Six strategies for extending XML schemas in a single namespace

Create flexible XML schemas that grow to fit changing information needs

Dale Waldt

January 19, 2010

The W3C XML Schema Definition Language allows several powerful techniques for extending schemas to include or redefine elements and attributes. In this article, learn six techniques to extend and redefine your schemas to enable development of robust information architectures that can accommodate enterprise information needs.

W3C XML Schemas have become the core of many business applications because of their powerful data typing and definition capabilities. But a data model isn't always static. Schemas often need ways to allow for extensibility over time to accommodate new information and element types. Several approaches can extend schemas to include new elements as needed: The six strategies described in this article provide techniques to extend single-namespace schemas. Using multiple namespaces to extend the data being processed requires an article of its own.

Note: This article focuses solely on W3C XML Schema version 1.0. The W3C XML Schema Working Group is nearing completion of version 1.1, but it is not yet ratified and might change. The examples here are all based on the current specification.

Frequently used acronyms

- CSS: Cascading stylesheets
- HTML: Hypertext Markup Language
- W3C: World Wide Web Consortium
- XHTML: Extensible HTML
- XML: Extensible Markup Language
- XSD: XML Schema Definition
- XSLT: Extensible Stylesheet Language Transformations

Generic elements

A good example of data that changes over time is code lists. A code list is a list of unique code values that have specific meanings, such as product descriptors, frequently used terms, and lists...
of countries or cities. These values are often stored in a database row that you can add to over
time and use to populate choices in an application window.

The simple code list of colors in Listing 1 illustrates how to extend a schema as new data choices
emerge. It defines a simple code list with the element type color, which contains four possible
elements, the first three of which are given known color names. The last element in the group is
sometimes called a generic element and is designed to allow any value to be inserted in the name
attribute, thereby allowing you to add new colors to the list as needed over time. If you want a new
color choice many months after the application has been completed and deployed, you can specify
a new color—perhaps purple—and use the other element with the attribute name="purple". When
validated, the <other> element is allowed, and you keep working with no changes to the schema
required.

Listing 1. Sample schema defining extensible color code list elements

```xml
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="color">
    <xs:complexType>
      <xs:choice maxOccurs="unbounded">
        <xs:element name="red" type="xs:string"/>
        <xs:element name="blue" type="xs:string"/>
        <xs:element name="green" type="xs:string"/>
        <xs:element name="other">
          <xs:complexType>
            <xs:attribute name="name" type="xs:string"/>
          </xs:complexType>
        </xs:element>
      </xs:choice>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

A sample of valid data associated with this schema that uses the generic element extension is in
Listing 2.

Listing 2. Valid data instance associated with the color code list schema

```xml
<color xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="color.xsd">
  <red/>
  <green/>
  <other name="purple">cc00cc</other> <!-- Extension data -->
  <other name="orange">ee9944</other> <!-- Extension data -->
</color>
```

As you can see, the schema does not define elements with the names purple or orange, but these
names were included in the data instance and parsed as valid because of the extension technique
used. This technique works where a static list exists but new items are added on an ongoing basis.
The creation of the data can be slightly more complicated, but maintaining the schema and related
applications is greatly simplified. Of course, this data could manage all color information in an
attribute instead of an element.

Processing this data requires special handling of the generic <other> element when it occurs in
the data instance. An XPath statement in an XQuery or XSLT stylesheet might test for one of the
predefined elements and also display the known color. Either language has the ability to select one of the known element names to process accordingly, or it can select the `<other>` element and read the attribute value for `name=` and the element content for the color value (expressed here as a CSS style value for the respective colors).

**Modular schema assembly**

You might modularize schemas for a lot of reasons, but this section focuses on using modularity to extend them. In short, creating several schema modules and including them into your base schema is a form of extending the base schema. The example in **Listing 3** uses the schema construct `<xs:include>` to bring in the schema module.

**Listing 3. Bringing in a schema module using xs:include**

```xml
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <!-- Reference to External Module containing USaddress definition -->
  <xs:include schemaLocation="USaddress.xsd"/>

  <!-- Element containing USaddress element from included module -->
  <xs:complexType name="contact">
    <xs:sequence>
      <xs:element name="Name" type="xs:string"/>
      <xs:element ref="USaddress"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

The code in **Listing 3** brings in the included schema module in **Listing 4**.

**Listing 4. The included schema module**

```xml
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="USaddress">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="street1" type="xs:string" minOccurs="0"/>
        <xs:element name="street2" type="xs:string" minOccurs="0"/>
        <xs:element name="city" type="xs:string"/>
        <xs:element name="state" type="xs:string"/>
        <xs:element name="zip" type="xs:string" minOccurs="0"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

The combined resulting schema allows the data instance in **Listing 5** to validate successfully.

**Listing 5. Data instance validated using the included schema**

```xml
<contact>
  <name>Dale Waldt</name>
  <USaddress>
    <city>New York</city>
    <state>NY</state>
  </USaddress>
</contact>
```
Note that the resolved schema will contain all declarations from both the original and the included schemas. Because the example in Listing 5 uses only one namespace, element and attribute names must be unique in the combined resolved form. Also, the occurrence rules must be consistent in the resolved form.

Although adding modules is a form of extending a schema, the potential for building a set of dynamically assembled modules to create flexible schemas for different environments and applications is a powerful concept for optimizing development and maintenance effort. You can create a library of predefined, consistent declarations for developers to selectively use throughout the enterprise. Even so, take special care to prevent naming collisions and other errors from occurring, especially in a single namespace.

Abstract elements and substitution groups

The W3C XML Schema allows for a class of element types that generally appear in the same locations to be treated as a group of equivalent elements in type definitions. For example, you might have several types of named objects (that is, people, places, things) that appear in text as inline elements, including person, city, lodging, restaurant, and museum. You can define a content model, like the one in Listing 6, for the textual <p> element that is defined as mixed content with child elements named <b>, <i>, and <inline>, because paragraphs are likely to have bold, italic, and other inline elements.

Listing 6. Defining elements that are members of a substitution group

```xml
<!-- Paragraph Element Type Definition with Abstract Element -->
<xs:element name="p">
  <xs:complexType>
    <xs:choice maxOccurs="unbounded" mixed="true">
      <xs:element name="b" type="xs:string"/>
      <xs:element name="i" type="xs:string"/>
      <xs:element ref="inline"/>
    </xs:choice>
  </xs:complexType>
</xs:element>

<xs:element name="inline" type="xs:string"/>

<!-- Substitution Element Types -->
<xs:element name="person" type="xs:string" substitutionGroup="inline"/>
<xs:element name="hotel" type="xs:string" substitutionGroup="inline"/>
<xs:element name="city" type="xs:string" substitutionGroup="inline"/>
<xs:element name="url" type="xs:string" substitutionGroup="inline"/>
<xs:element name="email" type="xs:string" substitutionGroup="inline"/>
<xs:element name="phone" type="xs:string" substitutionGroup="inline"/>
```

Note that in Listing 6 the <inline> element is referenced in the attribute substitutionGroup="inline", found in the element declarations that follow the one for <inline>. This means that all elements that are members of the substitution group that has the name of another element can be placed wherever that element is allowed (or an abstract element can serve this purpose). In this example, the substitution group element types <person>, <phone>, and so on are allowed anywhere the <inline> element is—in this case, inline in the text of the
paragraph. The data instance in Listing 7 is valid with this schema and its substitution group extensions. (You can also use this technique with substitutions that reference an abstract element.)

**Listing 7. A valid data instance using substitution group elements**

```xml
<p>This is a <b>paragraph</b> with several inline elements. This sentence mentions <city>Chicago</city>, <person>Mayor Daly</person> and <hotel>The Drake Hotel</hotel> which are named entities that have specific markup applied to them. </p>
```

Also note that all affected elements must be declared globally, not locally in the context of another type definition. The substitution group elements can appear in any order, because they are in a repeatable choice group with `maxOccurs="unbounded"`.

Over time, new uses might require the addition of inline elements. What makes this example extensible is that the schema developer only has to add a new element declaration indicating that it is a member of the substitution group. Of course, the block of element declarations that are substitution group members can be managed in a schema module that is stored separately and included in a main schema. Doing so might simplify the process of adding new element declarations to the substitution group and might even be managed and produced from an application interface, much in the same way code lists are managed and extended.

**Extension to an existing type**

The W3C XML Schema lets you extend existing type definitions to add additional sub-elements, adding additional elements to the data model's structure. You can apply extensions to the types of element or attributes. Given the example type definition in Listing 8, you can define the contents of elements that contain person name information.

**Listing 8. A simple type definition**

```xml
<xs:complexType name="nameType">
    <xs:sequence>
        <xs:element name="fname" type="xs:string"/>
        <xs:element name="lname" type="xs:string"/>
    </xs:sequence>
</xs:complexType>
```

This definition will parse the instance below as valid (assuming that the element `<name>` is defined using the `nameType` type):

```xml
<name><fname>Dale</fname><lname>Waldt</lname></name>
```

You can supply additional sub-elements to the complex type called `nameType` using the example in Listing 9. In this example, you can see that a new `complexType` named `extendedNameType` shows that the extension is to be applied to the base type `nameType` (defined above). Once extended, the base type will inherit the properties of the new extended type in addition to its own definition. In this case, you intend to add the sub-element `<gen>` to the `nameType` element defined above, which already has as sub-elements the `<fname>` and `<lname>` elements.
Listing 9. Adding elements to nameType using xs:extension

```xml
<xs:complexType name="extendedNameType">
    <xs:extension base="nameType">
        <xs:sequence>
            <xs:element name="gen" type="xs:string"/>
        </xs:sequence>
    </xs:extension>
</xs:complexType>

<xs:element name="para" ref="extendedNameType"/>
```

The `<para>` element defined in Listing 9 uses the `extendedNameType`, which has been defined to include all the sub-elements from its base type `nameType` as well as the extension element `<gen>`. This would allow the following instance to validate with no errors:

```xml
<para><fname>Dale</fname><lname>Waldt</lname><gen>Jr.</gen></para>
```

To better understand what is really happening during validation, the example in Listing 10 represents what results when the extension is resolved by the validator (this is a view of the schemas as it is processed, sometimes called the Post Schema Validation Infoset, or PSVI). As you can see, the parsing didn't simply insert the new declaration for the `<gen>` element right after the last element declaration for `lname` in the original `<xs:sequence>` element. Instead, it added a new `<xs:sequence>` element immediately following the original one, resulting in a sequence of sequences, and encapsulated it in an additional `<xs:sequence>` element to preserve the order. In fact, extensions can only be applied to the compositor `<xs:sequence>`, not its counterparts `<xs:choice>` or `<xs:all>`. (Actually, you can extend an `<xs:choice>` compositor, but it will end up inserting an `<xs:sequence>` element.)

Listing 10. Resolved extended type

```xml
<xs:complexType name="ExtendedNameType">
    <xs:sequence>
        <xs:sequence>
            <xs:element name="fname" type="xs:string"/>
        </xs:sequence>
        <xs:sequence>
            <xs:element name="lname" type="xs:string"/>
        </xs:sequence>
    </xs:sequence>
    <xs:sequence>
        <xs:element name="gen" type="xs:string"/>
    </xs:sequence>
</xs:complexType>
```

Again, the code in Listing 10 allows the following instance to validate:

```xml
<name><fname>Dale</fname><lname>Waldt</lname><gen>Jr.</gen></name>
```

You can also add attributes when you extend a type, as in Listing 11, where the `ParaType` complex type definition is extended to add an attribute for `label=`.

Listing 11. Extended ParaType complex type

```xml
<xs:complexType name="ExtendedParaType">
    <xs:sequence>
        <xs:element name="fname" type="xs:string"/>
        <xs:element name="lname" type="xs:string"/>
        <xs:element name="gen" type="xs:string"/>
    </xs:sequence>
    <xs:attribute name="label" type="xs:string"/>
</xs:complexType>
```

Again, the code in Listing 11 allows the following instance to validate:

```xml
<name><fname>Dale</fname><lname>Waldt</lname><gen>Jr.</gen><label>Junior</label></name>
```
**Listing 11. Adding attributes to a named type using xs:extension**

```xml
<xs:complexType name="ParaType">
  <xs:simpleContent>
    <xs:extension base="xs:string">
      <xs:attribute name="label" type="xs:string" use="required"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>

<xs:element name="para" type="ParaType"/>
```

The extension example in **Listing 11** allows the following text instance to validate successfully:

```xml
<para label="abc">Paragraph string text.</para>
```

**Redefining existing types**

Types defined in one schema can be reused and redefined in another schema module. This behavior can be handy if you inherit a schema but want to modify the definition somewhat to work better in your environment. Suppose you're given an industry-standard schema that defines a `simpleType` for the `yearType` simple type, as in **Listing 12**.

**Listing 12. Schema module that defines a simple type named yearType**

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:simpleType name="yearType">
    <xsd:restriction base="xsd:string"/>
  </xsd:simpleType>
  <xsd:element name="year" type="yearType"/>
</xsd:schema>
```

The `yearType` in **Listing 12** is defined strictly as an `<xs:string>`, which might not be rigorous enough for your local environment. Perhaps in the broader world, you might find years that have two or four digits and might even contain an apostrophe, as in '09. But in your internal environment, you might want to force the year always to be four numerical digits to be as unambiguous as possible. Consider a separate schema module that calls the original schema through the `schemaLocation` attribute and redefines it with the code in **Listing 13**.

**Listing 13. Schema module that redefines the simple type named yearType**

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:redefine schemaLocation="prod1.xsd">
    <xsd:simpleType name="yearType">
      <xsd:restriction base="yearType">
        <xsd:length value="4"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:redefine>
  <xsd:element name="fullYear" type="yearType"/>
</xsd:schema>
```
Because the `yearType` has been redefined, it is still referred to using the same name, but when applied to the `<fullYear>` element, it ensures that the `fullYear` will always contain four digits.

Note that redefines require that the old and new definitions have the same original type as their base data type. In the example in Listing 13, it is defined as `xs:string`.

**Wildcards**

The W3C XML Schema allows you to declare some elements using wildcards, or elements that can contain just about any other element or attribute—declared or otherwise. The wildcard `ANY` type is a placeholder whose content might or might not be validated against a schema. Validation is controlled by setting the `processContents` attribute to one of the following levels:

- **Skip**: Do not validate contents.
- **Lax**: Validate only if you can find a corresponding declaration.
- **Strict**: Validate against the current schema.

You define wildcards using the `<xs:any>` or `<xs:anyAttribute>` element for elements or attributes, respectively. The example in Listing 14 shows an element named `<HTMLExample>` that has a wildcard where subordinate elements or attributes would be declared. In other words, the `<HTMLExample>` element can contain any other elements as long as they have well-formed markup. You can add the XHTML Schema and allow the elements to parse against it, and then set the `processContent` level to `lax` to check that it is valid HTML markup. But this example doesn't bother checking the wildcard elements, so leave it set to `skip`.

**Listing 14. Element declaration using xs:any and xs:anyAttribute wildcards**

```xml
<xs:element name="HTMLExample">
  <xs:complexType mixed="true">
    <xs:sequence>
      <xs:any minOccurs="0" maxOccurs="unbounded" processContents="skip"/>
    </xs:sequence>
    <xs:anyAttribute processContents="skip"/>
  </xs:complexType>
</xs:element>
```

Specifically, the example in Listing 14 shows the element declaration to contain a complex type that contains a sequence. The sequence contains the `<any>` element, which is the wildcard placeholder indicating that any element markup can appear at this location. It also contains an `<xs:anyAttribute>` declaration, which allows the addition of any well-formed attributes to the element markup. Because the processing of the contents of this element should be skipped, no schema validation is performed on the contents between the start and end tags. Therefore, the entire element can contain any well-formed markup using any element or attribute names. In this way, you can add content from other document models inside this element, thus extending the types of elements allowed overall in the document—albeit in a specific location. The data instance in Listing 15 is valid given this example.
Listing 15. Valid HTMLExample data instance

```html
<HTMLExample href="http://www.w3.org">
  <tr>
    <th align="left">Table Head</th>
  </tr>
</HTMLExample>
```

In the example in **Listing 15**, the element type `<HTMLExample>` is validated normally against the schema, but the contents of that element, the table row, and the table head HTML elements are skipped.

Take care when you use wildcards if you expect to require validation or allow lax validation if a schema can be found. Resources, such as alternative schemas to validate against, must be made available to the processor. Namespaces must be managed correctly. Also, using wildcards in conjunction with optional or repeatable elements can cause ambiguities and non-deterministic conditions. The simplest use of wildcards with `processContents="skip"` will allow you to avoid most of this complexity.

**Conclusion**

As you can see from the examples in this article, the designers of the W3C XML Schema language had extensibility in mind when they created the standard. Take care to observe the rules for each extension type in order for them to work. These powerful techniques, although only working in a single namespace, can allow tremendous flexibility—especially when you work with schemas used in distributed and diverse environments.

Looking forward, keep an eye on the emerging XML Schema version 1.1 standard being produced in the W3C XML Schema Working Group. It has some interesting changes to the wildcards and other constructs that might affect the examples shown here.
Related topics

- **IBM certification**: Find out how you can become an IBM-Certified Developer in XML and related technologies.
- **XML technical library**: See the developerWorks XML Zone for a wide range of technical articles and tips, tutorials, standards, and IBM Redbooks.
- **IBM product evaluation versions**: Get your hands on application development tools and middleware products.

© Copyright IBM Corporation 2010
**Trademarks**
(www.ibm.com/developerworks/ibm/trademarks/)