Modelling XML Applications (part 2)

Patryk Czarnik

XML and Applications 2015/2016
Lecture 3 – 14.03.2016

Common design decisions
Natural language

- Which natural language to use?
  - It would be a nonsense not to use English in solutions that have a potential to be used worldwide.
  - But XML supports internationalisation. You may choose to honour you native language if the solution is dedicated for a particular country.
    - Law acts processed in Polish Parliament use Polish tags,
    - as well as XML-version forms for tax declarations.

```
<Podmiot1 poz="P_1A" rola="Podatnik">
    <etd:OsobaFizyczna>
        <etd:PESEL>00000000000</etd:PESEL>
        <etd:ImiePierwsze>a</etd:ImiePierwsze>
        <etd:Nazwisko>a</etd:Nazwisko>
        <etd:DataUrodzenia>1900-01-01</etd:DataUrodzenia>
    </etd:OsobaFizyczna>
```

Common design decisions
Element or attribute?

- Where should a field be written:
  - in an attribute

```
<teacher tel="5544458">
   <first-name>Patryk</first-name>
   <surname>Czarnik</surname>
</teacher>
```

  - or in an element?

```
<teacher>
   <first-name>Patryk</first-name>
   <surname>Czarnik</surname>
   <tel>5544458</tel>
</teacher>
```

- I would write this particular one in an element.

```
<tel type="office">55<internal>44458</internal></tel>
```

Common design decisions
Element or attribute?

- Advantages of attributes:
  - more compact syntax
  - (only in DTD) some features available only for attributes
- Technical restrictions of attributes:
  - only text, without structure marked up
  - multiple attributes with the same name forbidden
- General hints
  - Semantic hint: Use elements for **data**, attributes for **metadata** (whatever it means in you case ;)).
  - Presentational hint: If you had to print your document on paper, which parts of text would you print literally (they are elements) and which parts would only have some impact (or no impact) on the way things are presented (they should become attributes)?

Common design decisions
Names

- How descriptive (and long) should a name be?
- To use multipart names, or assume that the context is known?

```
<program>
   <item>Introduction to XML</item>
   <item>XML Schema</item>
</program>
```

```
<lecture-program>
   <lecture-program-item>Introduction to XML</lecture-program-item>
   <lecture-program-item>XML Schema</lecture-program-item>
</program>
```

  - Unique element names simplify some kinds of document processing
    - CSS, SAX, or DOM (in some cases) – substantially
    - XPath/XSLT/XQuery – not so important

Common design decisions
Wrappers

```
<group>
   <number>1</number>
   <students>
     <student>Jan Kowalski</student>
     <student>Anna Nowakowska</student>
     ...
   </students>
</group>
```

```
<group>
   <number>1</number>
   <student>Jan Kowalski</student>
   <student>Anna Nowakowska</student>
   ...
</group>
```

Types in XML Schema – once again

- Every element and attribute has a type
  - If not specified: xs:anyType or xs:anySimpleType, resp.
- "What an element/attribute may contain"
  but also
  "How to interpret a value"

Classification of types

Types by content model

- Simple type (value of a text node or an attribute; applicable to elements and attributes)
  - atomic type
  - list
  - union
- Complex type (structure model – subelements and attributes; applicable to elements)
  - empty content
  - element content
  - mixed content
  - simple content

Classification of types

By place of definition:

- anonymous – defined locally in place of use
- named – defined globally
  - built-in – defined in XML Schema specification
  - user-defined

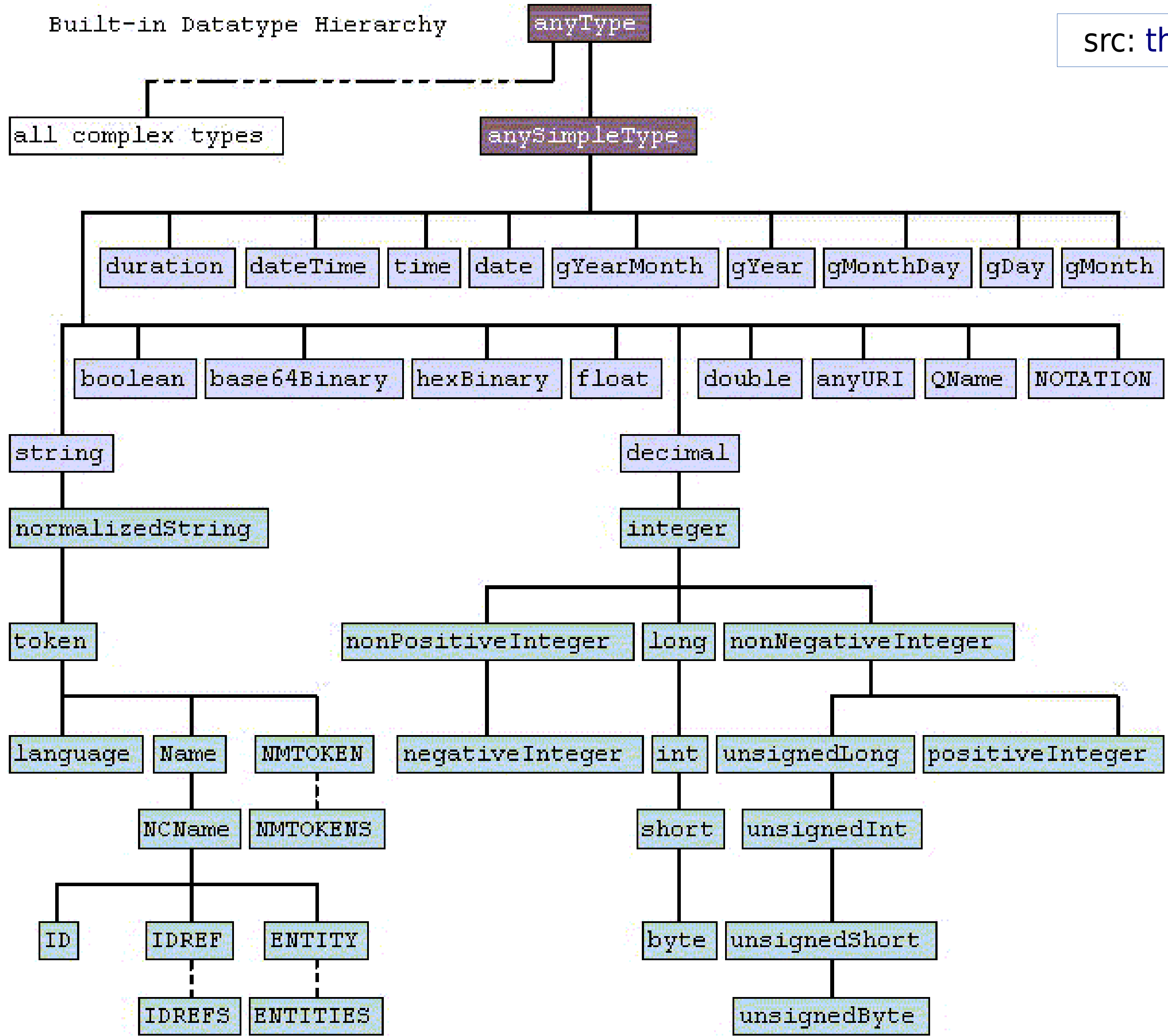
By means of definition:

- primitive (simple types)
- defined directly (complex type as a sequence etc.)
- derived (some built-in types are defined by derivation!)
  - by extension (complex types only)
  - by restriction (complex and simple types)
  - as a list or union (simple types only)

Simple types

- Rich set of built-in types
  - decimal, integer, nonNegativeInteger, long, int, ...
  - boolean, float, double
  - date, time, dateTime, duration, ...
  - string, token, base64Binary, hexBinary, ...
- Defining custom types basing on built-in types
  - by restriction
  - as a list
  - as an union

src: the recommendation



Value space vs lexical space

- A simple type specifies its
  - **value space** – set of abstract values
  - **lexical space** – set of valid text representations

Type	Text representations	Abstract value
xs:boolean	`0`, `false` `1`, `true`	False True
xs:decimal (and derivatives)	`13`, `013`, `13.00`	13
xs:string	`013` `   foo     bar     `	'013' ' foo bar '
xs:token	`   foo     bar     `	'foo bar'

Choosing the appropriate type

- Semantic meaning of a simple type:
  - not only a “set of allowed character strings”
  - also the way a value is interpreted!
- Types may affect the validation
  - e.g. leading zeros significant in strings, meaningless in numbers
- Processors may use the information about type, e.g.
  - schema-aware processing in XSLT 2.0 or XQuery
    - sorting, comparison, arithmetic operations
  - JAXB – generation of Java classes based on XSD
- Choosing the appropriate type sometimes not obvious
  - phone number, zip code, room number – number or string?

Defining simple types by restriction

- Constraining facets – properties we can restrict
  - enumeration
  - pattern
  - length, minLength, maxLength
  - totalDigits, fractionDigits
  - maxInclusive, maxExclusive
  - minInclusive, minExclusive
  - whiteSpace

Some of them available only for chosen primitive base types

- Used directly in simple type definition:

```
<xs:simpleType name="LottoNumber">
   <xs:restriction base="xs:integer">
      <xs:minInclusive value="1" />
      <xs:maxInclusive value="49"/>
   </xs:restriction>
</xs:simpleType>
```

```
<lottoNumber>12</lottoNumber>
```

List types

- List of values separated with whitespace.
- Not to confuse with sequences
  - list – simple type, no markup structure within
  - sequence – complex type, sequence of subelements
- Compact notation for lists of values

but

- Harder to process in XML processors (requires additional parsing using regexp etc. – not available e.g. in XSLT 1.0)

```
<xs:simpleType name="LottoNumberList">
   <xs:list itemType="LottoNumber" />
</xs:simpleType>
```

```
<lottoNumberList>12 2 47 6 33 12 27 18</lottoNumberList>
```

Union types

- Union of sets of values
- Possibility to mix values of different primitive types
  - Interpreting values as abstract values hard to perform
  - Nevertheless, a usable feature (e.g. unbounded in XML Schema)

```
<size>40</size>
<size>L</size>
```

```
<xs:simpleType name="ClothingSizeLetter">
  <xs:restriction base="xs:token">
    <xs:enumeration value="XS"/>
    <xs:enumeration value="S" />
    <xs:enumeration value="M" />
    <xs:enumeration value="L" />
    <xs:enumeration value="XL"/>
    <xs:enumeration value="XXL"/>
  </xs:restriction>
</xs:simpleType>
```

```
<xs:simpleType name="ClothingSizeNumber">
   <xs:restriction base="xs:integer">
      <xs:minInclusive value="20" />
      <xs:maxInclusive value="60" />
   </xs:restriction>
</xs:simpleType>
```

```
<xs:simpleType name="ClothingSize">
   <xs:union memberTypes="ClothingSizeNumber ClothingSizeLetter"/>
</xs:simpleType>
```

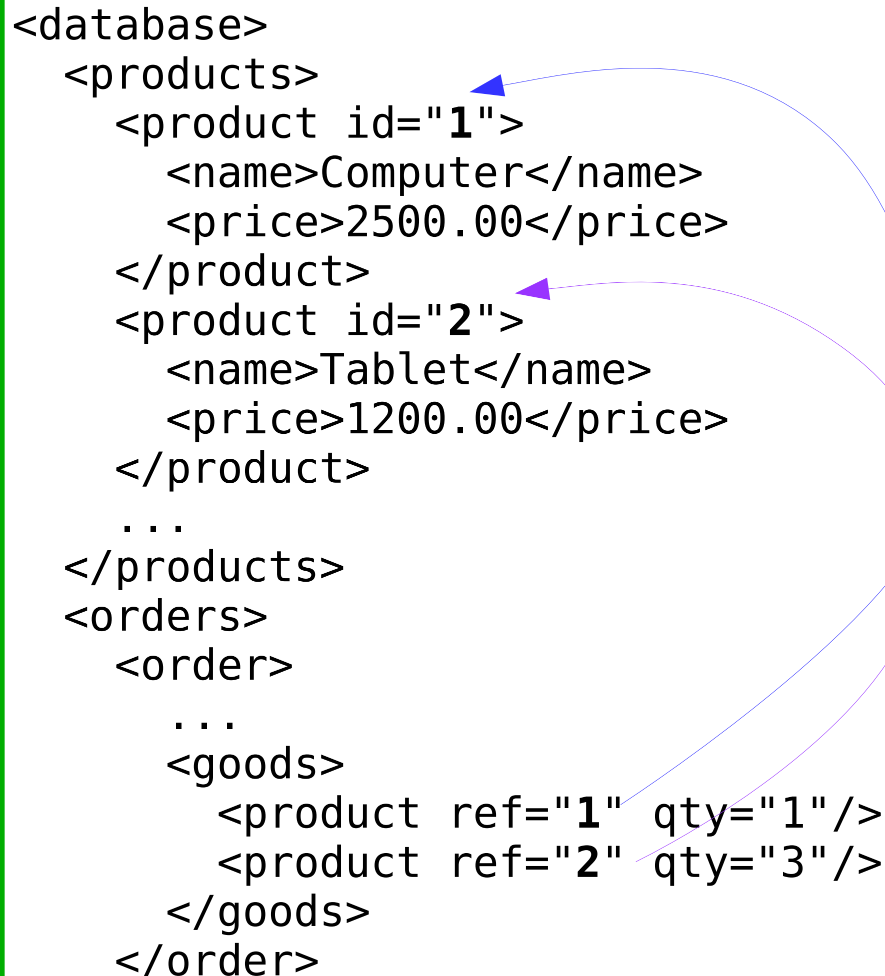
Identity constraints

- Constraints on uniqueness and references

Two mechanisms:

- DTD attribute types `ID` and `IDREF`
  - introduced in SGML DTD but still available in XML Schema
  - drawbacks:
    - one global scope, at most one ID per element
    - special form of values – only names allowed
    - IDs and references necessarily in attributes
- XML Schema identity constraints
  - `key`, `unique`, and `keyref` definitions
  - more powerful and more flexible than ID/IDREF

Global scope



- Identifiers unique within document scope
- References valid within document scope
- Expressible in DTD
  - using ID and IDREF, with some restrictions
- and XML Schema
  - using key and keyref in the root element

Global scope identifiers – DTD solution

```
DTD solution

...
<!ATTLIST product id ID #REQUIRED>

...
<!ATTLIST product-ref ref IDREF #REQUIRED>
```

```
<product id="P1">


<goods>
  <product-ref ref="P1 />
```

- Restrictions:
  - one global scope for all elements
  - only **names** may become identifiers
    - (they have to start with a letter)
  - ids must be written in attributes, not elements

XML Schema solution

XML Schema:

```
<xs:element name="database">
    ...
    <xs:key name="Products">
        <xs:selector xpath="tns:products/tns:product"/>
        <xs:field xpath="@id"/>
    </xs:key>

    <xs:keyref name="ProductsInOrders" refer="tns:Products">
        <xs:selector xpath="tns:orders/tns:order/tns:goods/tns:product"/>
        <xs:field xpath="@ref"/>
    </xs:keyref>
</xs:element>
```

XML Schema keys and references

- Other possibilities: (see *orders* example)
  - local scope constraints
  - tuples as key values
- Pay attention:
  - only simple type values can become ids
  - type of reference source and target must be the same
  - default namespace does not work in XPath expressions so we have to use a prefix (usually `tns:`)
- Using references can help to avoid data duplication in documents
  - but overuse can make XML look like a flat relational database instead of a nice structural tree
  - find the golden mean!