

XML

<subtitle>Models</subtitle>

<author>Prof. Dr. Christian Pape</author>

Content

- Overview
- Document Type Definitions
- XML Schema
- Modeling

Overview

- So far
 - Learned how to read and write XML documents
 - Syntax of XML documents defined only by “example”
- But
 - No information given whether a tag is optional or mandatory, whether it can be repeated or not, etc.
 - No definition of how data (content of an element) is represented, e.g. numbers, dates

```
<movies>
  <movie>
    <title>Man in Black</title>
    <rating>3</rating>
  </movie>
</movies>
```

- Is rating optional?
- Can there be more than one rating for a movie?
- Can there be more than one movie within the movies tag?
- Dates: 2005-05-7, 5/7/2005, ...

Overview

- So far
 - Learned how to read and write XML documents
 - Syntax of XML documents defined only by “example”
- But
 - No information given whether a tag is optional or mandatory, whether it can be repeated or not
 - No definition of how data (content of an element) is represented, e.g. numbers, dates

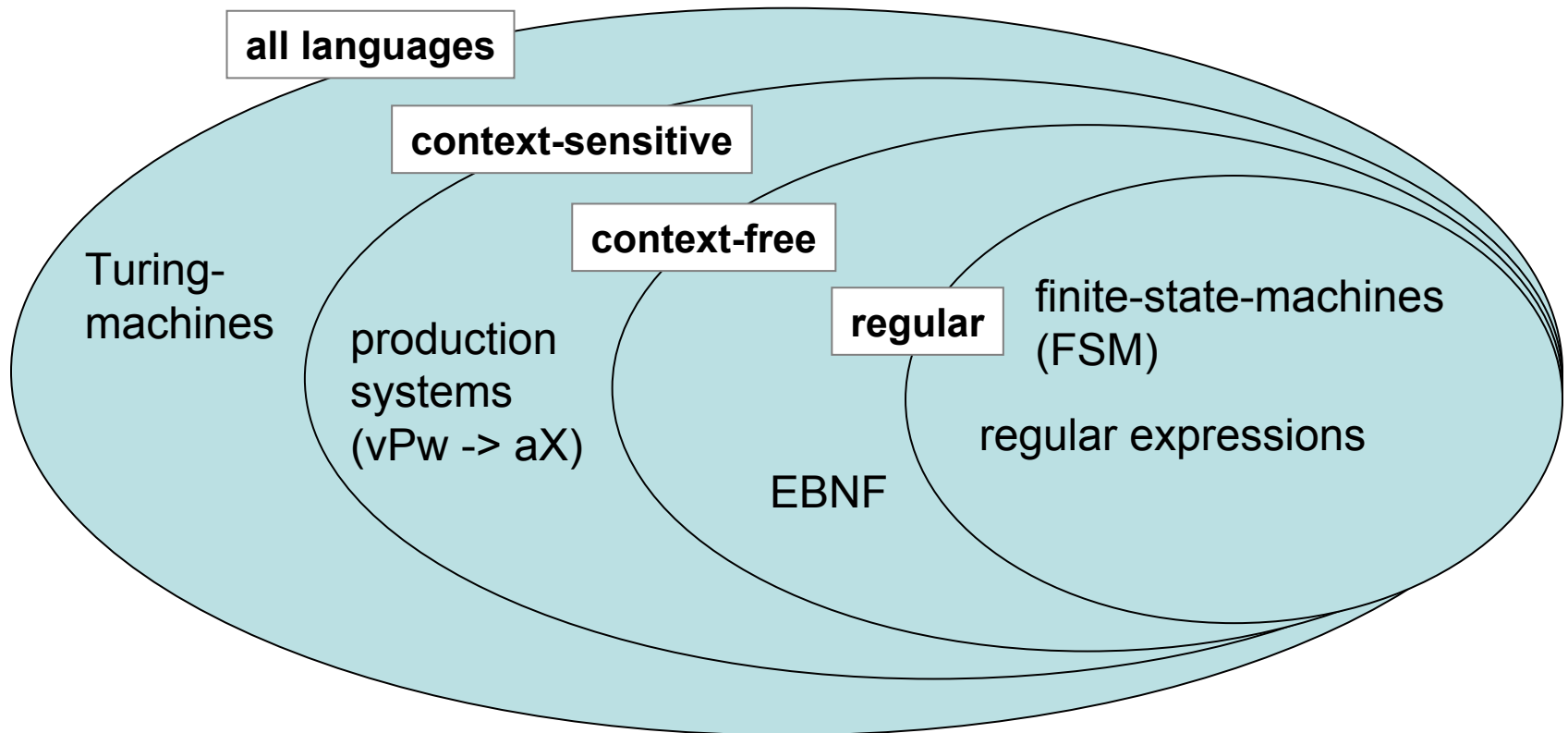
```
<movies>
  <movie>
    <title>Man in Black</title>
    <rating>3</rating>
  </movie>
</movies>
```

← Document Type Definition or XML Schemas

← Type system of XML Schemas

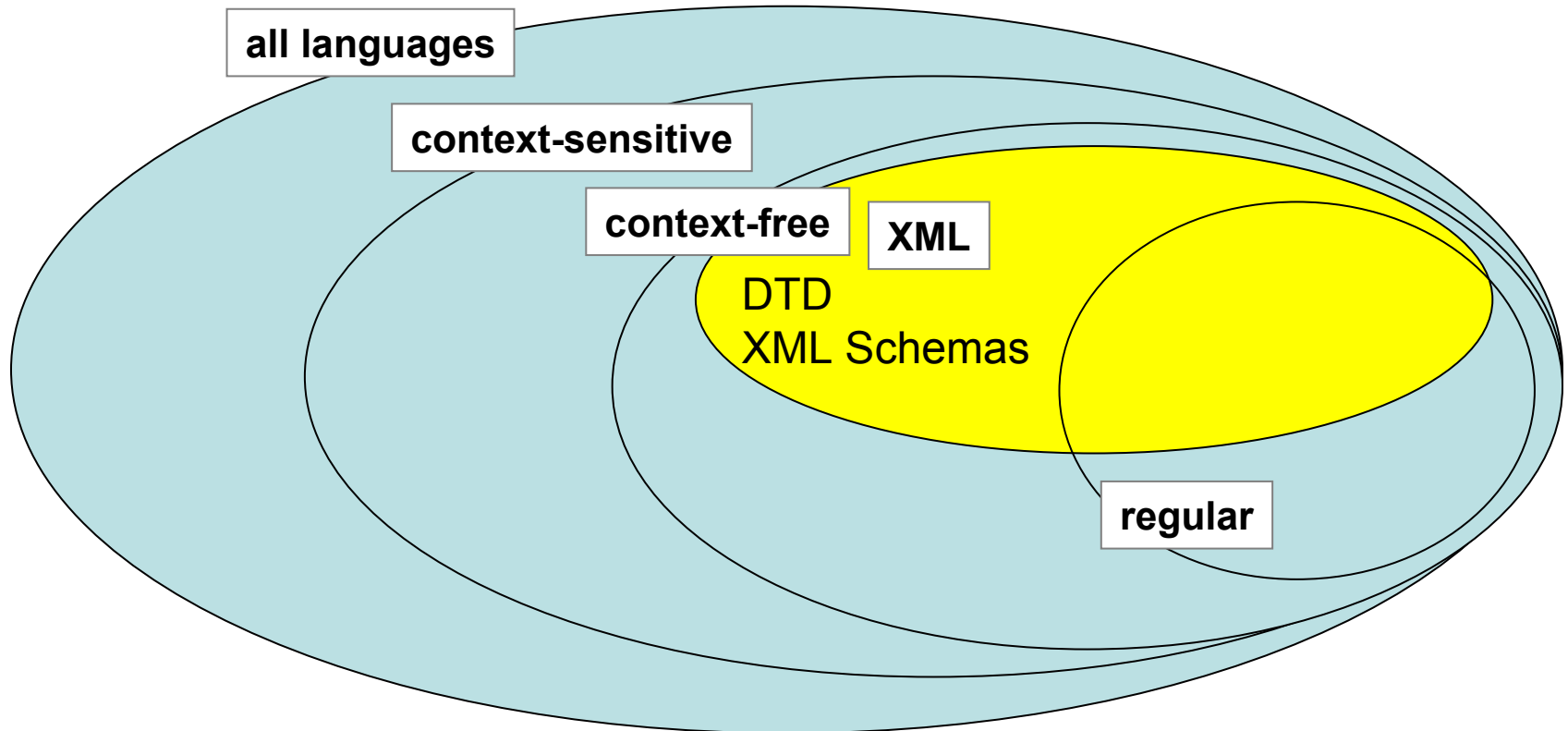
Overview

- Formal languages: Notations for specifying the syntax of documents unambiguously
- Chomsky Hierarchy



Overview

- XML defines a subset of context free languages
 - DTD, XML Schemas are tools to specify a XML language
 - Regular languages are **not** a subset of XML (see assignment)



Overview

- Document Type Definitions (DTD)
 - 20 year old, part of SGML
 - Developed before XML
 - Simple, proven solution
 - Not itself in XML syntax
 - No type system (content is text)
- XML Schemas
 - Developed after XML
 - More complicate than DTD
 - In XML syntax
 - Elaborated and extensible type system (content is number represented as text)
- Both are equally powerful with respect to the syntactical definition of an XML language

Overview

- Why DTD / XML Schema?
 - Well-formed documents not strict enough for automatically processing
 - Parser can check syntax of documents before processing or writing them
 - Electronics documents should be as unambiguous for automated processing as possible
- Why two different mechanism?
 - DTD was there, but not suitable for extensions (like type)
 - DTD not in XML itself
- When to use which Mechanism
 - DTD simple to use, sufficient to check documents validity
 - XML Schema more powerful and complicated, for rigorous type checking and automatic mapping (XML <-> Java)

Content

- Overview
- **Document Type Definitions (DTD)**
- XML Schema
- Modeling

DTD / Example

- One or more movie elements
- Mandatory title followed by mandatory rating
- Title and rating contain plain text

```
<?xml version="1.0"?>
<movies>
  <movie>
    <title>Man in Black</title>
    <rating>3</rating>
  </movie>
</movies>
```

DTD / Example

```
1: <!ELEMENT movies (movie+)>
2: <!ELEMENT movie (title, rating)>
3: <!ELEMENT title (#PCDATA)>
4: <!ELEMENT rating (#PCDATA)>
```

DTD

1. Defines a tag (Element) named “movies” that must contain one or more “movie” elements
2. Defines an element “movie” that must contain a “title” followed by a “rating”
3. Defines an element “title” that contains Parsed Character Data (PCDATA)
4. Defines an element “rating” that contains PCDATA

DTD / Example

File "movies.dtd":

```
<!ELEMENT movies (movie+)>
<!ELEMENT movie (title, rating)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT rating (#PCDATA)>
```

DTD fragment

„Link“ to XML's DTD

```
<?xml version="1.0"?>
<!DOCTYPE movies SYSTEM "movies.dtd">
<movies>
  <movie>
    <title>Man in Black</title>
    <rating>3</rating>
  </movie>
</movies>
```

DTD / Example

```
<!ELEMENT movies (movie+)>
<!ELEMENT movie (title, rating)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT rating (#PCDATA)>
```

DTD

well-formed

```
<movies>
  <movie>
    <rating>3</rating>
    <title>Man in Black</title>
    <media>DVD</media>
  </movie>
</movies>
```

Not valid: rating must follow title, tag media does not exist

- A XML document is **valid** if it is well-formed and it is a word of the language defined by its attached DTD (or XML Schema)

DTD / Example

- XML document may contain its own DTD
 - Useful if system cannot store DTD (XML processing in a distribute environment)

```
<?xml version="1.0"?>
<!DOCTYPE movies [
  <!ELEMENT movies (movie+)>
  <!ELEMENT movie (title, rating)>
  <!ELEMENT title (#PCDATA)>
  <!ELEMENT rating (#PCDATA)>
]>
<movies>
  <movie>
    <title>Man in Black</title>
    <rating>3</rating>
  </movie>
</movies>
```

DTD / Example

- Mixing external and internal DTDs
 - Internal DTD overwrites external definitions
- Avoid! Error prone and confusing.

```
<?xml version="1.0"?>
<!DOCTYPE movies SYSTEM "movie.dtd" [
  <!ELEMENT movie (title, rating?)>
]>
<movies>
  <movie>
    <title>Man in Black</title>
  </movie>
</movies>
```

DTD / Doctype Declaration

```
<!DOCTYPE root SYSTEM URI>  
<!DOCTYPE root [ DTD ]>  
<!DOCTYPE root SYSTEM URI [ DTD ]
```

- *root* is element name defined in the DTD
 - Root element is not defined in DTD itself, but in the doctype declaration
- The location of the DTD (file) is given as a URI after the keyword SYSTEM
- If no external DTD is used, SYSTEM is omitted and the DTD is given in brackets []
- If internal and external DTD is used, the internal DTD is given in brackets after the URI of the external DTD

DTD / Element Declaration

`<!ELEMENT name content-model>`

- The element declarations define the vocabulary and structure of the XML dialect
 - *name* must be a valid tag name
 - *content-model* defines which content for *name* is valid
- When an application or parser reads an element *name*, then its content is checked against the rules defined in its content model

DTD / Element Declaration – Content models

`<!ELEMENT name content-model>`

- Three types of content models exist
 1. ANY any well-formed content is allowed for name
 2. EMPTY no content at all is allowed
 3. Regular expressions over tag name, regular operators, and #PCDATA

DTD / Element Declaration – ANY

```
<!ELEMENT example      (description, example-xml)>
<!ELEMENT example-xml ANY>
```

DTD (fragment)

```
<example>
  <description>
    An XML dialect for rating movies
  </description>
  <example-xml>
    <movies>
      <movie>
        <title>Man in Black</title>
        <rating>3</rating>
      </movie>
    </movies>
  </example-xml>
</example>
```

well-formed and valid

DTD / Element Declaration – EMPTY

```
<!ELEMENT movies (movie+)>
<!ELEMENT movie (title, rating, available-on-dvd?)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT rating (#PCDATA)>
<!ELEMENT available-on-dvd EMPTY>
```

DTD

```
<movies>
  <movie>
    <title>Man in Black</title>
    <rating>3</rating>
    <available-on-dvd/>
  </movie>
</movies>
```

well-formed and valid

invalid

well-formed

```
<movies>
  <movie>
    <title>Man in Black</title>
    <rating>3</rating>
    <available-on-dvd>yes</available-on-dvd>
  </movie>
</movies>
```

DTD / Element Declaration - EMPTY

- (X)HTML: br (or img) are empty

```
<!ELEMENT br EMPTY>
```

```
<h1>Hello world</h1>  
  This is text with a line break <br/>  
  This test is on the next line  
  This is <br>not</br> valid.
```

DTD / Element Declaration – Regular Expression

```
<!ELEMENT movies (movie+)>
<!ELEMENT movie (title, rating*, media?)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT rating (#PCDATA)>
```

DTD (fragment)

```
<movies>
  <movie>
    <title>Man in Black</title>
    <rating>3</rating>
    <rating>5</rating>
  </movie>
</movies>
```

well-formed and valid

```
<movies>
  <movie>
    <title>Man in Black</title>
    <media>DVD</media>
  </movie>
</movies>
```

- + one or more occurrence
- * Zero or more
- ? Zero or one

DTD / Element Declaration – Regular Expression

| Operator | Semantics |
|----------------|----------------------------------|
| tag name t | Child element with tag name t |
| #PCDATA | Any text (parsed character data) |
| r_1, r_2 | r_1 followed by r_2 |
| $r_1 \mid r_2$ | r_1 or alternatively r_2 |
| r^+ | One or more times r |
| r^* | Zero or more times r |
| $r?$ | Zero or one time r |

Regular expression in parenthesis ()

Sub expressions can be set in parenthesis if necessary

DTD / Element Declaration – Regular Expression

```
<address-book>
  <entry>
    <name>Mr. X</name>
    <comment>Address not know</comment>
  </entry>
  <entry>
    <name>Christian Pape</name>
    <address>
      <street>Moltkestrasse 30</street>
      <location>
        <town>Karlsruhe</town>
      </location>
    </address>
  </entry>
</address-book>
```

DTD / Element Declaration – Regular Expression

```
<!ELEMENT address-book (entry+)>
<!ELEMENT entry          (name, address*, comment?)>
<!ELEMENT address        (street, location)>
<!ELEMENT location       ( town | ( zip-code, town) )>
```

DTD (PCDATA omitted)

```
<address-book>
  <entry>
    <name>Mr. X</name>
    <comment>Address not know</comment>
  </entry>
  <entry>
    <name>Christian Pape</name>
    <address>
      <street>Moltkestrasse 30</street>
      <location>
        <town>Karlsruhe</town>
      </location>
    </address>
  </entry>
</address-book>
```

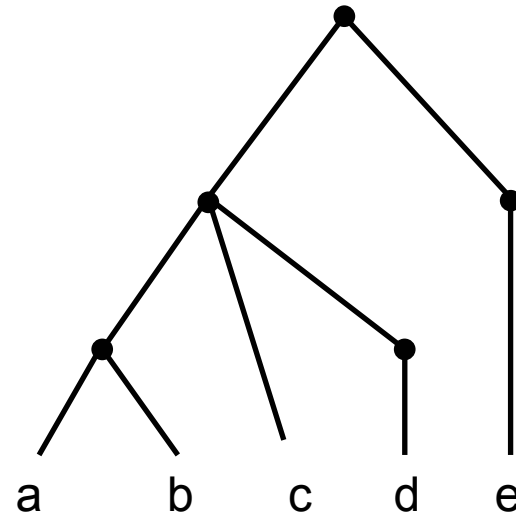
well-formed and valid

DTD / Element Declaration – Regular Expression

- Recursion in regular expressions is allowed

```
<!ELEMENT tree (node*)>  
<!ELEMENT node ( (node | leaf)* ) >  
<!ELEMENT leaf (#PCDATA)>
```

```
<tree>  
  <node>  
    <node>  
      <leaf>a</leaf>  
      <leaf>b</leaf>  
    </node>  
    <leaf>c</leaf>  
    <node>  
      <leaf>d</leaf>  
    </node>  
  </node>  
  <node>  
    <leaf>e</leaf>  
  </node>  
</tree>
```



DTD / Element Declaration – Mixed Content

```
<!ELEMENT address-book (entry+)>
<!ELEMENT entry          (name, address*, comments?)>
<!ELEMENT comments       (#PCDATA | b)*>
<!ELEMENT b               (#PCDATA)>
```

DTD (fragment)

```
<address-book>
  <entry>
    <name>Mr. X</name>
    <comments>Address <b>not</b> know</comments>
  </entry>
</address-book>
```

- **Mixed Content:** Element contains #PCDATA and a regular expression
 - #PCDATA must be separated with | from regular expression
 - The mixed content must always be repeated (*)

DTD / Element Declaration – Regular Expression

```
<!ELEMENT book-cover ( (title, author) | (title, subtitle) )>
```

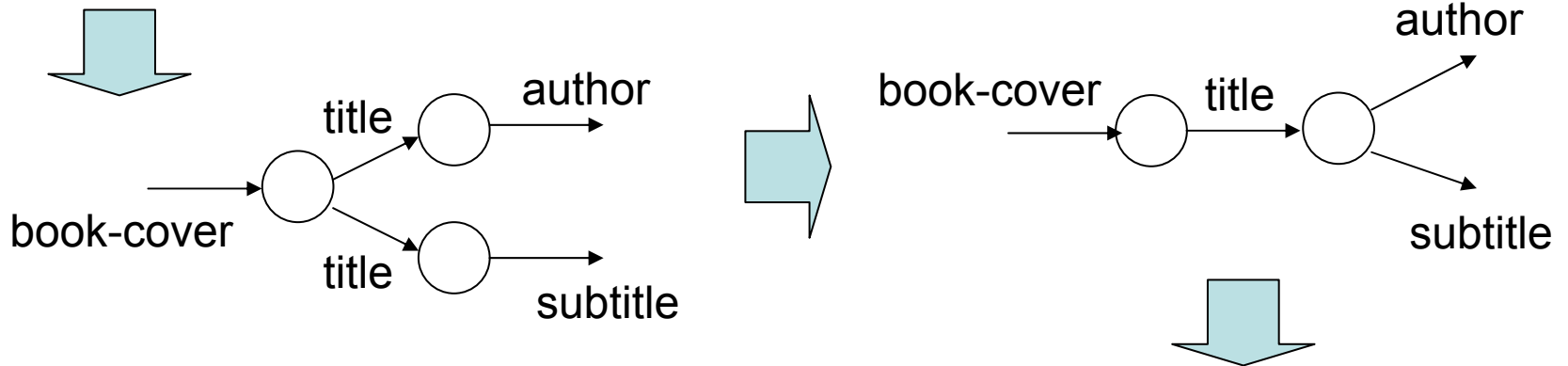
```
<book-cover>
  <title>
    XML by Example
  </title>
  <author>
    Benoît Marchal
  </author>
</book-cover>
```

- When reading title parser can not decide whether to choose (title, author) or (title, subtitle)
- Regular Expressions must be deterministic
 - Validating a document by reading one element at time
 - Advantage: better performance for XML parsers

DTD / Element Declaration – Regular Expression

- Every indeterministic regular expression can be transformed to a deterministic one
 - For an indeterministic regular expression there exists an equivalent indeterministic finite-state-machine (FSM)
 - For an indeterministic FMS there exists a deterministic FSM
 - For the deterministic FSM there exists a equivalent regular expression

<!ELEMENT book-cover ((title, author) | (title, subtitle))>



<!ELEMENT book-cover (title, (author | subtitle))>

DTD / Attributes

- Attributes must be declared in DTD, too
 - Keyword ATTLIST
 - Can occur anywhere in the DTD
 - should be placed directly behind declaration of corresponding element
 - Element can have several attributes
 - No order of attributes

```
<!ELEMENT email EMPTY>
<!ATTLIST email href          CDATA          #REQUIRED
                preferred (true | false) "false">
```

```
<email href="mailto:pach0003@fh-karlsruhe.de" preferred="true"/>
```

well-formed and valid

```
<email href="mailto:pach0003@fh-karlsruhe.de"/>
```

DTD / Attribute Declaration

```
<!ATTLIST element-name attr-name-1 type-1 default-1  
...  
          attr-name-n type-n default-n>
```

- Attribute name
 - Attribute name, unique among the given *element-name*
- Type
 - Defines which values are valid for this attribute
- Default
 - Defines whether attribute is mandatory, optional, and/or a default value
- Attributes in list are considered unordered

DTD / Attribute Declaration - Types

```
<!ATTLIST element-name attr-name-1 type-1 default-1  
...  
          attr-name-n type-n default-n>
```

| Attribute Type | Semantics |
|------------------------------------|---|
| CDATA | Character data, without <, but with entity references like < |
| $v_1 \mid v_2 \mid \dots \mid v_n$ | Attribute can be one of the given literal values |
| ID | Value is a document wide unique identifier |
| IDREF | Value is a reference to a value of a ID attribute |

DTD / Attribute Declaration - Types

```

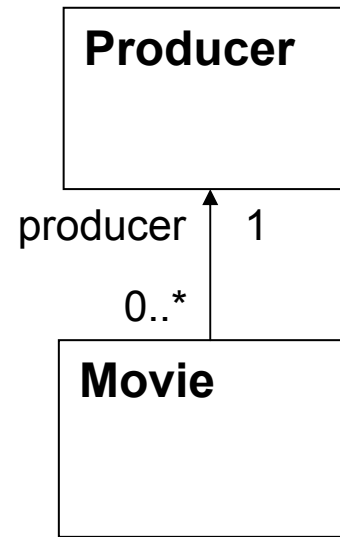
<!ELEMENT movies (producer+, movie+)>
<!ELEMENT producer (#PCDATA)>
<!ATTLIST producer pid ID #REQUIRED>
<!ELEMENT movie (title)>
<!ATTLIST movie download CDATA "www.download.com"
              rating (P|PG| ... ) #REQUIRED
              producer IDREF #REQUIRED>

```

```

<movies>
  <producer pid="123">columbia Pictures</producer>
  <movie producer="123"
           rating="PG">
    <title>Man in Black</title>
  </movie>
  <movie producer="123"
           download="www.mib2.com"
           rating="PG">
    <title>Man in Black II</title>
  </movie>
</movies>

```



DTD / Attribute Declaration

```
<!ATTLIST element-name attr-name-1 type-1 default-1  
...  
          attr-name-n type-n default-n>
```

| Attribute Default | Semantics |
|-------------------|---|
| #REQUIRED | Attribute is mandatory |
| #IMPLIED | Attribute is optional with no default |
| v (a value) | Attribute is optional with default v |
| #FIXED v | Attribute is optional when given, then value must be v |

DTD / Attribute Declaration

<!ELEMENT an-element EMPTY>

<!ATTLIST an-element an-attribute CDATA #REQUIRED>

<!ATTLIST an-element an-attribute CDATA #IMPLIED>

<!ATTLIST an-element an-attribute (yes | no) "yes">

<!ATTLIST an-element an-attribute #FIXED "yes">

<an-element/>



<an-element an-attribute="yes"/>



<an-element an-attribute="no"/>



<an-element an-attribute="maybe"/>



DTD / Other Features – Entity References

- User defined entity references (like "e; ;)

```
<!ENTITY cp "Columbia Pictures">
```

- Parser replaces reference &cp; by given text

```
<movies>
  <producer ID="123">&cp;</producer>
  <movie publisher="123"
    rating="PG">
    <title>Man in Black</title>
  </movie>
</movies>
```

DTD / Other Features – Entity References

- User defined entity references (like "e; ;)

```
<!ENTITY cp "Columbia Pictures">
```

- Parser replaces reference &cp; by given text

```
<movies>
  <producer ID="123">&cp;</producer>
  <movie publisher="123"
    rating="PG">
    <title>Man in Black</title>
  </movie>
</movies>
```

DTD / Namespaces

- DTD do not understand Namespaces
 - Namespaces younger than DTDs
 - Attribute xmlns:my, xmlns:yours and prefixes in elements are unknown
- DTD must provide attributes and prefixes

```
<movies xmlns:my="http://www.fh-karlsruhe.de/movies"
        xmlns:yours="http://fh-karlsruhe.de/movies">
  <movie>
    <title>Man in Black</title>
    <my:rating>3</my:rating>
    <yours:rating>Average</yours:rating>
    <rating>PG</rating>
  </movie>
</movies>
```

DTD / Namespaces

```
<!ELEMENT movies (movie)>
<!ELEMENT movie (title, my:rating, yours:rating)>
<!ATTLIST movies xmlns:my      CDATA #IMPLIED
                 xmlns:yours  CDATA #IMPLIED>
<!ELEMENT my:rating      #PCDATA>
<!ELEMENT yours:rating  #PCDATA>
```

DTD (fragment)

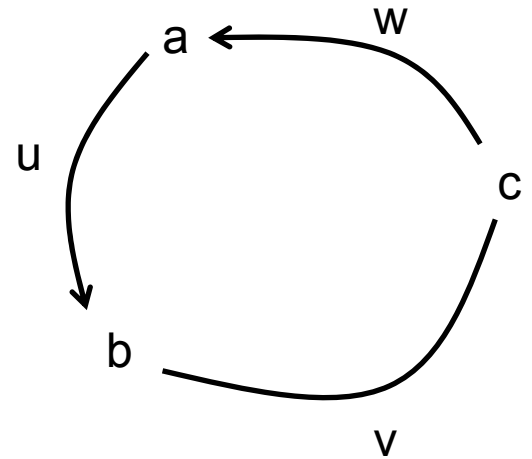
```
<movies xmlns:my="http://www.fh-karlsruhe.de/movies"
        xmlns:yours="http://fh-karlsruhe.de/movies">
  <movie>
    <title>Man in Black</title>
    <my:rating>3</my:rating>
    <yours:rating>Average</yours:rating>
    <rating>PG</rating>
  </movie>
</movies>
```

DTD / Example - GraphML

GraphML powerful yet easy to use format to represent arbitrary graphs.

- Graphs (element graph) are specified as lists of nodes and edges.
- Edges point from source to target.
- Nodes and edges may be annotated using arbitrary descriptions and data.
- Edges may be directed (and attribute edgedefault of graph).
- Edges may be attached to nodes at specific ports (north, west, ...).

```
<graphml>
  <graph edgedefault="directed">
    <node id="a"/>
    <node id="b"/>
    <node id="c"/>
    <edge id="u" source="a" target="b">
    <edge id="v" source="b" target="c"
      directed="false"/>
    <edge id="w" source="c" target="a"/>
  </graph>
</graphml>
```



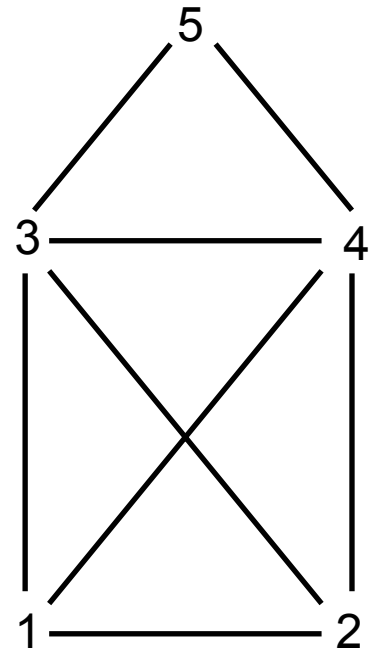
DTD / Example – GraphML DTD (1/2)

```
<!ELEMENT graphml ((desc)?,(key)*,((data)|(graph))*)>
<!ELEMENT locator EMPTY>
<!ATTLIST locator
  xmlns:xlink CDATA #FIXED "http://www.w3.org/TR/2000/PR-xlink-20001220/"
  xlink:href CDATA #REQUIRED
  xlink:type (simple) #FIXED "simple">
<!ELEMENT desc (#PCDATA)>
<!ELEMENT graph ((desc)?,((((data)|(node)|(edge)|(hyperedge))*|(locator))))>
<!ATTLIST graph id ID #IMPLIED
  edgedefault (directed|undirected) #REQUIRED>
<!ELEMENT node (desc?,(((data|port)*,graph?)|locator))>
<!ATTLIST node id ID #REQUIRED>
<!ELEMENT port ((desc)?,((data)|(port))*)>
<!ATTLIST port name NMTOKEN #REQUIRED>
```

DTD / Example – GraphML DTD (2/2)

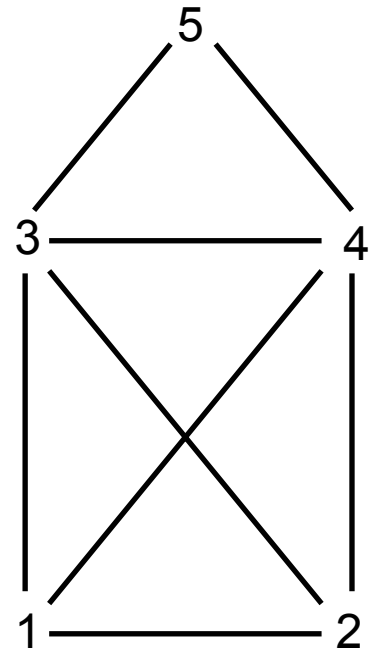
```
<!ELEMENT edge ((desc)?,(data)*,(graph)?)>
<!ATTLIST edge id ID #IMPLIED
               source IDREF #REQUIRED
               sourceport NMTOKEN #IMPLIED
               target IDREF #REQUIRED
               targetport NMTOKEN #IMPLIED
               directed (true|false) #IMPLIED>
<!ELEMENT hyperedge ((desc)?,((data)|(endpoint))*,(graph)?)>
<!ATTLIST hyperedge id ID #IMPLIED>
<!ELEMENT endpoint ((desc)?)>
<!ATTLIST endpoint id ID #IMPLIED
                 node IDREF #REQUIRED
                 port NMTOKEN #IMPLIED
                 type (in|out|undir) "undir">
<!ELEMENT key (#PCDATA)>
<!ATTLIST key id ID #REQUIRED
            for (graph|node|edge|hyperedge|port|endpoint|all) "all">
<!ELEMENT data (#PCDATA)>
<!ATTLIST data key IDREF #REQUIRED
            id ID #IMPLIED>
```

DTD / Example - GraphML



DTD / Example - GraphML

```
<graphml>
  <graph edgedefault="undirected">
    <node id="1"/>
    <node id="2"/>
    <node id="3"/>
    <node id="4"/>
    <node id="5"/>
    <edge source="1" target="2"/>
    <edge source="1" target="4"/>
    <edge source="1" target="3"/>
    <edge source="2" target="3"/>
    <edge source="2" target="4"/>
    <edge source="3" target="4"/>
    <edge source="3" target="5"/>
    <edge source="4" target="5"/>
  </graph>
</graphml>
```



Content

- Overview
- Document Type Definitions (DTD)
- **XML Schema**
- Modeling

XML Schema

- DTD's limitations
 - Not itself in XML Syntax
 - Does not “understand” namespaces
 - No other type than “Strings”: (P)CDATA
- XML Schema
 - Is a XML dialect itself
 - Support for namespaces
 - **More types and ability to build own types**
- Extensible type systems allows far reaching automated processing of XML documents
 - Rigor automated checking of document content
 - Automated conversion of XML document to – for instance – Java Object and vice versa

XML Schema / Hello World Example

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="hello-world" type="xsd:string"/>
</xsd:schema>
```

XML Schema

```
<!ELEMENT hello-world (#PCDATA)>
```

DTD

- XML schema starts with `xsd:schema`
 - Namespace `xsd` used
- Elements declared with `xsd:element`
 - Attribute for name and for the type of the element's content

```
<?xml version="1.0"?>
<hello-world xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="helloWorld.xsd">
  hello world
</hello-world>
```

Valid XML

- XML used `xsd` namespace for Schema **instances**
 - Schema included via `xsi:noNamespaceSchemaLocation=URI`

XML Schema / Example

XML Schema

```

<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:complexType name="movie-type">
    <xsd:sequence>
      <xsd:element name="title" type="xsd:string"/>
      <xsd:element name="rating" type="xsd:int"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:element name="movies">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="movie" type="movie-type"
          maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>

```

Type
declaration

Element
definition

Anonymous
type decl.

Element
definition

```

<!ELEMENT movies (movie+)>
<!ELEMENT movie (title, rating)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT rating (#PCDATA)>

```

DTD

XML Schema / Example

```
<?xml version="1.0"?>
<movies xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="02_movies.xsd">
  <movie>
    <title>Die Hard</title>
    <rating>1</rating>
  </movie>
</movies>
```

Valid

Invalid ("Good" is not of type int)

```
<?xml version="1.0"?>
<movies xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="02_movies.xsd">
  <movie>
    <title>Die Hard</title>
    <rating>Good</rating>
  </movie>
</movies>
```

XML Schema / Extension

- Complex types have Complex content
- Complex content: sequence of elements, nested elements, ...

```
<xsd:complexType name="person-type">
  <xsd:complexContent>
    <xsd:sequence>
      <xsd:element name="name"
        type="xsd:normalizedString"/>
      <xsd:element name="birthDate"
        type="xsd:date"/>
    </xsd:sequence>
  </xsd:complexContent>
</xsd:complexType>
```

- `complexContent` / `simpleContent` **optional** inner element of `complexType` if no inheritance/restriction is used

- Simple types have simple content
- Simple content: strings, numbers, ...

```
<xsd:simpleType name="person-type">
  <xsd:simpleContent>
    <!-- def. of simple type (later) -->
  </xsd:simpleContent>
</xsd:simpleType>
```

- `simpleContent` **optional** inner element of `simpleType` if no restriction is used

XML Schema / Types

- Atomic types
 - Build in. integer, boolean, date, and more
 - Can be used directly with declaration of elements
- Simple types
 - Derived from atomic types
 - Can be restricted to a set of values, e.g. by reg.-expr.
- Complex types
 - Contain elements and attributes
 - Build from simple types
- Each element has a type for its content
 - Complex type of content consists of elements
 - Simple type if content is pure data

XML Schema / Atomic types

| Type name | Semantics |
|---|--|
| anyType, anyURI | Any simple or complex type, any URI |
| string | Character string |
| normalizedString | String without carriage return, line feed, or tabs |
| long, int, short, byte unsignedLong, ... | 64, 32, 16, 8 bit integer (like in Java) non-negative long, ... |
| decimal, float, double | Arbitrary precision, IEEE-32, 64 Bit floating point |
| boolean | true, false |
| dateTime date Time | ISO coded date and time: 1971-11-08T10:00:00 Format: YYYY-DD-MM Format: HH:MM:SS |
| ID, IDREF, ENTITY | As defined in by DTD |

XML Schema / Atomic types

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:complexType name="person-type">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:normalizedString"/>
      <xsd:element name="birthDate" type="xsd:date"/>
      <xsd:element name="salary" type="xsd:decimal"/>
      <xsd:element name="comment" type="xsd:string"/>
      <xsd:element name="male" type="xsd:boolean"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:element name="person" type="person-type"/>
</xsd:schema>
```

XML Schema
"person.xsd"

```
<person xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="person.xsd">
  <name>Mr. X</name>
  <birthDate>1958-06-13</birthDate>
  <salary>2500.00</salary>
  <comment>This person does not
    really exists</comment>
  <male>>true</male>
</person>
```

XML Schema / Simple types

- Simple types are declared stand-alone
- (optional) restriction tag for restriction values
 - attribute base for defining the base type
 - multiple enumeration-elements for defining all possible values of the new type
- country-code can be use as a new type

```
<xsd:simpleType name="country-code">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="DE"/>
    <xsd:enumeration value="EN"/>
    <xsd:enumeration value="FR"/>
  </xsd:restriction>
</xsd:simpleType>
<xsd:element name="country" type="country-code"/>
```

XML Schema / Simple types

- Restrictions in XML Schema are called *facets*
- Most common facets for string types:

| Tag name | Semantics |
|-------------|---|
| enumeration | Limits the set of values to those in the enumerations |
| length | Forces a string to the given length |
| minLength | Sets the minimal length of a string |
| maxLength | Sets the maximum length of a string |
| pattern | Limits the values of a string to a regular expression r^* zero or more times a r^+ one or more times a $r^?$ zero or one times a (a is optional) $[a-z]$ all character from “a” to “z” $r_1 r_2$ either r_1 or r_2 $r\{n,m\}$ n- up to m-times r r all characters except r (r a concatenation of chars) |

XML Schema / Simple types

```
<xsd:element name="customer-identification">
  <xsd:simpleType>
    <xsd:restriction base="xsd:string">
      <xsd:length value="8"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element>
```

Valid

```
<customer-identification>no-12345</customer-identification>
```

```
<customer-identification>
  too-many-characters
</customer-identification>
```

Invalid (more than 8 characters)
Leading spaces count, too!

XML Schema / Simple types – Pattern

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:simpleType name="string-decimal">
    <xsd:restriction base="xsd:string">
      <xsd:pattern value="(\+|-)?[0-9]+(\.[0-9]+)?"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

- Escape symbols \+ for special character “+”
- The dot (.) matches any character, escape with \.
- Valid matches: “-123.45”, “12”, “12.0”

XML Schema / Simple types - Pattern

| Escape Character | Semantics |
|-------------------------|-----------------------------|
| \n | the newline character (#xA) |
| \r | the return character (#xD) |
| \t | the tab character (#x9) |
| \\ | “\” |
| \ | “ ” |
| \. | “.” |
| \- | “-” |
| \^ | “^” |
| \?, \+, * | “?”, “+”, “*” |
| \[, \], \(\, \), \{, \} | “]”, etc. |

XML Schema / Simple types – Pattern

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:simpleType name="string-decimal">
    <xsd:restriction base="xsd:string">
      <xsd:pattern value="(\+|-)?[0-9]+(\.[0-9]+)?"/>
      <xsd:length value="8"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

- Facets can be combined
- Only “decimal numbers” with eight characters are valid

XML Schema / Complex types

```
<!ELEMENT address-book (entry+)>
<!ELEMENT entry        (name, address*, comment?)>
<!ELEMENT address      (street, location)>
```

- Complex types can be sequences of elements, attributes, simple or complex content
- Control of repetitiveness with min/maxOccurs

XML Schema / Complex types

```

<!ELEMENT address-book (entry+)>
<!ELEMENT entry        (name, address*, comment?)>
<!ELEMENT address      (street, location)>

```

- Complex types can be sequences of elements, attributes, simple or complex content
- Control of repetitiveness with min/maxOccurs

```

<xsd:complexType name="entry-type">
  <xsd:sequence>
    <xsd:element name="name"      type="xsd:string"
                 minOccurs="1"   maxOccurs="1" />
    <xsd:element name="address"  type="xsd:address-type"
                 maxOccurs="unbound" />
    <xsd:element name="comment"  type="xsd:string"
                 minOccurs="0" />
  </xsd:sequence>
</xsd:complexType>

```

Default

another complex-type

XML Schema / Complex types

| MinOccurs | MaxOccurs | Semantics | DTD equivalent |
|-----------|-----------|---|-----------------|
| 1 | 1 | Element must occur exactly once (default) | (element) |
| 0 | 1 | Zero or once | (element?) |
| 0 | unbounded | Zero or more | (element*) |
| 1 | unbounded | Once or more | (element+) |
| 2 | 4 | Two to four times | not applicable? |

XML Schema / Complex types

Complex types can be anonymous

Content of an element declaration, omit name of type

```

<xsd:element name="entry">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="address" maxOccurs="unbound">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="street" type="xsd:string"/>
            <xsd:element name="location" type="xsd:string"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="comment" type="xsd:string" minOccurs="0"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
  <!--ELEMENT address-book (entry+)-->
  <!--ELEMENT entry (name, address*, comment?)-->
  <!--ELEMENT address (street, location)-->

```

XML Schema / Complex types

xsd:sequence Declared elements are ordered in sequence

xsd:any Declared elements are unordered
Must be top-most in a content-model and constituents must be elements

xsd:choice One of the declared elements had to be chosen

```
<xsd:element name="address">
```

```
  <xsd:complexType>
```

```
    <xsd:any>
```

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

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

```
    </xsd:any>
```

```
  </xsd:complexType>
```

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

```
<address>
  <street>Moltkestr. 30</street>
  <location>Karlsruhe</location>
</address>
```

Both are valid

```
<address>
  <location>Karlsruhe</location>
  <street>Moltkestr. 30</street>
</address>
```

XML Schema / Complex types

xsd:sequence Declared elements are ordered in sequence

xsd:all Declared elements are unordered
Must be top-most in a content-model and constituents must be elements

xsd:choice One of the declared elements had to be chosen

```
<xsd:element name="address">
  <xsd:complexType>
    <xsd:choice>
      <xsd:element name="street"
        type="xsd:string"/>
      <xsd:element name="location"
        type="xsd:string"/>
    </xsd:choice>
  </xsd:complexType>
</xsd:element>
```

```
<address>
  <street>Moltkestr. 30</street>
</address>
```

```
<address>
  <location>Karlsruhe</location>
</address>
```

Both are valid

```
<address>
  <street>Moltkestr. 30</street>
  <location>Karlsruhe</location>
</address>
```

invalid

XML Schema / Extending types

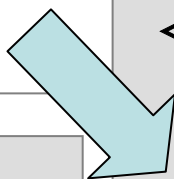
- Types (atomic, simple, complex) can be **extended** to new types
- Additional complexContent / simpleContent element necessary

```
<xsd:complexType name="address-type">
  <xsd:all>
    <xsd:element name="street"
      type="xsd:string"/>
    <xsd:element name="location"
      type="xsd:s..."/>
  </xsd:all>
</xsd:complexType>
```

```
<address>
  <location>...</location>
  <street>...</street>
</address>
```

```
<xsd:complexType name="address-full">
  <xsd:complexContent>
    <xsd:extension base="address-type">
      <xsd:all>
        <xsd:element name="house-number"
          type="xsd:string"/>
        <xsd:element name="zip-code"
          type="xsd:string"/>
      </xsd:all>
    </xsd:extension/>
  </xsd:complexContent>
</xsd:complexType>
```

```
<other-address>
  <zip-code>...</zip-code>
  <location>...</location>
  <street>...</street>
  <house-number/>
</other-address>
```



XML Schema / Attributes

Attributes declared with **xsd:attribute** within the type definition and the name of attribute, its type, use and value

```
<address preferred="true">  
  <location>...</location>  
  <street>...</street>  
</address>
```

XML Schema / Attributes

Attributes declared with **xsd:attribute** within the type definition and the name of attribute, its type, use and value

```
<xsd:complexType name="address-type">
  <xsd:all>
    <xsd:element name="street"
      type="xsd:string"/>
    <xsd:element name="location"
      type="xsd:string"/>
  </xsd:all>
  <xsd:attribute name="preferred"
    type="xsd:boolean"
    use="default"
    value="false"/>
</xsd:complexType>
```

```
<address preferred="true">
  <location>...</location>
  <street>...</street>
</address>
```

XML Schema / Attributes

| Attribute | optional | Semantics |
|--|----------|--|
| name="aName" | no | Name of the attribute unique among the type/element |
| type="aTyp" | no | a simple type |
| use="optional" use="required" use="default" value="v" use="fixed" value="v" | yes | zero or once exactly once zero or once, with given default value exactly once, with fixed given value |

```
<attribute name="id" type="xsd:ID" use="required"/>  
<attribute name="age" type="xsd:string" use="optional"/>  
<attribute name="language" type="xsd:string" use="default"  
value="de"/>
```

XML Schema / Namespaces

- DTD
 - No support of XML namespaces
 - Prefix is direct part of element name in DTD
 - Changing prefix results in change of whole DTD
 - One defined, prefixes have to be used in document instances
- XML Schemas
 - Prefix declaration separated from element and attribute names: easier to change namespace
 - Only root element need to be qualified in document instance

XML Schema / Namespaces

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:mv="http://www.fh-karlsruhe.de/movies"
            targetNamespace="http://www.fh-karlsruhe.de/movies"
            elementFormDefault="unqualified"
            attributeFormDefault="unqualified">
  <xsd:complexType name="movie-type">
    <xsd:sequence>
      <xsd:element name="title" type="xsd:string"/>
      <xsd:element name="rating" type="xsd:int"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:element name="movies">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="movie" type="movie-type"
                    maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

XML Schema / Namespace

- Global elements (in particular the root element) must **always** be qualified with a namespace prefix
- Local elements (attributes)
 - may or may not be qualified with a namespace prefix, if `elementFormDefault` (`attributeFormDefault`) is set to “unqualified”
 - Must always be qualified, otherwise

XML Schema / Namespaces

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:mv="http://www.fh-karlsruhe.de/movies"
            targetNamespace="http://www.fh-karlsruhe.de/movies"
            elementFormDefault="unqualified"
            attributeFormDefault="unqualified">
  <xsd:complexType name="movie-type">
    <xsd:sequence>
      <xsd:element name="title" type="xsd:string"/>
      <xsd:element name="rating" type="xsd:int"/>
    </xsd:sequence>
  </xsd:complexType>
```

```
<?xml version="1.0"?>
<mv:movies xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xmlns:mv="http://www.fh-karlsruhe.de/movies"
           xsi:schemaLocation="02_movies.xsd">
  <movie>
    <title>Die Hard</title>
    <rating>1</rating>
  </movie>
</mv:movies>
```

Prefixes optional

Modeling / Comparison

| XML Schema | DTD |
|---|---|
| <ul style="list-style-type: none">•to? elaborated, more difficult to use | <ul style="list-style-type: none">•one type only•new syntax•no type system |
| <ul style="list-style-type: none">•XML Syntax•XSD definition controls root element | <ul style="list-style-type: none">•Elements are ordered•Document controls root element |
| <ul style="list-style-type: none">•Compatible to namespaces•More control on element content•Elements can be unordered•Elaborated type system | <ul style="list-style-type: none">•Simple to use |

Content

- Overview
- Document Type Definitions (DTD)
- XML Schema
- **Modeling**

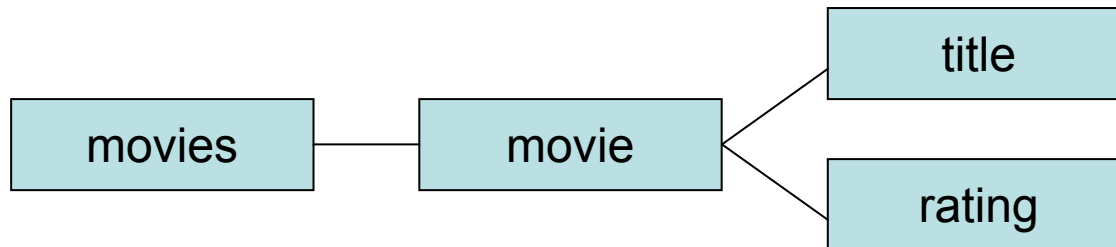
Modeling

- First step in modeling
 - Avoid modeling
 - Look for existing models
 - Learn from existing models
- In particular
 - Look for existing standard models
 - If a vendors product is used: use their XML models
 - Look up your company used XML models
 - W3C: MathML, GraphML, ebXML (for electronic commerce and business integration), etc.

Modeling

XML Model: tree-like structure

```
<!ELEMENT movies (movie+)>  
<!ELEMENT movie (title, rating)>  
<!ELEMENT title (#PCDATA)>  
<!ELEMENT rating (#PCDATA)>
```



Start with a tree of the main elements

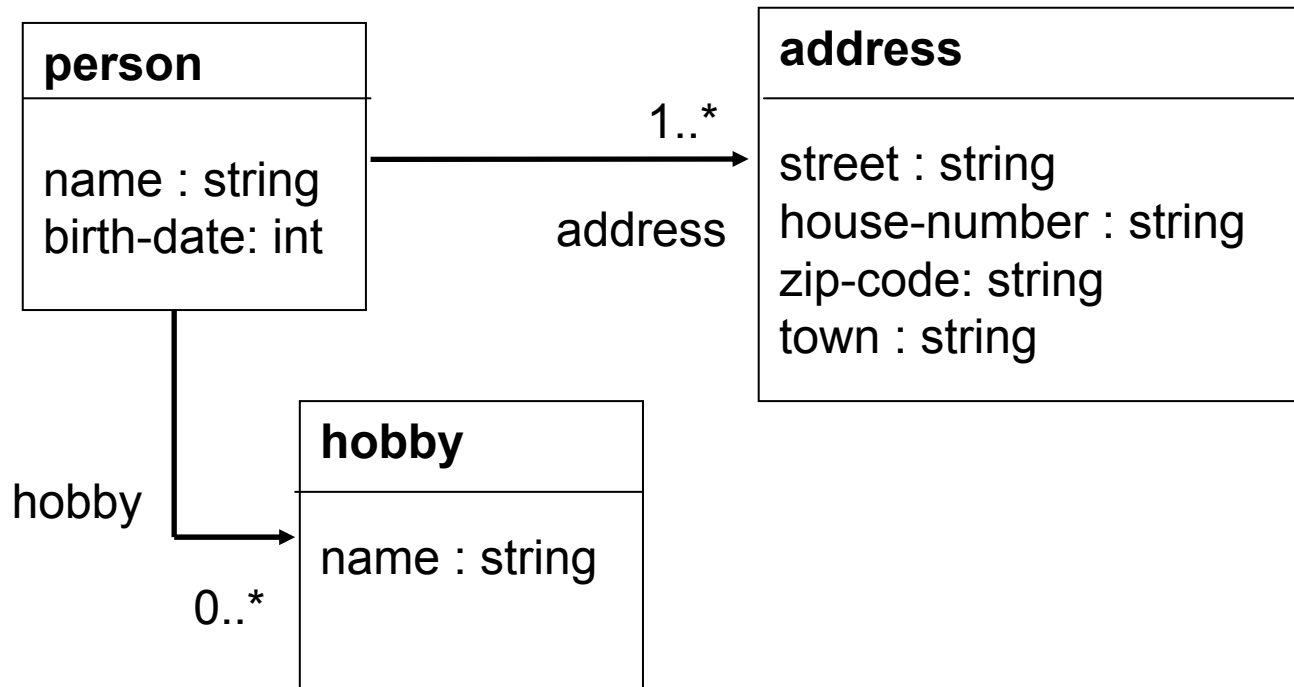
Use a graphical representation

Use a tool

Modeling / UML

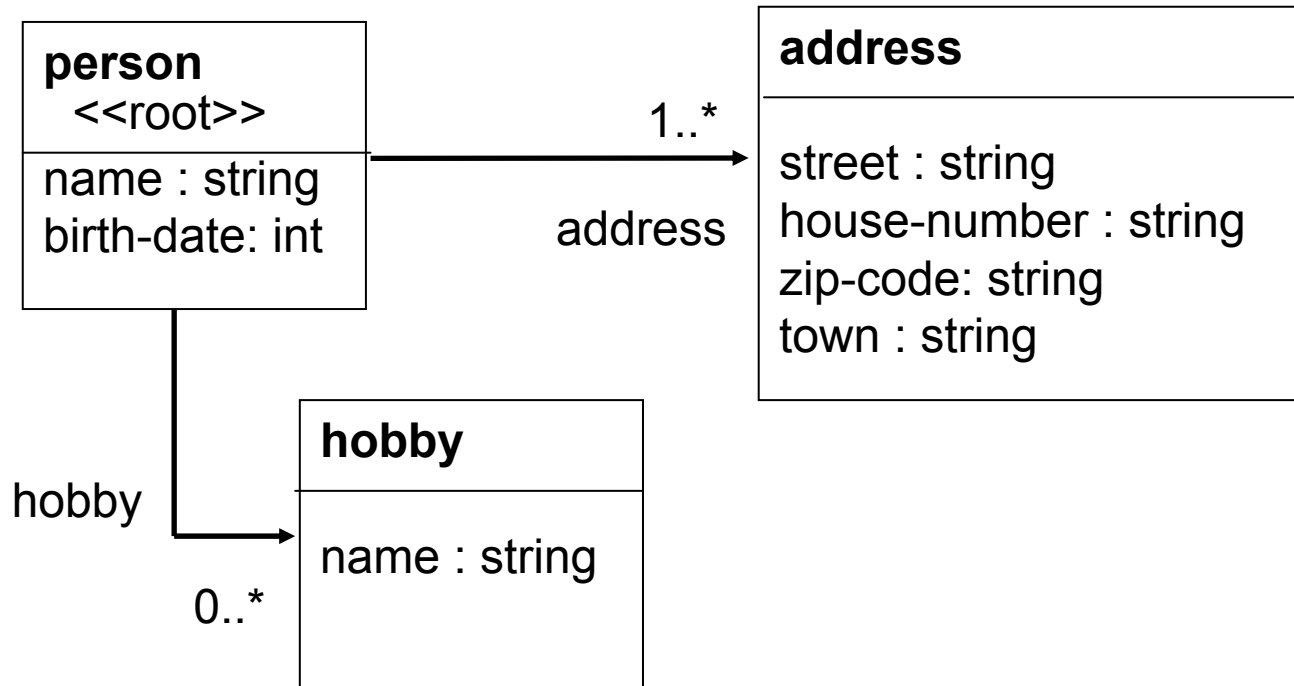
- UML (Unified Modeling Language)
- Unified? Then why not use UML for modeling XML?
 - UML is graphical
 - Trees can be drawn with UML (class) diagrams
 - UML is extensible (to a certain degree)
- There exists an official UML representation of (low level) XML

Model / From UML to XML



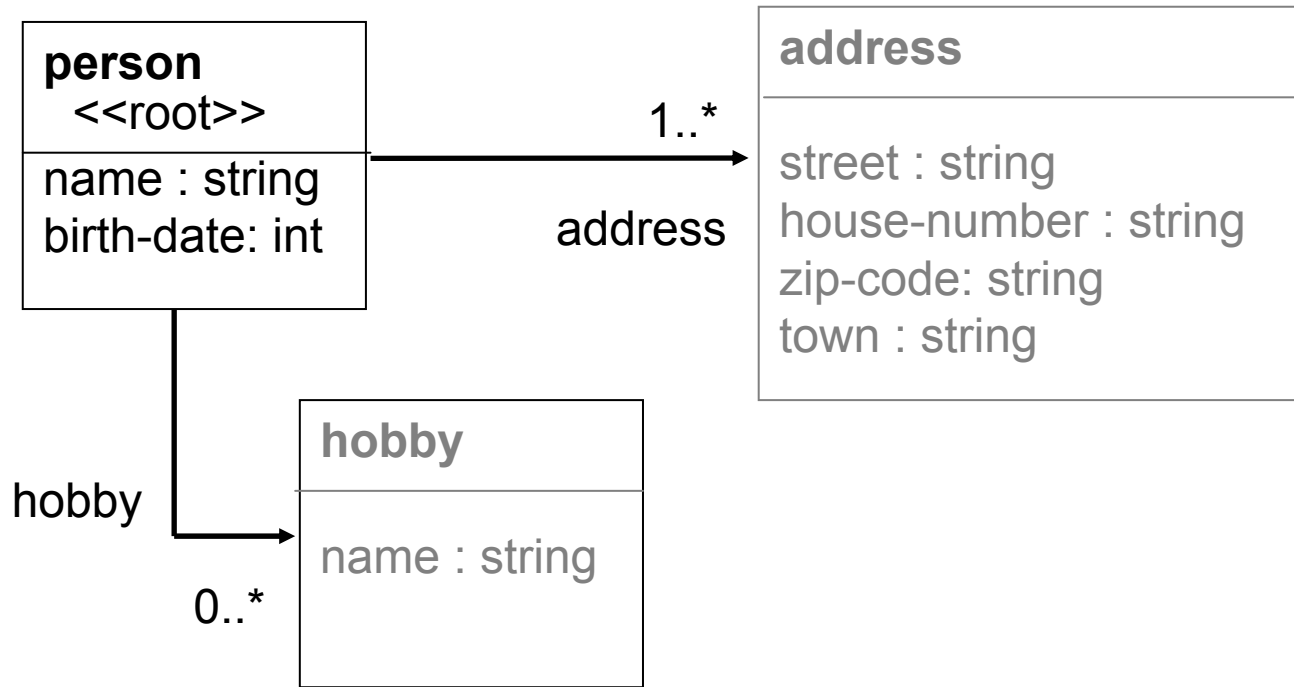
- Atomic types: Use them directly
- Classes: Complex-type elements
- Attributes / Relations: Simple-type elements
- Use directed relations when possible; (ID, IDREF otherwise)
- Use a naming convention of XML

Model / From UML to XML



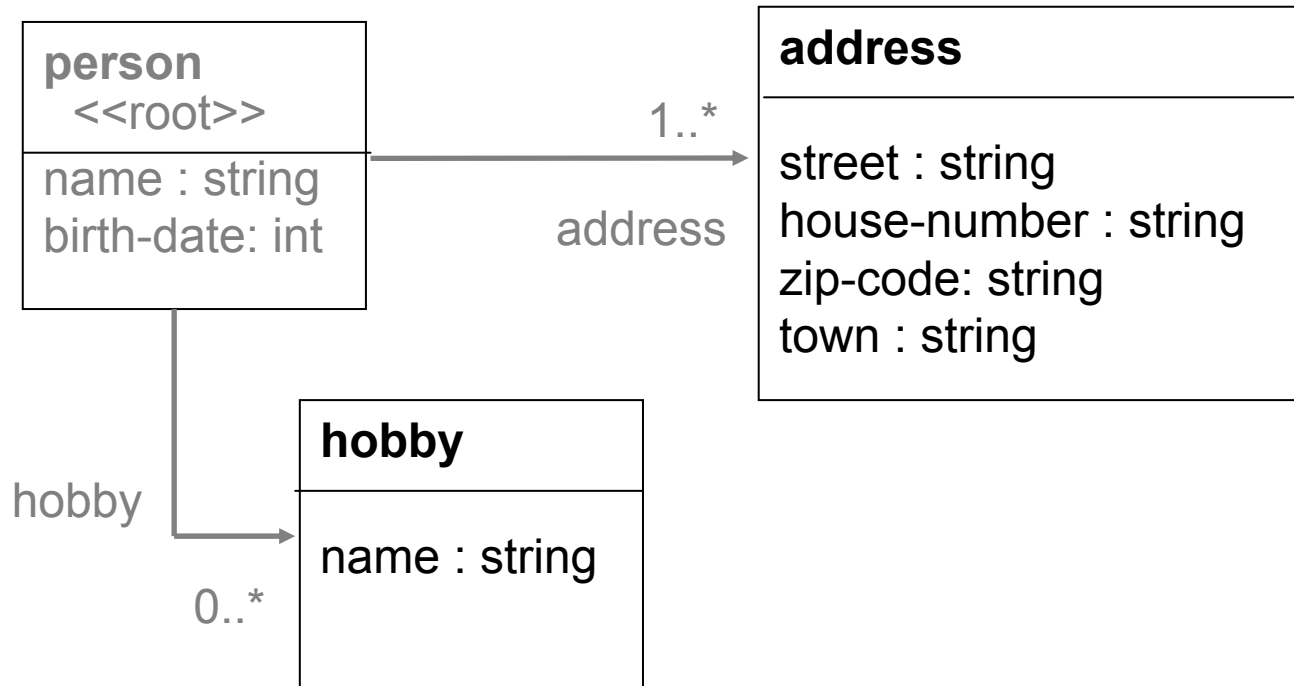
- Use UML **stereotypes** for denoting a root (or top level) element
- Find a (sub)tree with the chosen root (in this example it already is a tree)

Model / From UML to XML



```
<!ELEMENT person (name, birth-date, address+, hobby*)>
```

Model / From UML to XML

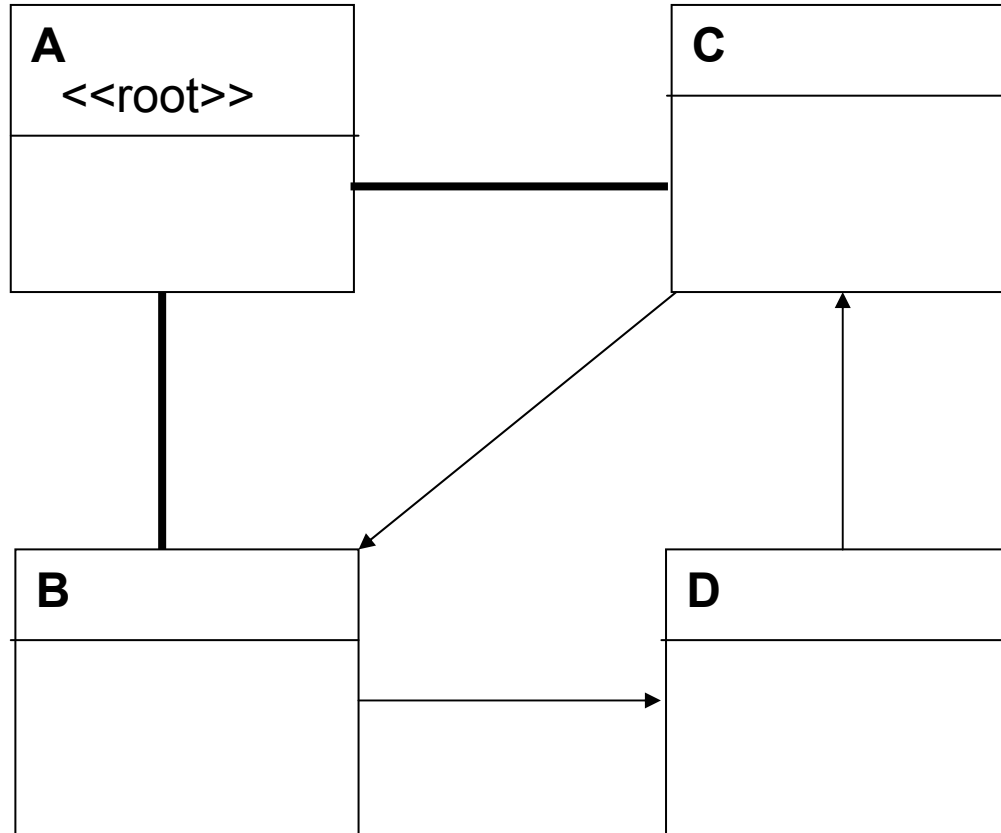


```
<!ELEMENT person (name, birth-date, address+, hobby*)>
<!ELEMENT hobby (name)>
<!ELEMENT address (street, house-number,
                    zip-code, town)>
```

Model / From UML to XML

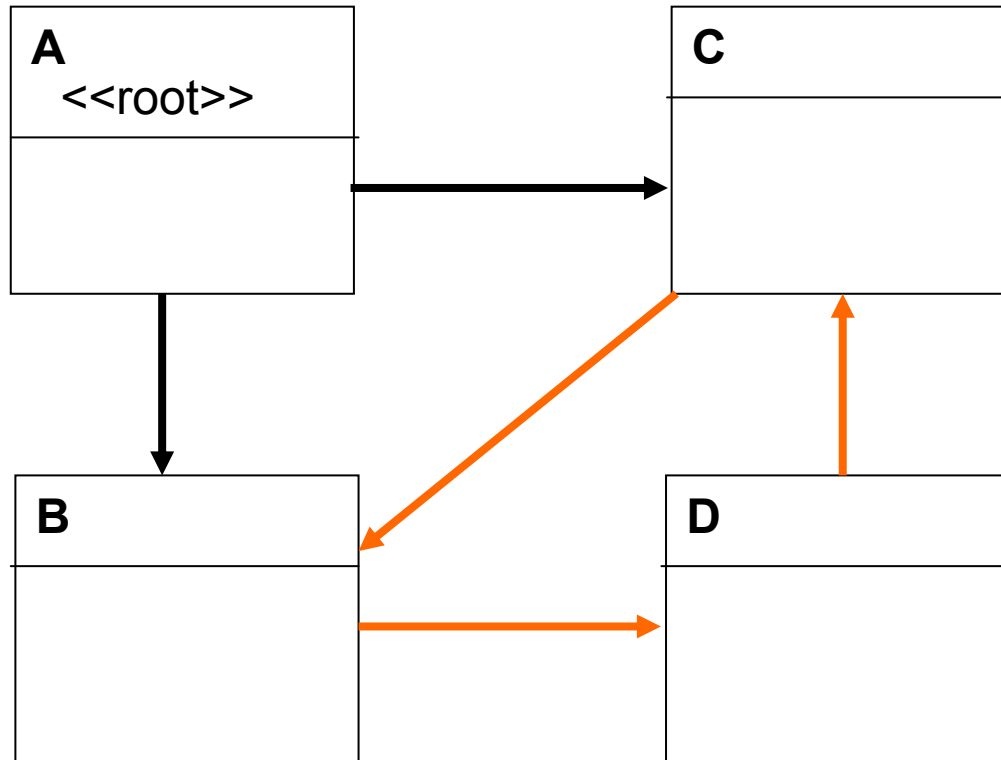
```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="person">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="name" type="xsd:string"/>
        <xsd:element name="birth-date" type="xsd:date"/>
        <xsd:element name="hobby">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="name" type="xsd:string"/>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
        <xsd:element name="address">
          ...
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Model / From UML to XML



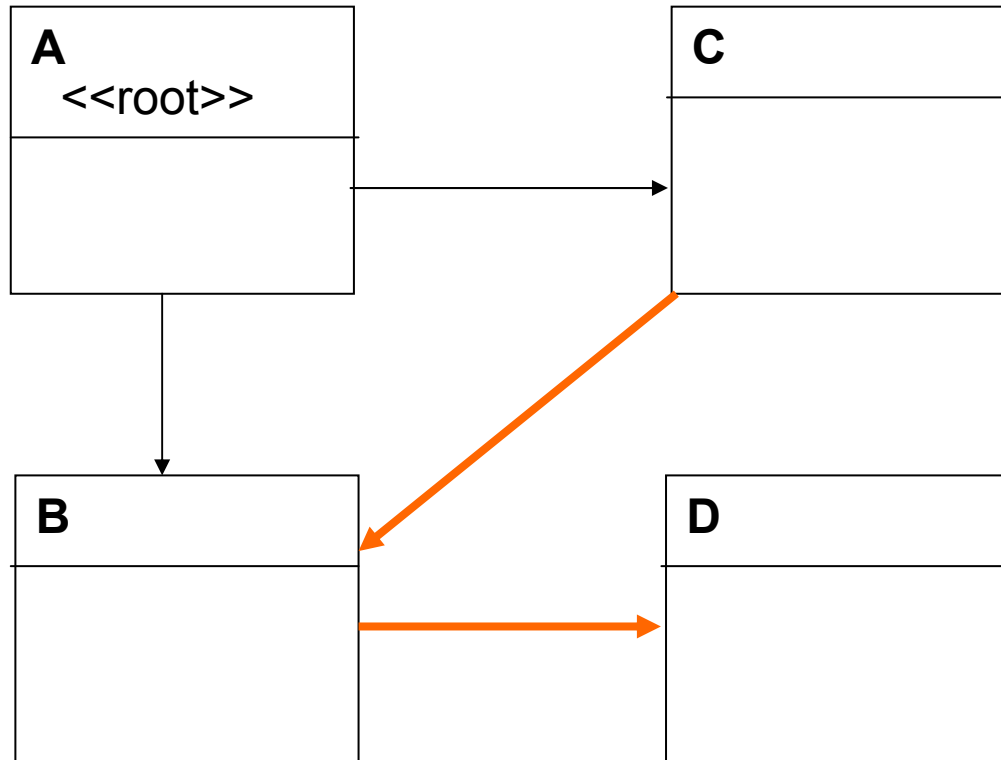
- Directing all undirected relationships without producing cycles
- Cycles?

Model / From UML to XML



- Break Cycles by:
 - Omitting “unimportant” relationships
 - If nothing more is unimportant use attributes with ID, IDREF and model relationships with links

Model / From UML to XML



- Break Cycles by:
 - Omitting “unimportant” relationships
 - If nothing more is unimportant use attributes with ID, IDREF and model relationships with links

Modeling / From UML to XML

- Advantage
 - Existing UML tool can be used
 - Often XML data has to be created from programs: use existing UML class diagram as basis for XML modeling
 - Late decision whether to use DTD or XML Schema
 - XML Model can be automatically created by UML tool
- Disadvantage
 - Addition model to maintain
 - Additional UML “dialect” to learn
- Recommendation
 - Use your XML editors graphical notation, if possible; use UML only on high level, only
 - If no tool available: use flipcharts or chalk for graphical modeling

Modeling

- Choose a naming convention
- Identify “main” elements (nodes of the tree)
 - Similar to choosing classes
- Choosing the root
 - Depends on the application
- Choose less important elements
 - Similar to identifying attributes of classes
 - assign them to a “main” element
- Attributes
 - Avoid
 - Choose if unimportant elements contains technical data
- Functionality (methods) plays no role in XML
- Do not use “values” as element/attribute names (e.g., <red/>, <blue/>, red is a color: <color>red</color>)