Principles of XML design: Considering container elements

When to use elements to wrap structures of other elements

Uche Ogbuji

A critical issue in designing XML formats is figuring out how to arrange elements and represent relationships between them. Element design works best when it naturally corresponds to how people think about the concepts that each element represents. This article discusses best practices for organizing information into XML elements, focusing on the use of container elements.

In my last article in this series, I started looking at how to organize elements into structures in XML design. In this article, I continue discussion of element structures, considering the problem of when and how to use container elements around groups of related elements.

XML modeling is but a subset of data modeling. I define data modeling as any activity by which real-world phenomena are translated into abstract, formal models in order to support computer applications. A great many who are involved in XML design have backgrounds in design for computer programming. This background involves familiarity with common data structures, including tools for identity and reference of data structure instances. Some of these issues are much the same in good XML design, and some require a bit of a shift in perspective.

Wrapper elements

A very common question I've heard from XML designers is when to wrap groups of elements with a container element (also called a wrapper element). In this section I demonstrate what a container element is. Listing 1 is an inventory file for a library, which is an example of a homogeneous group of elements.
Listing 1. Example of a group of elements with no special wrapping

```xml
<library>
  <name>The XML Institute Public Library</name>
  <book isbn="0764547607">
    <title>The XML Bible, 2nd Edition</title>
  </book>
  <book isbn="0321150406">
    <title>Effective XML</title>
  </book>
  <book isbn="1861005946">
    <title>Beginning XSLT</title>
  </book>
</library>
```

The `book` elements simply appear in a series, as siblings of the `name` element. The question is whether it is better design practice to interpolate a container element, as in Listing 2.

Listing 2. Example of a group of homogeneous elements with a container

```xml
<library>
  <name>The XML Institute Public Library</name>
  <books>
    <book isbn="0764547607">
      <title>The XML Bible, 2nd Edition</title>
    </book>
    <book isbn="0321150406">
      <title>Effective XML</title>
    </book>
    <book isbn="1861005946">
      <title>Beginning XSLT</title>
    </book>
  </books>
</library>
```

The `books` element serves as a container. You may want to use a container element for a number of reasons. Containers are far more common in programming design. As an example, Listing 3 is a sort of pseudo-code data structure that corresponds to the library inventory document.

Listing 3. Example of a programming language structure based on Listing 1

```plaintext
structure library
  begin
    string name;
    list<book> books;  
  end;

structure book
  begin
    string isbn;
    string title;
  end;
```

The `library` structure comprises a reference, `name`, to an object of type `string`, and a reference, `books`, to a list object designated to hold objects of type `book`. The structure `book` is pretty straightforward. The need for the `books` reference in programming often sets programmers up to expect to need such constructs in XML design. This is not the only carry-over that I am used to seeing. Another is that many people are used to the fact that the relationship between the `book` structure and its `isbn` member is the same as that to its `title` member; these people often end up...
expressing both as either element or attribute, without considering that in XML design, the former
is better expressed as an attribute and the latter as a child element (see earlier articles in this
series for more on this).

The real world versus the code world

Take another quick look at Listing 3. These programming structures are actually very different in
nature from the element structures in Listing 1 and even Listing 2. The structure members are
actually named references to other data. The key point is that the names -- name, books, and so on
-- do not represent the actual string or list-of-book data items (called objects in OO languages),
but rather the relationship between each instance of the structure and these objects (called
associations in OO modeling). In most programming languages, you deal with object references
and the actual objects being referenced in tandem, without considering the distinction between the
two (although advanced techniques and a firm command of most languages require programmers
to thoroughly understand this distinction). Listing 4 is a very literal translation to XML that takes this
distinction into account.

Listing 4. Literal translation from the structures in Listing 3 to XML

```xml
<library>
  <name>
    <string>The XML Institute Public Library</string>
  </name>
  <books>
    <list member-type="book">
      <member index="0">
        <book isbn="0764547607">
          <title>The XML Bible, 2nd Edition</title>
        </book>
      </member>
      <member index="1">
        <book isbn="0321150406">
          <title>Effective XML</title>
        </book>
      </member>
      <member index="2">
        <book isbn="1861005946">
          <title>Beginning XSLT</title>
        </book>
      </member>
    </list>
  </books>
</library>
```

You probably know better than to be this literal when designing XML, because you know you're
modeling an actual library inventory concept from real life rather than its likely representation
in computer code. This bit of understanding is key to realizing when container elements are
appropriate (and key to many other aspects of XML design).

In XML design, avoid constructions that do not correspond to any of the real-world phenomena you
are modeling.

You can immediately apply this test to Listing 2, as the element books doesn't really correspond
to anything useful in the real-world phenomena being modeled. The library itself is an implicit
collection of books, so there is no real need for the explicit books container. As a further example, suppose the article were organized into sets of books that were collectively donated. In this case, it does make sense to add a container element, such as in Listing 5, because it corresponds to a real-world concept.

**Listing 5. Example of meaningful container elements**

```xml
<library>
    <name>The XML Institute Public Library</name>
    <endowment>
        <donor>IBM</donor>
        <book isbn="0764547607">
            <title>The XML Bible, 2nd Edition</title>
        </book>
        <book isbn="0321150406">
            <title>Effective XML</title>
        </book>
    </endowment>
    <endowment>
        <donor>W3C</donor>
        <book isbn="1861005946">
            <title>Beginning XSLT</title>
        </book>
    </endowment>
</library>
```

Notice that I prefer to use the more meaningful name "endowment", rather than just "books" to represent the element structure for a collection of books within the library.

**But what about the tools?**

Some developers always use container elements because they find it somewhat easier for their tools to process such XML. This is almost always a problem with their selection of tools, and not a good reason to introduce contrivances to the XML design. As author Eric van der Vlist noted in an e-mail thread that helped inspire this article:

> I tend to think that tools should have a limited impact on document design (of course not going to the point where the documents can't be processed at all) and that a good design isn't necessarily one which imports all the restrictions of all the tools.

Luckily the exemplar tools of XML have no silly restrictions in favor of container elements. Listing 6 is a snippet of XSLT that illustrates the ease of processing XML such as in **Listing 1**.
### Listing 6. XSLT skeleton for processing element groups without containers

```xml
<xsl:transform
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0"
>
  <xsl:template match="library">
    <xsl:apply-templates/>
  </xsl:template>

  <xsl:template match="name">
    <xsl:apply-templates/> <!-- only text node children -->
  </xsl:template>

  <xsl:template match="book">
    <!-- Add code to process @isbn attribute here -->
    <xsl:apply-templates/> <!-- Takes care of title child -->
  </xsl:template>
</xsl:transform>
```

The fact that this listing has no container element does not limit the power or the expressiveness of XSLT, especially if you're using the pull methodology that centers heavily around relatively independent templates. For example, if you wanted to process all children of `library` in the order they appear in the document, and then process only the `book` children, you could easily manage this using template modes, as in Listing 7.

### Listing 7. Sample XSLT skeleton for even more flexible processing of element groups without containers

```xml
<xsl:transform
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0"
>
  <xsl:template match="library">
    <!-- process all children in document order -->
    <xsl:apply-templates/>
    <!-- process only book element children -->
    <xsl:apply-templates select="book" mode="books"/>  
  </xsl:template>

  <xsl:template match="name">
    <xsl:apply-templates/> <!-- only text node children -->
  </xsl:template>

  <xsl:template match="book">
    <!-- When this template is invoked, there may be other types of elements in this (the default) mode -->
    <xsl:apply-templates/>
  </xsl:template>

  <xsl:template match="book" mode="books">
    <!-- Only book elements are processed in this mode, allowing more specialized processing -->
    <xsl:apply-templates/>
  </xsl:template>
</xsl:transform>
```
Now that you are dealing with actual processing of the XML document, it makes perfect sense to introduce program-specific abstractions such as books (which in this example manifests as an XSLT mode). Keeping these processing artifacts out of the source data is one of the most important considerations in XML design. Decades of computer history have shown that code comes and code goes, but data is practically forever.

Wrap-up

In this article I have focused on the choice of containers for groups of homogeneous elements. The basic issues and principles are similar even if the group of elements under consideration for a container is heterogeneous. An example of a heterogeneous container is the head element in XHTML, which wraps elements ranging from title to meta. It's worth noting that in my experience, I have found that it is not uncommon to find real-world analogues for containers of heterogeneous elements.

As I mentioned earlier, some of these XML design issues can be handled in much the same manner as in programming design, whereas some issues require a bit of a shift in perspective. Even if XML design is a somewhat inconvenient detour for you from regular programming or database design, the fresh viewpoint it introduces will help make you a better overall data designer. One principle that may be a new one is the complete separation between modeling the data and modeling the process. I have found that keeping this principle in the back of my mind (based on my experience with XML) has made me a better code designer. Even where my code designs end up merging the data and processing models, the combined models are a bit more fine-tuned from the discipline of having considered data and code separately.
Related topics

- Don't miss the earlier articles in this series on XML design:
  - "When to use elements versus attributes" (developerWorks, March 2004).
  - "Use XML namespaces with care" (developerWorks, April 2004).
  - "Element structures for names and addresses" (developerWorks, August 2004).
- This article was inspired in part by a few private e-mail threads that have been summarized in this Usenet (comp.text.xml) posting.
- Learn about push versus pull techniques for XSLT processing in articles such as the XSL FAQ entry "Push vs Pull" by Clark C. Evans. I do disagree with the conclusion of the latter article that pull processing is apt for less narrative document types. In my experience, push processing is superior almost regardless of the properties of the underlying XML.
- Find more XML resources on the developerWorks XML zone, including Uche Ogbuji's Thinking XML column. Some of the vocabularies Uche discusses in this column -- for example UBL (February 2003) -- define name and address structures, but don't really add any additional points of interest.
- Find out how you can become an IBM Certified Developer.

© Copyright IBM Corporation 2004
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)