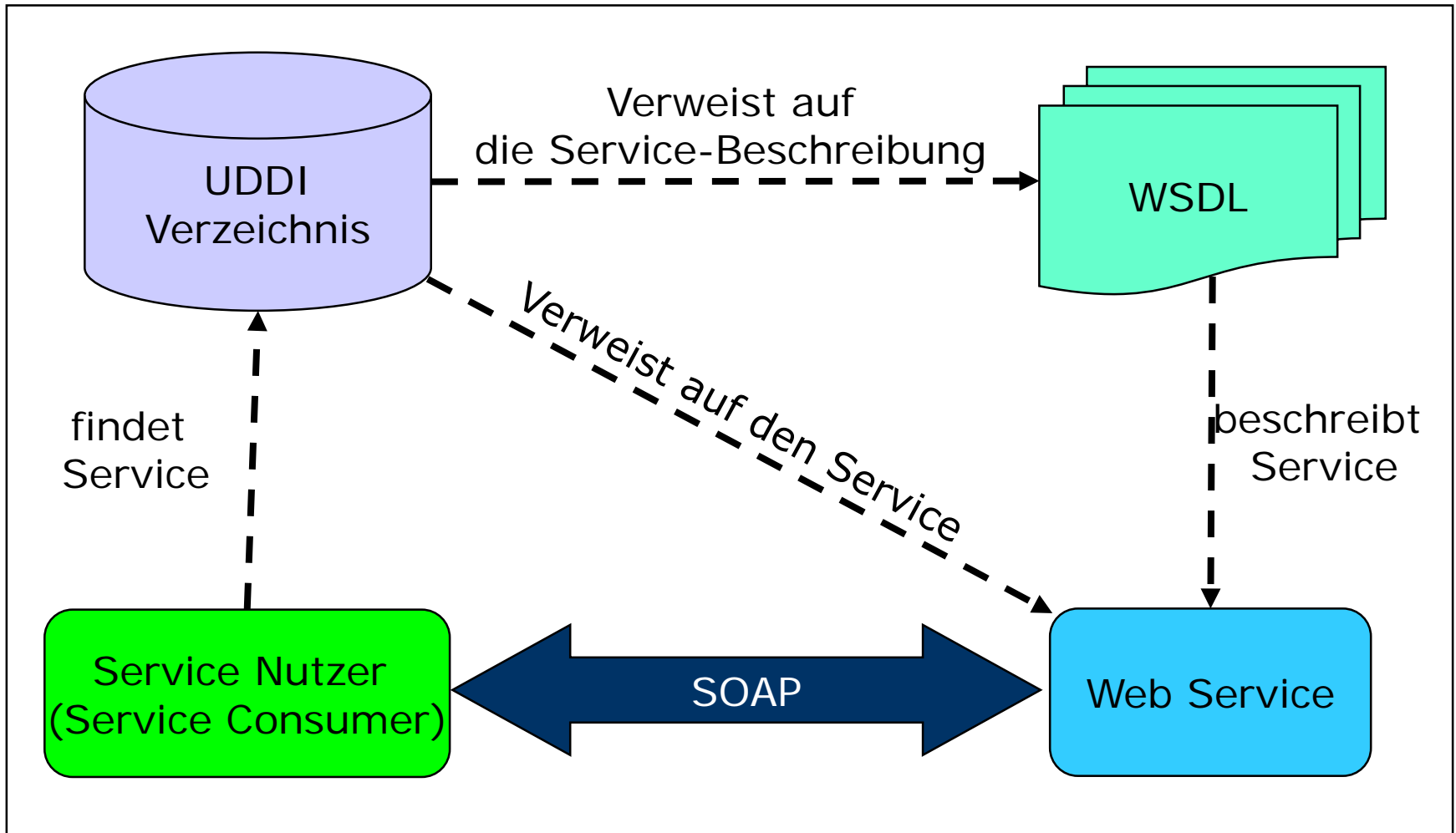
kein  
Template  
anwendbar

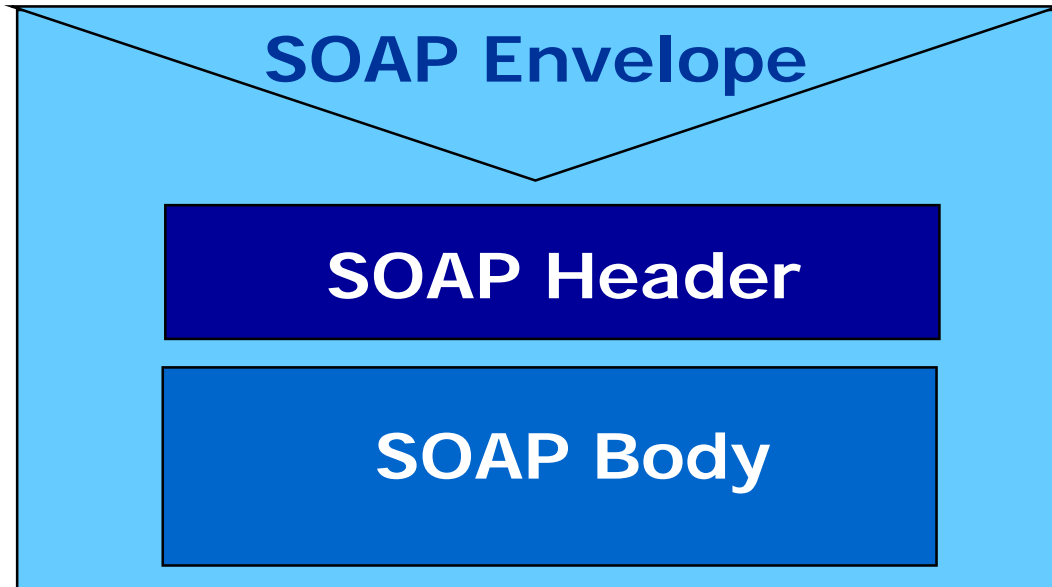
Template  
"A" wird  
angewandt

## Ausgabe

a1  
a2

Template "B"  
wäre anwendbar,  
es werden aber  
keine Templates  
aufgerufen!





XML-Deklaration

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

```
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap/envelope/">
```

```
<!-- SOAP Header -->
```

```
<!-- SOAP Body -->
```

```
</env:Envelope>
```

SOAP Version 1.2

SOAP Version 1.1:

<http://schemas.xmlsoap.org/soap/envelope/>

- abgeleitet vom Prinzip zur Aufteilung von Aufgaben („separation of concerns“)
  - erlaubt unabhängige Evolution der Einzelkomponenten
  - Beispiel:
    - Rendering von HTML im Web-Browser (Client) – Servieren von HTML-Seiten auf einem Web-Server (Server)

- jede Nachricht enthält alle notwendigen Informationen, die dem Empfänger die Verarbeitung erlauben
  - kein gespeicherter Kontext am Server
  - Sitzungszustand beim Client
    - Beispiele:
      - Cookies
      - Session IDs

- Antworten auf Anfragen implizit oder explizit als cachable oder non-cachable klassifizieren
  - Client darf cachable-Antworten für spätere Anfragen wiederverwenden

- Vereinfachung der Systeminfrastruktur und hohe Sichtbarkeit von Interaktionen durch standardisierten Zugriff auf Komponenten
  - Ressourcenidentifikation
    - universelle Syntax für Identifier
    - Identifikation von „Dingen“ (Things)
  - Manipulation von Repräsentationen
    - wohldefinierte Aktionen auf einer Sequenz von Bytes + Metadaten (=Repräsentation), die den aktuellen oder gewünschten Zustand einer Ressource darstellt
- selbstbeschreibende Nachrichten
- Hypermedia

- Vereinfachung der Systeminfrastruktur und hohe Sichtbarkeit von Interaktionen durch standardisierten Zugriff auf Komponenten
  - Ressourcenidentifikation
  - Manipulation von Repräsentationen
- selbstbeschreibende Nachrichten
  - die Semantik von Nachrichten ist für alle verarbeitenden Komponenten (Mittler) sichtbar
  - Mittler können Inhalte verändern
- Hypermedia
  - alle Inhalte UND Informationen zum Zustandsübergang (Hyperlinks) werden an den Client weitergegeben



- Unabhängigkeit der einzelnen Komponenten durch beschränkte Sicht auf das hierarchisch geschichtete Gesamtsystem
  - Komponenten „sehen“ nur bis zum Interaktionspartner

- Erweiterbarkeit des Systems durch Download von Code nach dem Deployment
  - Beispiele:
    - Applets
    - Scripte
  - optionales Prinzip, weil es die Sichtbarkeit von Interaktionen reduziert

- **HTTP**
  - Stateless Client-Server
  - Caching
  - Uniform Interface: standardisierte Manipulation von Repräsentationen via GET, PUT, POST, DELETE
  - Layered System
- **HTTP-URIs**
  - Uniform Interface: Ressourcenidentifikation
- **Hypertext und Hyperlinks**
  - Uniform Interface: standardisierte Repräsentation von Ressourcenzustand und Zustandsübergängen

Client

```
GET / HTTP/1.1
User-Agent: Mozilla/5.0 ... Firefox/10.0.3
Host: markus-luczak.de:80
Accept: */*
```

```
HTTP/1.1 200 OK
Server: Apache/2.0.49
Content-Language: en
Content-Type: text/html
Content-length: 2990
```

```
<!DOCTYPE html>
<html xml:lang="en"
...
```

Server

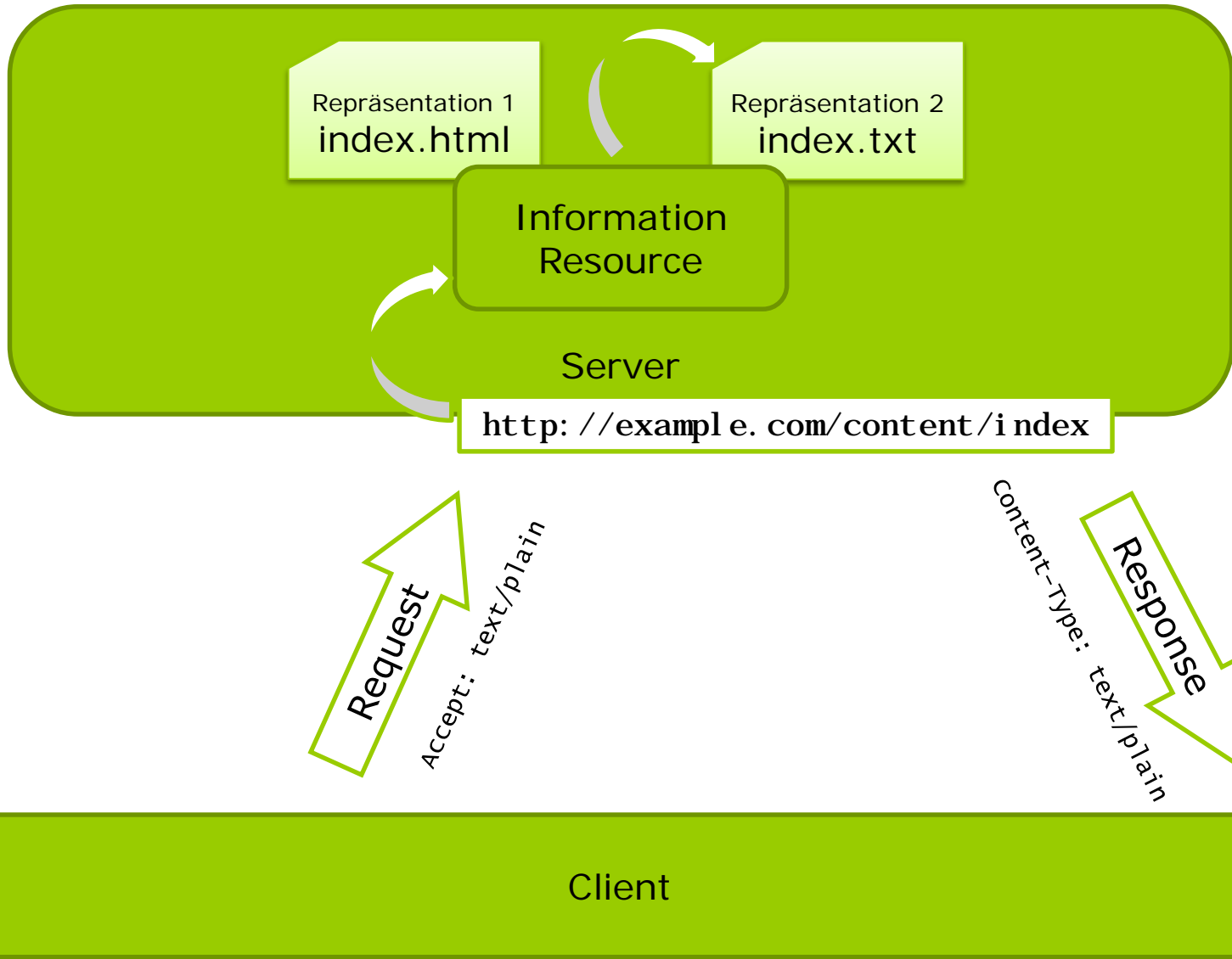
- **Information Resources**

- URI-identifizierte Ressourcen, deren Repräsentation in einer Nachricht elektronisch übertragen werden kann
- eine bestimmte Repräsentation anzufordern und zu erhalten nennt man **Dereferenzierung**

- **Non-Information Resources**

- abstrakte Konzepte wie z.B. in Ontologien modelliert
  - Beispiel: Meter  
Ich kann lediglich andere Information Resources ausliefern, die „Meter“ beschreiben aber eben keine Repräsentation von Meter sind.
- URI-identifizierbar aber nicht dereferenzierbar

# Content Negotiation – Req.-Resp.



- `<Buch>Dieses Buch</Buch> hat den Titel  
<Titel>Semantic Web Grundlagen</Titel>`
- `<foo>Dieses Buch</foo> hat den Titel  
<bar>Semantic Web Grundlagen</bar>`
- natürliche Sprache
- Mehrdeutigkeit

# <Apple>



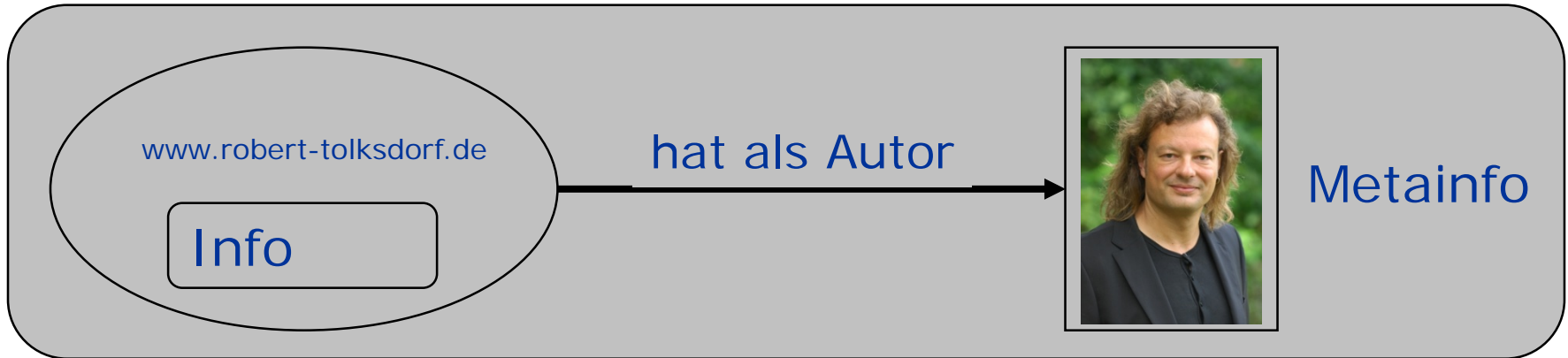
<http://apple.com/>



<http://applefuits.com/>



- Informationen und Metainformationen:



- In RDF als Satz ausgedrückt:

"www.robert-tolksdorf.de"	Subjekt
hat als Autor	Prädikat
Robert Tolksdorf"	Objekt

- „RDF-Welt“: Gerichteter Graph
  - Knoten (Ressourcen)
  - Kanten (Properties)
- Ressourcen (RDF Resource)
  - Alles worüber man Aussagen machen kann
  - Identifiziert durch URIs (qualified URIs = URI + fragment identifier)
  - Aussagen sind auch Ressourcen
- Eigenschaften/Beziehungen (RDF Property)
  - Verbinden Ressourcen miteinander oder Ressourcen zu Werten (RDF Literal)
- Aussagen (RDF Statement)
  - (Subjekt, Prädikat, Objekt)
  - “Resource has Property with Value”



# Als XML-Baum

```
<author>
  <uri>page</uri>
  <name>Ora</name>
</author>
```

```
<document>
  <details>
    <uri>href="page"</uri>
    <author>
      <name>Ora</name>
    </author>
  </details>
</document>
```

```
<document>
  <author>
    <uri>href="page"</uri>
    <details>
      <name>Ora</name>
    </details>
  </author>
</document>
```

```
<document href="page">
  <author>Ora</author>
</document>
```

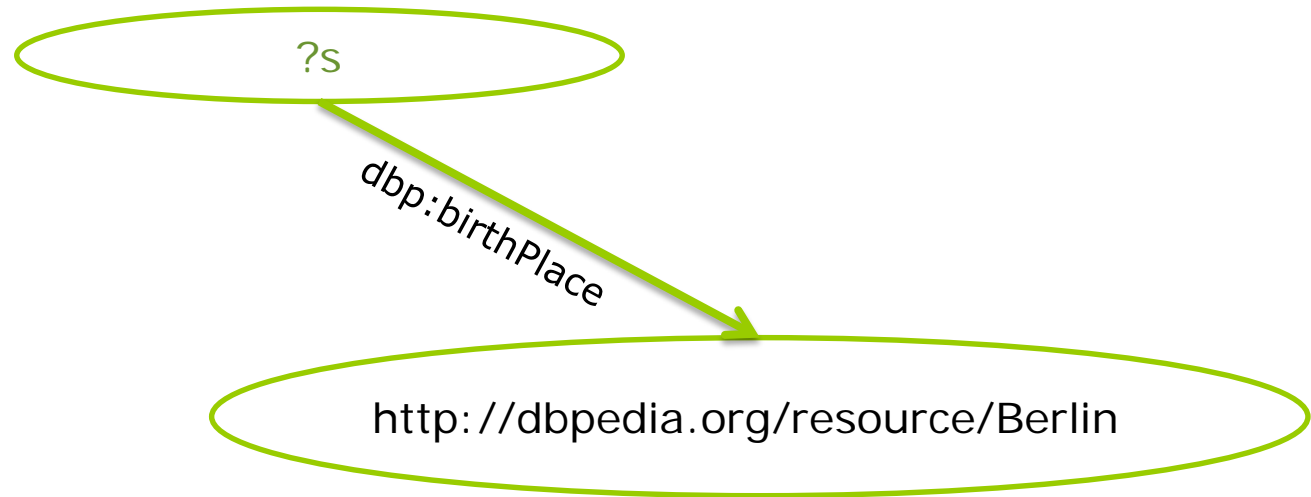
```
<document href="http://www.w3.org/test/page" author="Ora" />
```

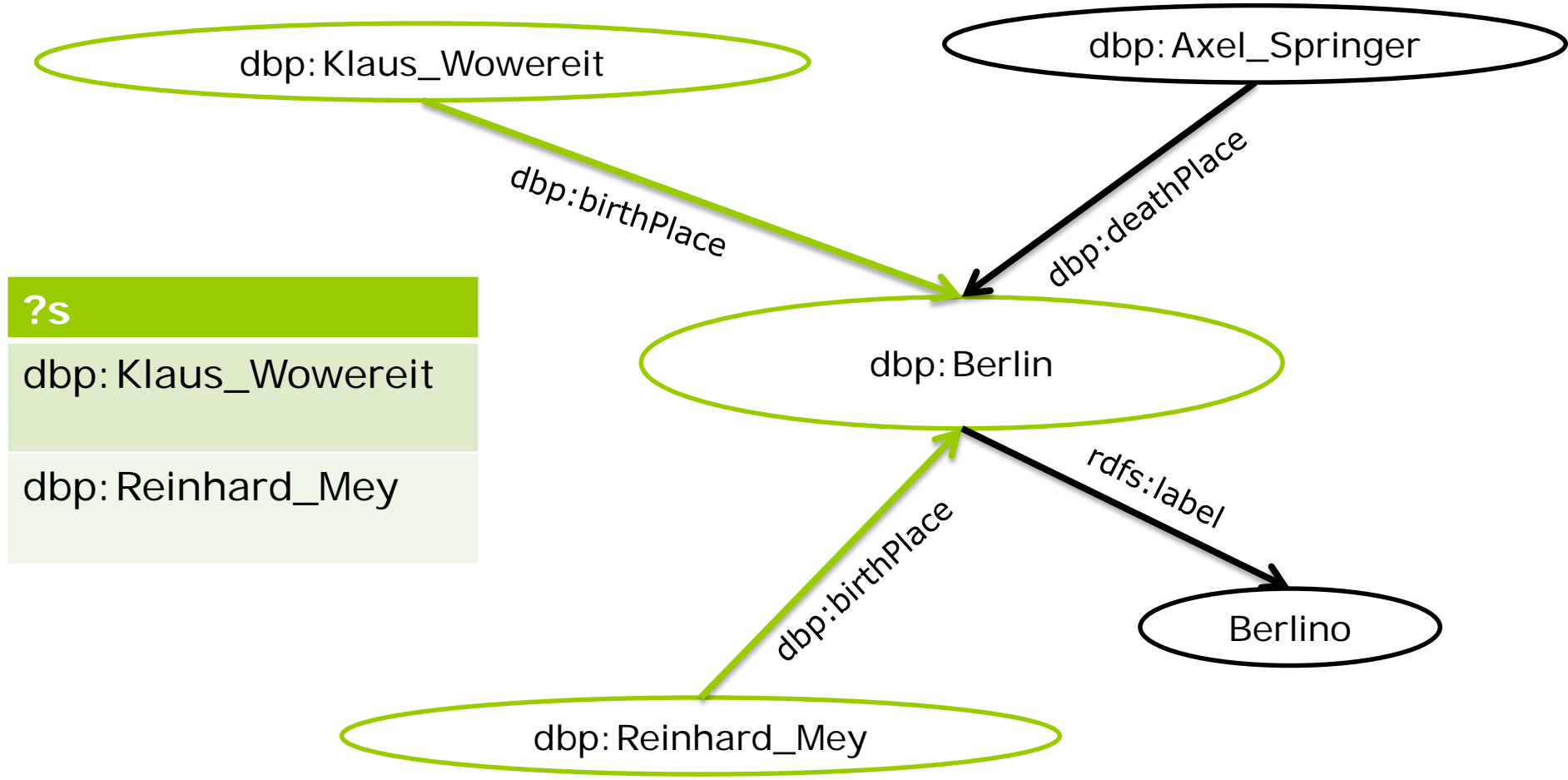
```
<Description about="http://www.w3.org/test/page"  
  s:Author ="http://www.w3.org/staff/Ora" />
```

...



- SPARQL: Anfragesprache für RDF-Daten
- Grundsatz: Pattern-Matching
  - beschreibe Graphpattern
  - frage RDF-Graph mit diesem Pattern an
  - Subgraphen, die Pattern matchen kommen in die Ergebnismenge





1. URIs als Namen für alle "Dinge"

`http://dbpedia.org/resource/Berlin`

2. `http://` URIs damit man im Web auf diese Namen zugreifen kann

*Content Negotiation*

3. Wenn eine URI aufgerufen wird sollen sinnvolle Informationen entsprechend der Standards (RDF, SPARQL) geliefert werden

`http://dbpedia.org/page/Berlin`  
`http://dbpedia.org/data/Berlin`

4. Links zu anderen URIs, damit Nutzer mehr "Dinge" finden können

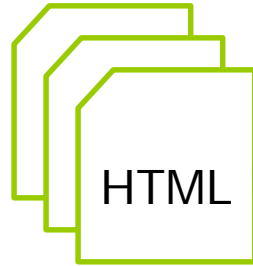
`yago-res:Berlin`      S  
                         owl:sameAs      P  
                         dbpedia:Berlin      O



HTTP GET



`http://dbpedia.org/resource/Klaus_Wowereit`



HTML



RDF

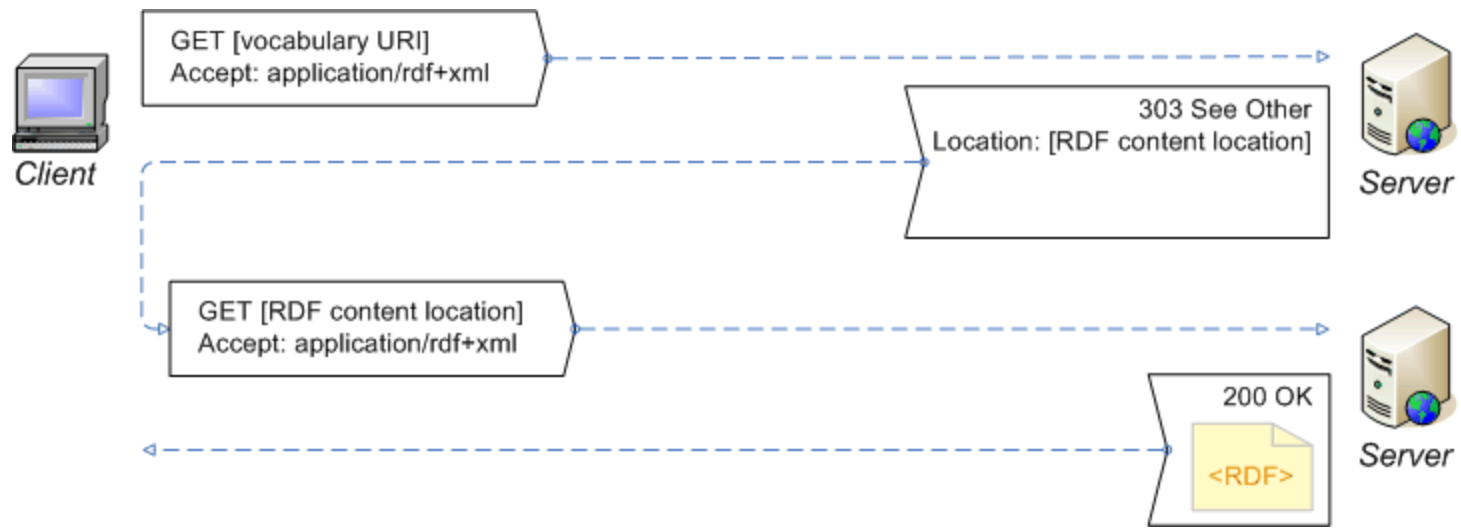
`http://dbpedia.org/page/Klaus_Wowereit`

`http://dbpedia.org/data/Klaus_Wowereit`

Linked Data Server  
Infrastructure



Data Source



# Microdata, RDFa, Microformats

Feature	RDFa 1.1	Microdata 1.0	Microformats 1.0
Relative Complexity	High	Medium	Low
Data Model	Graph	Tree	Tree
Item optionally identified by IRI	Yes	Yes	No
Item type optionally specified by IRI	Yes	Yes	No
Item properties specified by IRI	Yes	Yes	No
Multiple objects per page	Yes	Yes	Yes
Overlapping objects	Yes	Yes	No
Plain Text properties	Yes	Yes	Yes
IRI properties	Yes	Yes*	No
Typed Literal properties	Yes	No	No
XML Literal properties	Yes	No	No
Language tagging	Yes	Yes	Inconsistent

...

<http://manu.sporny.org/2011/uber-comparison-rdfa-md-uf/>