7. DTDs; XSD [1]

XML Foundations (IS 290A-1) - 20 September 2005

Copyright © 2005 Robert J. Glushko

Plan for Today's Class

- Limitations of DTDs
- XSD to the rescue (part 1, will finish on 22 September)
- Assignment 4: DTD {and,vs,or} XSD

Limitations of DTDs [1]

- DTDs use a syntax different from that for XML instances
  - `<!ELEMENT ElementName>` in declarations
  - `<ElementName>` in instances
- DTDs have weak datatyping – Most content gets defined as #PCDATA, which is "string" in XML-ese
- DTDs have weak occurrence constraints – 0 or 1, 0 or more, 1 or more may be sufficient to encode models of narrative documents but transactional documents require much more precision

Limitations of DTDs [2]

- A DTD doesn't understand namespaces; it defines a single document type and has no mechanisms for reusing elements declared in other document types other than by "cut and paste" where the connection to the other vocabulary is lost
- DTDs lack object-oriented modeling capabilities - you can't easily define one element to be a template or archetype for another, you can't express concepts like property inheritance, and you can't easily build libraries of DTD fragments for reuse
- DTDs treat comments (and other embedded documentation) as 2nd class citizens

Limitations of DTDs: Different Syntax from Instances

- DTDs use a syntax for declaring elements that is different from the syntax for creating elements in XML documents
- This was for compatibility with SGML, but we've long known that DTD syntax was strange
- DTDs need a different parser than the XML documents they are validating!
- Creates extra work for implementers of XML tools and programmers
- Makes it harder to learn XML
- DTDs can't even be easily stored and retrieved using XML-aware databases

## Limitations of DTDs: Weak Datatyping

- DTDs have weak datatyping – most content is defined as "PCDATA, which is "string" in datatype-ese
- This reflects XML's SGML heritage as a publishing technology, where content was primarily complex documents composed of text
- But it is a severe limitation for more data-intensive and e-business applications where a stronger type repertoire is required

## Limitations of DTDs: Weak Occurrence Constraints

- DTDs can declare that an element is:
  - ? – optional, can occur zero or one time
  - + – one or more
  - * – zero or more
- These may be adequate to encode models of narrative documents but transactional documents require much more precision
- But saying "exactly 5 times" or "between 3 and 9 times" requires contortions with DTDs

## Limitations of DTDs: No Support for Object-oriented Design [1]

- The rapid evolution in the past few years toward a document-centric architecture for distributed computing, web services, and so on, made another modeling limitation of DTDs very apparent
- Programmers looked to DTDs wanting a strongly typed, object-oriented document definition language (like database schemas or Java classes) and didn't find DTDs to be very "programmer-friendly"
- In DTDs you can't easily define one element to be a template or archetype for another, you can't model concepts like property inheritance, and you can't easily build libraries of DTD fragments for reuse
Design [2]

- Object-oriented reuse implies being able to take a standard declaration and extend it by adding additional elements or attributes, or else to refine it by restriction
  - Example: "Buyer" and "Seller" are based on "Party"
  - Example: "Student" and "Faculty" are based on "Person"
  - Example: "ShippingAddress" and "BillingAddress" are based on "Address"
  - Example: "Semester" is based on "Year"

Limitations of DTDs: Comments are "Second-Class Citizens"

- <!-- simple DTD to describe computers (used in Doc Eng lectures)  
  author: bob glushko  
  version 1.6 last revised: 6 Feb 2004   -->

```xml
<!ELEMENT Computer (OEM, ProductLine, Model, Specs?)>
<!ATTLIST Computer type (server | desktop | notebook) #IMPLIED>
<!-- price is optional and assumed to be "USD"  
 Some currencies should probably be replaced by "EUR" for "Euro" -->

<!ELEMENT Specs (Speed, MemorySize, DiskSize, Weight?, Price*)>
```

- Not clear to which declarations they apply
- Are they intended for people or for applications?
- Often impossible to extract them and process them intelligently
- Parsers are not obligated to preserve them and "pass them along" to applications

XML Schema to the rescue...

- The W3C XML Schema, aka XML Schema Definition Language (XSD or XSDL) in May 2001 wasn't the first XML schema language to address the limitations of DTDs, but because it is the W3C-recommended schema language, it is likely to survive (along with DTDs) no matter what else comes along
- When people say "XML Schema" (with a capital "S") they mean XSDL (vs "XML schema language..." which applies to the other schema languages as well)
- Strengths of XSDL
  - XSDL uses the same syntax as XML instances
  - XSDL has strong datatyping
  - XSDL has strong occurrence constraints
XSDL understands XML namespaces
XSDL supports object-oriented design
XSDL supports "literate programming"

Comparing a DTD and a Schema

- The XML instance: `product-db.xml` and an HTML visualization `product-db.htm`

- DTD

```xml
<!ELEMENT Products (Product+)>
<!ELEMENT Product (Name, PartNumber, Inventory, Cost, WholesalePrice?, RetailPrice*)>
<!ELEMENT Name (#PCDATA)>
<!ELEMENT PartNumber (#PCDATA)>
<!ELEMENT Inventory (#PCDATA)>
<!ELEMENT Cost (#PCDATA)>
<!ELEMENT WholesalePrice (#PCDATA)>
<!ELEMENT RetailPrice (#PCDATA)>
```

- The XML Schema (expressed in XSDL): `product_db.xsd`

If DTDs Are So Limited, Why Are We Still Using Them?

- XSDL will replace DTDs in many situations, especially for transactional documents
- But DTDs will probably stick around for relatively simple publication types, because the greater power of XSD comes with significantly more complexity

Modeling is NOT Schema Design

- Regardless of which schema language you use to encode your model
- Don't let the modeling restrictions of the schema language stop you from modeling
- Document what your real datatypes and constraints are, even if you can't completely validate them

XSDL Schemas Expressed in "Instance Syntax"

- Notice that the browser displayed the schema even though it won't display a DTD
- This is because XSDL schemas are expressed in "instance syntax" – they use the same markup symbols as XML instances
- That is because XDSL is itself defined as a new XML vocabulary exactly the same way that any DTD defines a new vocabulary; XSDL markup is distinguished only by virtue that it uses the "xs:" namespace
• This kind of "bootstrapping" approach to building a schema language is the key to giving it the new capabilities it needs to improve on DTDs

• It is exactly as if we've built a DTD for a new kind of DTD

---

**Declaring Elements and Attributes**

- `<Book category="fiction" publicationYear="1851">
  <Title>Moby Dick</Title>
  <Author>Melville</Author>
</Book>`

- `<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <xs:element name="Book">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Title" type="xs:string"/>
        <xs:element name="Author" type="xs:string"/>
      </xs:sequence>
      <xs:attribute name="category" type="xs:string" use="required"/>
      <xs:attribute name="publicationYear" type="xs:int" use="required"/>
    </xs:complexType>
  </xs:element>
</xs:schema>`

---

"**All Tags Have Type**" [1]

• "All tags have type" – every element declaration uses a "type" attribute to define its type

• All attributes have types, too - of the datatype sort

• The types for elements can be *anonymous* or implicit but to make them reusable they have to be given names and defined separately from the elements that refer to them

• This is comparable to defining a class and using it to create an object

---

"**All Tags Have Type**" [2]

• With named types you can have two elements that are the same type with different names but with identical content models:
  - `<xs:element name="ShippingAddress" type="AddressType"/>`
  - `<xs:element name="BillingAddress" type="AddressType"/>`

---

**Naming Elements and Attributes**

• You should follow a naming convention that distinguishes elements from attributes

• EXAMPLE: UpperCamelCase for elements and types, lowerCamelCase for attributes
The name of an element's corresponding type can be `ElementName + 'Type'` - for example, the `Course` element has a corresponding `CourseType` complex type.

Exception: when you have multiple derived elements of the same type. For example, you might have an `AddressType`, and elements for `LocalAddress`, `BillingAddress`, and `PermanentAddress`, all of which are of the `AddressType` type.

Do not use abbreviations except for "identifier" (ID) and "Uniform Resource Identifier" (URI).

---

**Simple and Complex Types**

- Elements can have simple or complex types, but all attributes must be of simple types

  - **Simple types**
    - Describe the content of a text node or attribute; (these contain content but no other elements or attributes)
    - Precise control possible over datatype and value ranges
    - The basic building blocks of the schema; many are built-in datatypes
    - Attributes, defined using very similar syntax, always have simple types

  - **Complex types**
    - Most elements are of `complexType`
    - This means they contain subelements or attributes, not just content

---

**Occurrence Constraints**

- Much more precision than enabled by ?, +, and * in DTDs
  - `minOccurs=0, maxOccurs=1` (equivalent to "?")
  - `minOccurs=0, maxOccurs=unbounded` (equivalent to "*")
  - `minOccurs=1, maxOccurs=unbounded` (equivalent to "+")
  - `minOccurs=1, maxOccurs=41` (not in a DTD without contortions)
  - `minOccurs=4, maxOccurs=11` (minOccurs and maxOccurs can be any positive integers)
  - `minOccurs=1, maxOccurs=1` (default if no values provided)

---

**Specifying How Elements are Arranged**

- `<xs:sequence>` – All elements in the group can occur in the specified order in any document instance, with precise occurrence constraints specified [EXAMPLE]
• <xs:choice> – Only one element in the group can occur in a document instance
  ○ Don’t confuse this with enumerations that restrict the valid values for an element or attribute, requiring one and only one of them

• <xs:all> – Elements can occur in any order but only once each

Repeating Elements, Lists, and Containers

• Elements with maxOccurs > 1 should be wrapped in a container element

• The container provides a logical handle for the child elements, and ensures that the nodeset returned by an XPath expression on the container is well-formed XML (not the case without the container)

• In BABL elements such as Course that can repeat, this convention would suggest a container element named Courses or ListOfCourse

• The preferred rule is to name the container as the plural of the element it contains.

Mixed Content

• <xs:complexType name="Book" mixed="true">
  <xs:choice minOccurs="0" maxOccurs="unbounded">
    <xs:element name="Title" type="xs:string"/>
    <xs:element name="Author" type="xs:string"/>
    <xs:element name="Category" type="xs:string"/>
    <xs:element name="PublicationYear" type="xs:int"/>
  </xs:choice>
</xs:complexType>

• Book
  My favorite book is <Title>Moby Dick</Title> by <Author>Melville</Author>. It is a <Category>fiction</Category> work first published in <PublicationYear>1851.</PublicationYear>
</Book>

• Sentence
  My favorite book is <Title>Moby Dick</Title> by <Author>Melville</Author>. It is a <Category>fiction</Category> work first published in <PublicationYear>1851.</PublicationYear>
</Sentence>

Element and Type Declarations

• <xs:element name="Product" type="ProductType"/>

  <xs:complexType name="ProductType">
    <xs:sequence>
      <xs:element ref="Name"/>
      <xs:element ref="PartNumber"/>
      <xs:element ref="Inventory"/>
      <xs:element ref="Cost"/>
      <xs:element ref="WholesalePrice" minOccurs="0"/>
      <xs:element ref="RetailPrice" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
Named Types

- Using a "named" type to make the element type reusable: the Product element is defined as type ProductType, whose definition (as a sequence of elements) follows separately
- The element declaration is empty - no end tag
- This style of declaration makes every type global and reusable

Anonymous Types

- Using an "anonymous" type in which the definition of the sequence as a complexType is contained in the element's definition.
- See how in this style of element declaration the <xs:element /> start and end tags wrap around everything else?
- Declaring each element inside of the element in which it appears yields a deeply nested schema with lots of local types.
- Since there is no named type to refer to, this definition can't be reused
Taxonomy of Datatypes in XML Schema

- Built-in datatypes are defined in the XSDL specification
  - Primitive datatypes
  - Derived datatypes
- User-defined datatypes
  - User-defined derived datatypes
  - All complex types

Built-In Primitive Datatypes

Built-In Derived Datatypes
Facets [1]

- A facet is a dimension on which the set of values for the datatype can be restricted in order to create a more specific datatype

- Facets for all datatypes
  - pattern (regular expression)
  - enumeration (list of possible values)

- String facets
  - length (or minLength or maxLength) in number of characters
  - whiteSpace (preserve, replace, or collapse)

Facets [2]

- Numeric facets
  - minInclusive
  - minExclusive
  - maxInclusive
  - maxExclusive
  - totalDigits
  - fractionDigits

- Date and time facets
Facets to Restrict Ranges in User-defined Derived Datatypes

- Use `<xs:restriction>` to define a type whose legal values are a subset of another one

```xml
<xs:simpleType name="Inventory">
  <xs:restriction base="xs:byte">
    <xs:minInclusive value="0"/>
    <xs:maxInclusive value="100"/>
  </xs:restriction>
</xs:simpleType>
```

Facets to Restrict Patterns in User-defined Derived Datatypes

- `<xs:restriction>` can be used to define a type whose legal values are a subset of another one

```xml
<xs:simpleType name="PartNumber">
  <xs:restriction base="xs:string">
    <xs:pattern value="[A-Z]-[0-9][0-9]?"/>
  </xs:restriction>
</xs:simpleType>
```

Enumerations and Code Lists [1]

- Enumerated datatypes or "Code Lists" should be used anywhere that a known set of expected values is allowed
- You can think of these as "controlled vocabularies" for content
- Code lists should be expected to change more frequently than the definitions of other content elements
- A code list that is used in very few contexts (which implies that you're defining it yourself) is an *internal* code list that you encode within the namespace of the schema in which the codes will be used

Enumerations and Code Lists [2]

- In contrast, a code list that is reusable is an *external* code list
  - A code list might be created and maintained by some other person or organization
  - It would be poor design to enumerate the possible values for codes like this in your schema
  - Instead of enumerating the code values, you incorporate the *definition* of the code list by reference
  - If your own code lists are reusable, you should likewise define them in a separate namespace so they can be incorporated by reference in different schemas that need them

Enumerations and Code Lists [3]
Using Refined Datatypes in "product.xsd"

- The XML Schema with more precise datatypes: product_db_refined.xsd

Readings for 22 September

- Chapter 5 of Beginning XML, 3rd Edition [pages 206-220] (second part of XML Schemas)

Assignment 4

- 10 XML instances, book1.xml -> book10.xml
- book.dtd and book.xsd
- understanding schema syntax and validation