Creating Variable Content Container Elements

Table of Contents

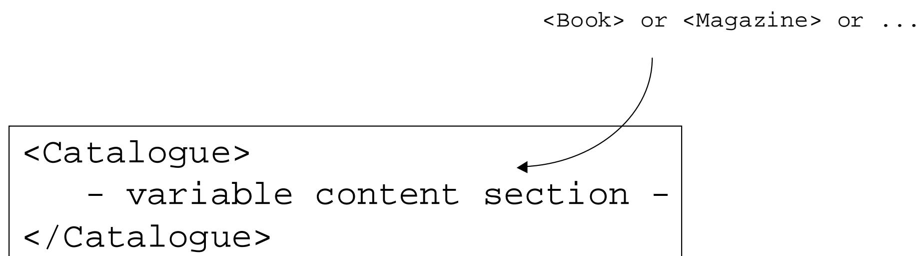
Issue
Introduction
Example
Method 1: Implementing variable content containers using an abstract element and element substitution
Method 2: Implementing variable content containers using a <choice> element
Method 3: Implementing variable content containers using an abstract type and type substitution
Method 4: Implementing variable content containers using a dangling type
Best Practice

Issue

What is the Best Practice for implementing a container element that is to be comprised of variable content?

Introduction

A typical problem when creating an XML Schema is to design a container element (e.g., Catalogue) which is to be comprised of variable content (e.g., Book, or Magazine, or ...)



Catalogue is called a *variable content container*

Some things to consider:

- Do we allow the elements in the variable content container to come from disjoint sources, i.e., do we allow the container element to contain dissimilar, independent, loosely coupled elements?
- How do we design the variable content container so that the kinds of elements which it may contain can grow over time, i.e., how do we design an *extensible* variable content container?

Example

Throughout this discussion we will consider variable content containers (e.g., <Catalogue>) which are comprised of a collection of elements, where each element is variable.

Here's an example of a <Catalogue> container element comprised of two different kinds of elements:

```
<Catalogue>
  <Book> ... </Book>
  <Magazine> ... </Magazine>
  <Book> ... </Book>
</Catalogue>
```

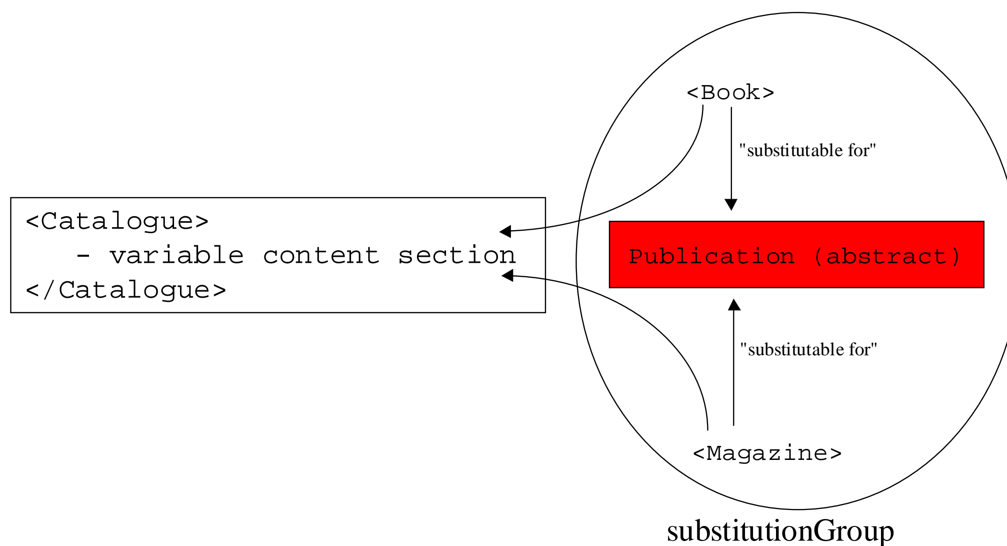
Below are four methods for implementing variable content containers.

Method 1: Implementing variable content containers using an abstract element and element substitution

Description:

There are five XML Schema concepts that must be understood for implementing this method:

- an element can be declared abstract.
- abstract elements cannot be instantiated in instance documents (they are only placeholders).
- in instance documents the abstract element must be substituted by non-abstract (i.e., concrete) elements which have been declared to be in a substitutionGroup with the abstract element.
- elements may be declared to be in a substitutionGroup with the abstract element iff their type is the same as, or derives from the abstract element's type.
- the abstract element and all elements in its substitutionGroup must be declared as global elements.



Implementation:

Declare an abstract element (Publication):

```
<xsd:element name="Publication" abstract="true"
       type="PublicationType"/>
```

Declare a variable content container element (Catalogue) to have as its content the abstract element ("ref" to the abstract element declaration):

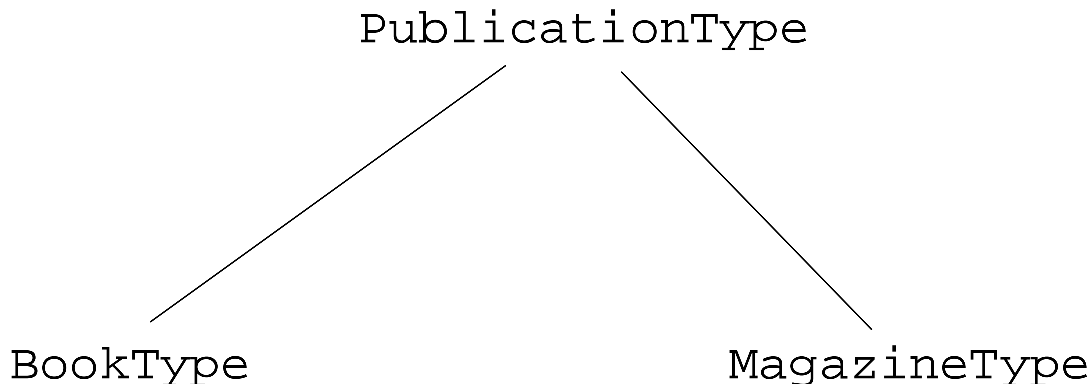
```
<xsd:element name="Catalogue">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="Publication"

             maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

Note that maxOccurs="unbounded", so Catalogue may contain a collection (one or more) of Publication elements.

Declare the concrete elements (Book and Magazine) that are to be the contents of the variable content container and declare them to be in a substitutionGroup with the abstract element:

```
<xsd:element name="Book" substitutionGroup="Publication"
       type="BookType"/>
<xsd:element name="Magazine" substitutionGroup="Publication"
       type="MagazineType"/>
```

In order for Book and Magazine to substitute for Publication, their types (BookType and MagazineType) must derive from Publication's type (PublicationType).



Here are the type definitions:

PublicationType - the base type:

```
<xsd:complexType name="PublicationType">
  <xsd:sequence>
    <xsd:element name="Title" type="xsd:string"/>
    <xsd:element name="Author" type="xsd:string"
           minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element name="Date" type="xsd:gYear"/>
  </xsd:sequence>
</xsd:complexType>
```

BookType - extends PublicationType by adding two new elements, ISBN and Publisher:

```
<xsd:complexType name="BookType">
  <xsd:complexContent>
    <xsd:extension base="PublicationType">
      <xsd:sequence>
        <xsd:element name="ISBN" type="xsd:string"/>
        <xsd:element name="Publisher" type="xsd:string"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

MagazineType - restricts PublicationType by striking out the Author element:

```
<xsd:complexType name="MagazineType">
  <xsd:complexContent>
    <xsd:restriction base="PublicationType">
      <xsd:sequence>
        <xsd:element name="Title" type="xsd:string"/>
        <xsd:element name="Author" type="xsd:string"
               minOccurs="0" maxOccurs="0"/>
        <xsd:element name="Date" type="xsd:gYear"/>
      </xsd:sequence>
    </xsd:restriction>
  </xsd:complexContent>
</xsd:complexType>
```

The following page shows what an instance document looks like with this method:

```xml
<?xml version="1.0"?>
<Catalogue xmlns="http://www.catalogue.org"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation=
                     "http://www.catalogue.org
                      Catalogue.xsd">
    <Book>
        <Title>Illusions The Adventures of a Reluctant Messiah</Title>
        <Author>Richard Bach</Author>
        <Date>1977</Date>
        <ISBN>0-440-34319-4</ISBN>
        <Publisher>Dell Publishing Co.</Publisher>
    </Book>
    <Magazine>
        <Title>Natural Health</Title>
        <Date>1999</Date>
    </Magazine>
    <Book>
        <Title>The First and Last Freedom</Title>
        <Author>J. Krishnamurti</Author>
        <Date>1954</Date>
        <ISBN>0-06-064831-7</ISBN>
        <Publisher>Harper &amp; Row</Publisher>
    </Book>
</Catalogue>
```

Advantages:

Extensible: This method allows you to extend the set of elements that may be used in the variable content container element, *even if the schema for the variable content container element is outside your control*. For example, suppose that you do not have privilege to modify the above Catalogue schema. Currently, the Catalogue element can only contain Book and Magazine elements. But suppose that your application has a hard requirement for CD elements as well:

```xml
<Catalogue>
  <Book> ... </Book>
  <CD> ... </CD>
  <Magazine> ... </Magazine>
  <Book> ... </Book>
</Catalogue>
```

How can you extend the set of elements that Catalogue may be comprised of, *without modifying its schema*?

Answer: You can create your own separate schema which contains a declaration of CD (with a type, CDType, that extends the PublicationType in the Catalogue schema), and declares CD to be in the Publication substitutionGroup:

```
<xsd:include schemaLocation="Catalogue.xsd"/>
<xsd:complexType name="CDType">
  <xsd:complexContent>
    <xsd:extension base="PublicationType">
      <xsd:sequence>
        <xsd:element name="RecordingCompany"
               type="xsd:string"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="CD" substitutionGroup="Publication"
       type="CDType"/>
```

The CD element meets the requirements for being in the variable content container:

- its type (CDType) derives from the PublicationType, and
- it is a member of the Publication element's substitutionGroup.

Book, Magazine, and CD may now be used within the Catalogue element, e.g.,

```
<?xml version="1.0"?>
<Catalogue xmlns="http://www.catalogue.org"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation=
            "http://www.catalogue.org
             CD.xsd">
    <Book>
        <Title>Illusions The Adventures of a Reluctant Messiah</Title>
        <Author>Richard Bach</Author>
        <Date>1977</Date>
        <ISBN>0-440-34319-4</ISBN>
        <Publisher>Dell Publishing Co.</Publisher>
    </Book>
    <CD>
        <Title>Timeless Serenity</Title>
        <Author>Dyveke Spino</Author>
        <Date>1984</Date>
        <RecordingCompany>Dyveke Spino Productions</RecordingCompany>
    </CD>
    ...
</Catalogue>
```

Thus, we see that this method allows us to extend the set of elements that may be used in the Catalogue element, without modifying its schema. Nice!

Semantic Cohesion: the elements in the variable content container all descend from the same type hierarchy (PublicationType). This type hierarchy binds them together, giving a structural (and, by implication, semantic) coherence to all the elements that may be in the variable content container.

Disadvantages:

No Independent Elements: The type of the elements that are to be used in the variable content container must all descend from the abstract element's type (PublicationType). Further, the elements must be in a substitutionGroup with the abstract element. Thus, the variable content container cannot contain elements whose type does not derive from the abstract element's type, or is not in the substitutionGroup with the abstract element - as would typically be the case with independently developed elements. For example, suppose another schema author creates a "Newspaper" element, with a type that does not descend from PublicationType. <Catalogue> would not be able to contain the <Newspaper> element.

Limited Structural Variability: Over time a schema will evolve, and the kinds of elements which may occur in the variable content container will typically grow. There is no way to know apriori in what direction it will grow. The new elements may be *conceptually* related but *structurally* vastly different from the original set of elements. The abstract element's type (e.g., PublicationType) may have been a good base type for the original set of elements which were all structurally related, but may not be a good base type for the new elements which have vastly different structures.

So you are faced with a tradeoff:

- create a simple base type to support lots of different structures (but then you can make less assumptions about the structure of the members), or
- create a rich base type to support strong data type checking (but then you reduce the ability to add elements with radically different types)

Nonscalable Processing: Processing a collection of differently named elements requires a lot of special-case code. For example, consider a stylesheet to process each element in <Catalogue>:

```
<xsl:if test="Book">
 -- process Book --
</xsl:if>
<xsl:if test="Magazine">
 -- process Magazine --
</xsl:if>
```

This stylesheet snippet suffers from lack of scalability, i.e., it breaks as soon as a new element is added.

This argument needs some qualification. If the contents of <Catalogue> are just elements that substitute for the abstract Publication element, then each element can be uniformly processed, as follows:

```
<xsl:for-each select="Catalogue/*">
 -- process the element --
</xsl:for-each>
```

This stylesheet snippet processes each element within Catalogue, regardless of the element name. Obviously, this is scalable, and does not break when a new element is added.

Processing becomes non-scalable when Catalogue contains multiple abstract elements:

```
<xsd:element name="Catalogue">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="Publication"
             maxOccurs="unbounded"/>
      <xsd:element ref="Retailer"
             maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

Suppose that both Publication and Retailer are abstract elements, and there can be any number of each kind of element within Catalogue. Here's a sample instance:

```
<Catalogue>
  <Book> ... </Book>
  <Magazine> ... </Magazine>
  <Book> ... </Book>
  <MarketBasket> ... </MarketBasket>
  <Macys> ... </Macys>
</Catalogue>
```

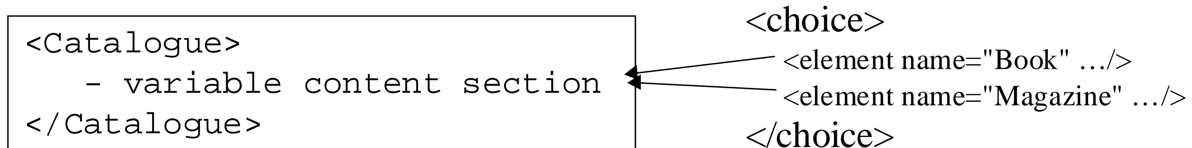
If you wish to process just the Publication elements (e.g., Book, Magazine) then you will need to write special-case code, as shown above. This is not scalable. Every time a new element is added into the collection of elements that may substitute for the Publication element then your code will have to be updated. This is costly.

No Control over Namespace Exposure: This method requires that the elements which may be used in the variable content container be in a substitutionGroup with the abstract element (e.g., Book and Magazine must be in a substitutionGroup with Publication). A requirement of using substitionGroup is that all elements must be declared globally. The namespace of global elements can never be hidden in instance documents. As a consequence, there is no way to hide (localize) the namespaces of the elements used in the variable content container. This fails the Best Practice rule which states that you should design your schema to be able to hide or expose namespaces at your discretion (using elementFormDefault as an *exposure switch*). (See the chapter titled Hide (Localize) Versus Expose Namespaces)

Method 2: Implementing variable content containers using a <choice> element

Description:

This method is quite straightforward - simply list within a <choice> element all the elements which can appear in the variable content container, and embed the <choice> element in the container element.



Implementation:

Declare within a <choice> element all the elements (e.g., Book, Magazine) that may be used in the variable content container. Embed the <choice> element within the container element (Catalogue):

```
<element name="Catalogue">
  <complexType>
    <choice maxOccurs="unbounded">
      <element name="Book" type="BookType"/>
      <element name="Magazine" type="MagazineType"/>
    </choice>
  </complexType>
</element>
```

Advantages:

Independent Elements: The elements in the variable content container do not need a common type ancestry. They don't have to be related in any way. Thus, the variable content container can contain dissimilar, independent, loosely coupled elements.

Disadvantages:

Nonextensible: Suppose that the Catalogue schema is outside your control. Currently the variable content container only supports Book and Magazine. Suppose that you have a hard requirement for your instance documents to use CD as well as Book and Magazine within Catalogue, e.g.,

```
<Catalogue>
  <Book> ... </Book>
  <CD> ... </CD>
  <Magazine> ... </Magazine>
  <Book> ... </Book>
</Catalogue>
```

This method requires that the <choice> element in the Catalogue schema be modified to include the CD element. However, we stipulated that the Catalogue schema is outside your control, so it cannot be modified. This method has serious extensibility restrictions!

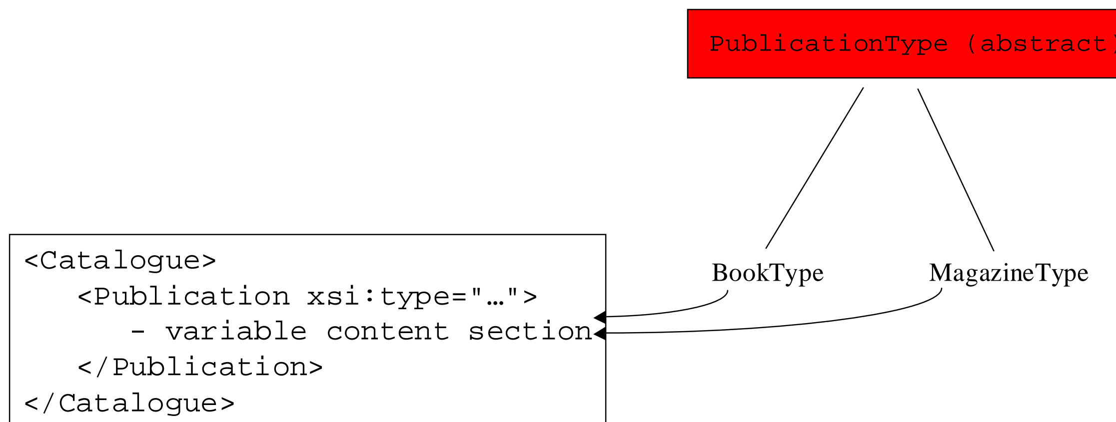
No Semantic Coherence: The <choice> element allows you to group together dissimilar elements. While that has been touted as an advantage, it is really a double-edged sword. The elements in the variable content container have no type hierarchy to bind them together, to provide structural (and, by implication, semantic) coherence among the elements. Thus, when processing an instance document you can make no assumptions about the structure of the elements.

Method 3: Implementing variable content containers using an abstract type and type substitution

Description:

There are three XML Schema concepts that must be understood for implementing this method:

- a complexType can be declared abstract.
- an element declared to be of an abstract type cannot have its type instantiated in instance documents (that is, the element can be instantiated, but its abstract content may not).
- in instance documents an element with an abstract type must have its content substituted by content from a non-abstract (concrete) type which derives from the abstract type. This is called type substitution.



Implementation:

Define an abstract base type (PublicationType):

```
<xsd:complexType name="PublicationType" abstract="true">
  <xsd:sequence>
    <xsd:element name="Title" type="xsd:string"/>
    <xsd:element name="Author" type="xsd:string"
           minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element name="Date" type="xsd:gYear"/>
  </xsd:sequence>
</xsd:complexType>
```

Declare the container element (Catalogue) to contain an element (Publication), which is of the abstract type:

```
<xsd:element name="Catalogue">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="Publication"

             type="PublicationType"
             maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

In instance documents, the content of <Publication> can only be of a concrete type which derives from PublicationType, such as BookType or MagazineType (we saw these type definitions in Method 1 above).

With this method instance documents will look different than we saw with the above two methods. Namely, <Catalogue> will not contain variable content. Instead, it will always contain the same element (Publication). However, that element will contain variable content:

```
<?xml version="1.0"?>
<Catalogue xmlns="http://www.catalogue.org"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation=
                  "http://www.catalogue.org
                   Catalogue.xsd">
      <Publication xsi:type="BookType">
            <Title>Illusions The Adventures of a Reluctant Messiah</Title>
            <Author>Richard Bach</Author>
            <Date>1977</Date>
            <ISBN>0-440-34319-4</ISBN>
            <Publisher>Dell Publishing Co.</Publisher>
      </Publication>
      <Publication xsi:type="MagazineType">
            <Title>Natural Health</Title>
            <Date>1999</Date>
      </Publication>
      ...
</Catalogue>
```

Advantages:

Extensible: Same extensibility benefits as method 1. Namely, this method allows you to easily extend the set of elements that may be used in the variable content container simply by creating new types which derive from the abstract type, e.g.,

```
<include schemaLocation="Catalogue.xsd"/>
<complexType name="CDType">
      <complexContent>
         <extension base="PublicationType" >
            <sequence>
               <element name="RecordingCompany" type="string"/>
            </sequence>
         </extension>
      </complexContent>
</complexType>
```

Now the content of <Publication> may be BookType, or MagazineType, or CDType

CD.xsd

We have extended the Catalogue schema without modifying it! Here's an example instance document with the new CD element:

```
<?xml version="1.0"?>
<Catalogue xmlns="http://www.catalogue.org"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation=
                  "http://www.catalogue.org
                   CD.xsd">
      <Publication xsi:type="BookType">
            <Title>Illusions The Adventures of a Reluctant Messiah</Title>
            <Author>Richard Bach</Author>
            <Date>1977</Date>
            <ISBN>0-440-34319-4</ISBN>
            <Publisher>Dell Publishing Co.</Publisher>
      </Publication>
      <Publication xsi:type="CDType">
            <Title>Timeless Serenity</Title>
            <Author>Dyveke Spino</Author>
            <Date>1984</Date>
            <RecordingCompany>Dyveke Spino Productions</RecordingCompany>
      </Publication>
      ...
</Catalogue>
```

Minimal Dependencies: This method has less dependencies (coupling) than method 1. To extend the collection of elements that may appear in a variable content container using method 1 you need access to both the abstract element (Publication) and its type (PublicationType). With method 3 you only need access to the abstract type. If we assume that in a typical scenario only the types will be put in publicly accessible schemas, then method 3 is the only viable method.

Scalable Processing: Processing a series of <Publication> elements is scalable. For example, a stylesheet could process each publication element as follows:

```
<xsl:for-each select="Publication">
 -- do something --
</xsl:for-each>
```

As new types are created (e.g., CDType) no change is needed to the code.

Semantic Cohesion: the elements in the variable content container all descend from the same type hierarchy. This type hierarchy binds them together, giving a structural (and, by implication, semantic) coherence among the elements.

Control over Namespace Exposure: the variable part of the variable content container are the element declarations that are embedded within type definitions. Consequently, we can control exposure of the namespaces of the variable content container elements. This is consistent with the Best Practice design recommendation we issued for hide (localize) versus expose namespaces. (See the chapter titled Hide (Localize) Versus Expose Namespaces)

Disadvantages:

No Independent Elements: Same weakness as with method 1. All types must descend from an abstract type. This requirement prohibits the use of types which do not descend from the abstract type, as would typically be the situation when the type is in another, independently developed schema.

Limited Structural Variability: Same weakness as with method 1. Namely, to facilitate strong type checking you want to have a rich base type, but this is in direct conflict with the desire for components with vastly different structures, which calls for a weak base type.

Method 4: Implementing variable content containers using a dangling type

Motivation:

Thus far our variable content container has contained complex content (i.e., child elements). Suppose that we want to create a variable content container to hold simple content? None of the previous methods can be used. We need a method that allows us to create simpleType variable content containers.

There is one key XML Schema concept that must be understood for implementing this method:

- with an <import> element the schemaLocation attribute is optional

Description:

Let’s take an example. Suppose that we desire an element, sensor, which contains the name of a weather station sensor. For example:

```
<sensor>Barometric Pressure</sensor>
```

There are several things to note:

1. This element holds a simpleType
2. Each weather station may have sensors that are unique to it. Consequently, we must design our schema so that the sensor element can be customized by each weather station

Here's an elegant design for making the contents of <sensor> customizable by each weather station:

Implementation:

Let's go through the design, step by step. In your schema, declare the sensor element:

```
<xsd:element name="sensor" type="s:sensor_type"/>
```

Note that the sensor element is declared to have a type "sensor_type", which is in a different namespace - the sensor namespace:

```
xmlns:s="http://www.sensor.org"
```

Now here's the key - when you <import> this namespace, don't provide a value for schemaLocation! (In an import element schemaLocation is optional.) For example:

```
<xsd:import namespace="http://www.sensor.org"/>
```

The *instance document* must then identify a schema that implements sensor_type. Thus, at *run time* (i.e., validation time) we are matching up the reference to sensor_type with an implementation of sensor_type. For example, an instance document may have this:

```
xsi:schemaLocation=
   "http://www.weather-station.org weather-station.xsd
    http://www.sensor.org boston-sensors.xsd"
```

In this instance document schemaLocation is identifying a schema, boston-sensors.xsd, which is to provide the implementation of sensor_type.

Let's take a look at the schemas and instance documents for the weather station sensor example we have been considering. Here's the main schema, which contains the dangling type:

weather-station.xsd

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                   targetNamespace="http://www.weather-station.org"
                   xmlns="http://www.weather-station.org"
                   xmlns:s="http://www.sensor.org"
                  elementFormDefault="qualified">
    <xsd:import namespace="http://www.sensor.org"/>
    <xsd:element name="weather-station">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="sensor" type="s:sensor_type"
                                     maxOccurs="unbounded"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
</xsd:schema>
```

An import with no schemaLocation!

Note that the <import> element does not have a schemaLocation attribute to identify a particular schema which implements sensor_type. (Stated differently, this schema does not *hardcode* in the identity of the schema which is to provide the implementation of sensor_type.) The schema validator will resolve the reference to sensor_type based upon the collection of schemas that is provided to it in the instance document.

The Boston weather station creates a schema which implements sensor_type:

boston-sensors.xsd

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                   targetNamespace="http://www.sensor.org"
                   xmlns="http://www.sensor.org"
                   elementFormDefault="qualified">
    <xsd:simpleType name="sensor_type">
        <xsd:restriction base="xsd:string">
            <xsd:enumeration value="barometer"/>
            <xsd:enumeration value="thermometer"/>
            <xsd:enumeration value="anenometer"/>
        </xsd:restriction>
    </xsd:simpleType>
</xsd:schema>
```

This schema provides an implementation for the dangling type, sensor_type.

Now an instance document can conform to weather-station.xsd and use boston-sensors.xsd as the implementation of sensor_type:

boston-weather-station.xml

```
<?xml version="1.0"?>
<weather-station xmlns="http://www.weather-station.org"
                 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                 xsi:schemaLocation=
                          "http://www.weather-station.org weather-station.xsd
                           http://www.sensor.org boston-sensors.xsd">
    <sensor>thermometer</sensor>
    <sensor>barometer</sensor>
    <sensor>anenometer</sensor>
</weather-station>
```

In the instance document we provide a schema which implements the dangling type.

Suppose that the London weather station has all the sensors that Boston has, plus some additional ones that are unique to the London weather patterns. Thus, London will create its own implementation of sensor_type:

london-sensors.xsd

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://www.sensor.org"
            xmlns="http://www.sensor.org"
            elementFormDefault="qualified">
  <xsd:simpleType name="sensor_type">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="barometer"/>
      <xsd:enumeration value="thermometer"/>
      <xsd:enumeration value="anenometer"/>
      <xsd:enumeration value="hygrometer"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

This schema provides a different implementation for the dangling type, sensor_type.

Note that this schema has an additional sensor_type that Boston does not have - hygrometer.

Just as with the Boston weather station instance document, the London weather station instance document will conform to a collection of schemas: weather-station.xsd and london-sensors.xsd:

london-weather-station.xml

```
<?xml version="1.0"?>
<weather-station xmlns="http://www.weather-station.org"
                 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                 xsi:schemaLocation=
                       "http://www.weather-station.org weather-station.xsd
                        http://www.sensor.org london-sensors.xsd">
   <sensor>thermometer</sensor>
   <sensor>barometer</sensor>
   <sensor>hygrometer</sensor>
   <sensor>anenometer</sensor>
</weather-station>
```

The London weather station is able to customize the content of <sensor> by using london-sensors.xsd, which defines sensor_type appropriately for the London weather station. Wow!

Summary:

This method represents an extraordinarily powerful design pattern. The key to this design pattern is:

1. When you declare the variable content container element give it a type that is in another namespace, e.g., s:sensor_type
2. When you <import> that namespace don't provide a value for schemaLocation, e.g., <xsd:import namespace="http://www.sensors.org"/>
3. Create any number of implementations of the dangling type, e.g.,
   - boston-sensors.xsd
   - london-sensors.xsd
4. In instance documents identify the schema that you want used to implement the dangling type, e.g.,

```
xsi:schemaLocation=
  "http://www.weather-station.org weather-station.xsd
   http://www.sensor.org london-sensors.xsd"
```

Both simpleType and complexType:

In our examples we have implemented the dangling type as a simpleType. The implementation of a dangling type does not have to be a simpleType. A schema could define it as a complexType.

Advantages:

Dynamic: A schema which contains a dangling type is very dynamic. It does not statically hard-code the identity of a schema to implement the type. Rather, it empowers the instance document author to identify a schema that implements the dangling type. Thus, at *instance-document-creation* the type implementation is provided (rather than at *schema-document-creation*)
Applicable to both Simple and Complex Types: A dangling type can be implemented as either a simpleType or a complexType. The other methods are only applicable to creating variable content containers with a complex type.

Disadvantages:

Different Namespace: The implementation of the dangling type must be in another namespace. It cannot be in the same namespace as the variable content container element. If you have a hard requirement that the contents of your variable content container have the same namespace as the container element then this method cannot be employed.

Best Practice

Which method you should use to create your variable content containers ultimately depends on your requirements. Here are some things to consider.

Use Method 1 (abstract element plus element substitution) when:

- It's okay for all the elements to descend from a common type.
- You need to provide the ability to extend the collection of elements in the variable content container without modifying its schema.
- You can live with the container elements all being namespace-exposed in instance documents.

Use Method 2 (<choice> element) when:

- You need to contain a collection of dissimilar, independent elements
- It is adequate to have an external authority (i.e., a human) verify the collection of legal elements. Verification is accomplished by the external authority selecting which elements shall be allowed in the <choice> element
- Growth of the collection of elements is tightly determined by the external authority that controls the schema.

Use Method 3 (abstract type with type substitution) when:

- All the elements in the variable content container are of the same type, or derived from the same type
- It's okay to give all the elements in a variable content container a uniform name.
- The collection of elements may grow, independent of the container schema.
- You need to support namespace-hiding.
- You need to support scalable processing.

Use Method 4 (dangling type) when:

- You need a simpleType variable content container
- You need to extend a simpleType
- You need very dynamic, customizable content

Best Practice: Method 4 is by far the most flexible approach. Unfortunately, as of today (August 16, 2001) none of the schema validators have implemented dangling types. The workaround is to use the anyType. For example: <xsd:element name="sensor" type="anyType"/>. We lose a bit of type checking with this, but it is the best that we can do today. Encourage the schema validator developers to support this capability!