Global versus Local

Table of Contents

Issue

When should an element or type be declared global versus when should it be declared local?

Introduction

[Recall that a component (element, complexType, or simpleType) is “global” if it is an immediate child of <schema>, whereas it is “local” if it is not an immediate child of <schema>, i.e., it is nested within another component.]

What advice would you give to someone who was to ask you, “In general, when should an element (or type) be declared global versus when should it be declared local”? The purpose of this chapter is to provide answers to that question.

Below is a snippet of an XML instance document. We will explore the different design strategies using this example.

```
<Book>
  <Title>Illusions</Title>
  <Author>Richard Bach</Author>
</Book>
```

Russian Doll Design

This design approach has the schema structure mirror the instance document structure, e.g., declare a Book element and within it declare a Title element followed by an Author element:

```
<xsd:element name=“Book”>
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name=“Title” type=“xsd:string”/>
      <xsd:element name=“Author” type=“xsd:string”/>
    </xsd:sequence>
  </xsd:complexType>
</element>
```

The instance document has all its components bundled together. Likewise, the schema is designed to bundle together all its element declarations.

This design represents one end of the design spectrum.

Salami Slice Design

The Salami Slice design represents the other end of the design spectrum. With this design we disassemble the instance document into its individual components. In the schema we define each component (as an element declaration), and then assemble them together:

```
<xsd:element name="Title" type="xsd:string"/>
<xsd:element name="Author" type="xsd:string"/>
<xsd:element name="Book">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="Title"/>
      <xsd:element ref="Author"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

Note how the schema declared each component individually (Title, and Author) and then assembled them together (by ref'ing them) in the creation of the Book component.

These two designs represent opposite ends of the design spectrum.

To understand these designs it may help to think in terms of boxes, where a box represents an element or type:

- The Russian Doll design corresponds to having a single box, and it has nested within it boxes, which in turn have boxes nested within them, and so on. (boxes within boxes, just like a Russian Doll!)
- The Salami Slice design corresponds to having many separate boxes which are assembled together (separate boxes combined together, just like Salami slices brought together in a sandwich!)

Let's examine the characteristics of each of the two designs. (In so doing it will yield insights into another design.)

Russian Doll Design Characteristics

[1] *Opaque content.* The content of Book is opaque to other schemas, and to other parts of the same schema. The impact of this is that none of the types or elements within Book are reusable.

[2] *Localized scope.* The region of the schema where the Title and Author element declarations are applicable is localized to within the Book element. The impact of this is that if the schema has set elementFormDefault="unqualified" then the namespaces of Title and Author are hidden (localized) within the schema.

[3] *Compact.* Everything is bundled together into a tidy, single unit.

[4] *Decoupled.* With this design approach each component is self-contained (i.e., they don't interact with other components). Consequently, changes to the components will have limited impact. For example, if the components within Book changes it will have a limited impact since they are not coupled to components outside of Book.

[5] *Cohesive.* With this design approach all the related data is grouped together into self-contained components, i.e., the components are cohesive.

Salami Slice Design Characteristics

[1] *Transparent content.* The components which make up Book are visible to other schemas, and to other parts of the same schema. The impact of this is that the types and elements within Book are reusable.

[2] *Global scope.* All components have global scope. The impact of this is that, irrespective of the value of elementFormDefault, the namespaces of Title and Author will be exposed in instance documents.

[3] *Verbose.* Everything is laid out and clearly visible.

[4] *Coupled.* In our example we saw that the Book element depends on the Title and Author elements. If those elements were to change it would impact the Book element. Thus, this design produces a set of interconnected (coupled) components.

[5] *Cohesive.* With this design approach all the related data is also grouped together into self-contained components. Thus, the components are cohesive.

The two design approaches differ in a couple of important ways:

- The Russian Doll design facilitates hiding (localizing) namespace complexities. The Salami Slice design does not.
- The Salami Slice design facilitates component reuse. The Russian Doll design does not.

Is there a design which facilitates hiding (localizing) namespace complexities, **and** facilitates component reuse? Yes there is!

Consider the Book example again. An alternative design is to create a global type definition which nests the Title and Author element declarations within it:

```
<xsd:complexType name="Publication">
  <xsd:sequence>
    <xsd:element name="Title" type="xsd:string"/>
    <xsd:element name="Author" type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:element name="Book" type="Publication"/>
```

This design has both benefits:

- it is capable of hiding (localizing) the namespace complexity of Title and Author, and
- it has a reusable Publication type component.

Venetian Blind Design

With this design approach we disassemble the problem into individual components, as the Salami Slice design does, but instead of creating element declarations, we create type definitions. Here's what our example looks like with this design approach:

```
<xsd:simpleType name="Title">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="Mr."/>
    <xsd:enumeration value="Mrs."/>
    <xsd:enumeration value="Dr."/>
  </xsd:restriction>
</xsd:simpleType>
<xsd:simpleType name="Name">
  <xsd:restriction base="xsd:string">
    <xsd:minLength value="1"/>
  </xsd:restriction>
</xsd:simpleType>
<xsd:complexType name="Publication">
  <xsd:sequence>
    <xsd:element name="Title" type="Title"/>
    <xsd:element name="Author" type="Name"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:element name="Book" type="Publication"/>
```

This design has:

- maximized reuse (there are four reusable components - the Title type, the Name type, the Publication type, and the Book element)
- maximized the potential to hide (localize) namespaces [note how this has been phrased: "maximized the *potential* ..."Whether, in fact, the namespaces of Title and Author are hidden or exposed, is determined by the elementFormDefault "switch"].

The Venetian Blind design espouses these guidelines ...

- Design your schema to maximize the *potential* for hiding (localizing) namespace complexities.
- Use elementFormDefault to act as a *switch* for controlling namespace exposure - if you want element namespaces exposed in instance documents, simply turn the elementFormDefault switch to "on" (i.e, set elementFormDefault= "qualified"); if you don't want element namespaces exposed in instance documents, simply turn the elementFormDefault switch to "off" (i.e., set elementFormDefault="unqualified").
- Design your schema to maximize reuse.
- Use type definitions as the main form of component reuse.
- Nest element declarations within type definitions.

Let's compare the Venetian Blind design with the Salami Slice design. Recall our example:

Salami Slice Design:

```
<xsd:element name=“Title" type=“xsd:string"/>
<xsd:element name=“Author" type=“xsd:string"/>
<xsd:element name=“Book">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref=“Title"/>
      <xsd:element ref=“Author" />
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

The Salami Slice design also results in creating reusable (element) components, but it has absolutely no potential for namespace hiding.

“*However*", you argue, “*Suppose that I want namespaces exposed in instance documents.* [We have seen cases where this is desired.] *So the Salami Slice design is a good approach for me. Right?*"

Let’s think about this for a moment. What if at a later date you change your mind and wish to hide namespaces (what if your users hate seeing all those namespace qualifiers in instance documents)? You will need to redesign your schema (possibly scraping it and starting over).

Better to adopt the Venetian Blind Design, which allows you to control whether namespaces are hidden or exposed by simply setting the value of elementFormDefault. No redesign of your schema is needed as you switch from exposing to hiding, or vice versa.

[That said ... your particular project may need to sacrifice the ability to turn on/off namespace exposure because you require instance documents to be able to use element substitution. In such circumstances the Salami Slice design approach is the only viable alternative.]

Here are the characteristics of the Venetian Blind Design.

Venetian Blind Design Characteristics:

[1] *Maximum reuse.* The primary component of reuse are type definitions.

[2] *Maximum namespace hiding.* Element declarations are nested within types, thus maximizing the potential for namespace hiding.

[3] *Easy exposure switching.* Whether namespaces are hidden (localized) in the schema or exposed in instance documents is controlled by the elementFormDefault switch.

[4] *Coupled.* This design generates a set of components which are interconnected (i.e., dependent).

[5] *Cohesive.* As with the other designs, the components group together related data. Thus, the components are cohesive.

Best Practice

[1] The Venetian Blind design is the one to choose where your schemas require the flexibility to turn namespace exposure on or off with a simple switch, and where component reuse is important.

[2] Where your task requires that you make available to instance document authors the option to use element substitution, then use the Salami Slice design.

[3] Where mimimizing size and coupling of components is of utmost concern then use the Russian Doll design.