Zero, One, or Many Namespaces?

Table of Contents

Issue
Introduction
Example

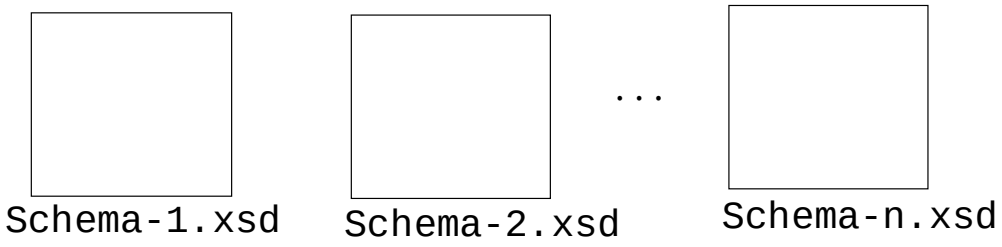
Heterogeneous Namespace Design
Homogeneous Namespace Design
Chameleon Namespace Design

Impact of Design Approach on Instance Documents
<redefine> - only Applicable to Homogeneous and Chameleon Namespace Designs
Default Namespace and the Chameleon Namespace Design
Avoiding Name Collisions with Chameleon Components
Creating Tools for Chameleon Components
Best Practice

Issue:

In a project where multiple schemas are created, should we give each schema a different targetNamespace, or should we give all the schemas the same targetNamespace, or should some of the schemas have no targetNamespace?

Managing Multiple Schemas - Same or Different targetNamespaces?



. . . or no targetNamespace?

Introduction

In a typical project many schemas will be created. The schema designer is then confronted with this issue: “shall I define one targetNamespace for all the schemas, or shall I create a different targetNamespace for each schema, or shall I have some schemas with no targetNamespace?” What are the tradeoffs? What guidance would you give someone starting on a project that will create multiple schemas?

Here are the three design approaches for dealing with this issue:

[1] **Heterogeneous Namespace Design**:
give each schema a different targetNamespace

[2] **Homogeneous Namespace Design**:
give all schemas the same targetNamespace

[3] **Chameleon Namespace Design**:
give the “main” schema a targetNamespace and give no targetNamespace to the “supporting” schemas (the no-namespace supporting schemas will take-on the targetNamespace of the main schema, just like a Chameleon)

To describe and judge the merits of the three design approaches it will be useful to take an example and see each approach “in action”.

Example: XML Data Model of a Company

Imagine a project which involves creating a model of a company using XML Schemas. One very simple model is to divide the schema functionality along these lines:

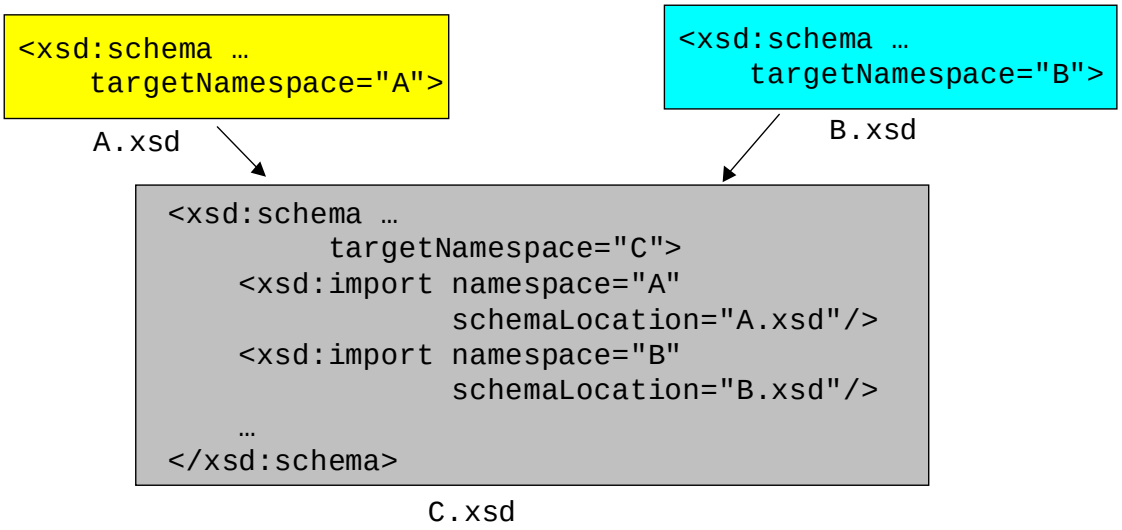
```
Company schema
  Person schema
  Product schema
```

“A company is comprised of people and products.”

Here are the company, person, and product schemas using the three design approaches.

[1] Heterogeneous Namespace Design

This design approach says to give each schema a different targetNamespace, e.g.,



Below are the three schemas designed using this design approach. Observe that each schema has a different targetNamespace.

Product.xsd

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://www.product.org"
            xmlns="http://www.product.org"
            elementFormDefault="qualified">
  <xsd:complexType name="ProductType">
    <xsd:sequence>
      <xsd:element name="Type" type="xsd:string" minOccurs="1" maxOccurs="1"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

Person.xsd

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                targetNamespace="http://www.person.org"
                xmlns="http://www.person.org"
                elementFormDefault="qualified">
  <xsd:complexType name="PersonType">
    <xsd:sequence>
      <xsd:element name="Name" type="xsd:string"/>
      <xsd:element name="SSN" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

Company.xsd

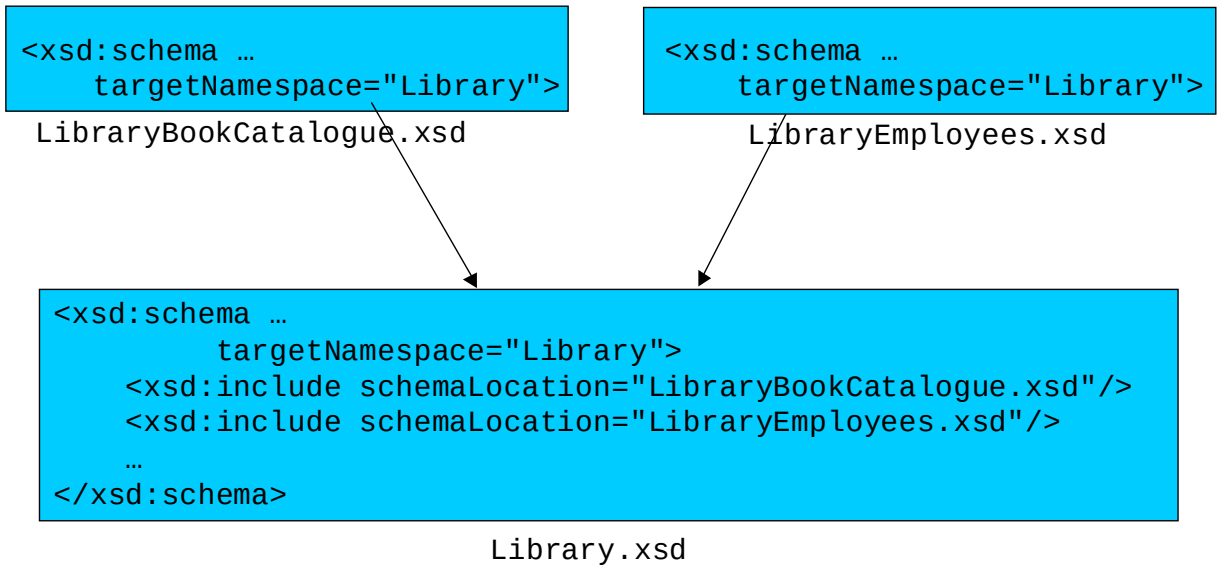
```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                targetNamespace="http://www.company.org"
                xmlns="http://www.company.org"
                elementFormDefault="qualified"
                xmlns:per="http://www.person.org"
                xmlns:pro="http://www.product.org">
  <xsd:import namespace="http://www.person.org"
          schemaLocation="Person.xsd"/>
  <xsd:import namespace="http://www.product.org"
          schemaLocation="Product.xsd"/>
  <xsd:element name="Company">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Person" type="per:PersonType" maxOccurs="unbounded"/>
        <xsd:element name="Product" type="pro:ProductType" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Note the three namespaces that were created by the schemas:

```
http://www.product.org
http://www.person.org
http://www.company.org
```

[2] Homogeneous Namespace Design

This design approach says to create a single, umbrella targetNamespace for all the schemas, e.g.,



Below are the three schemas designed using this approach. Observe that all schemas have the same targetNamespace.

Product.xsd

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                targetNamespace="http://www.company.org"
                xmlns="http://www.product.org"
                elementFormDefault="qualified">
  <xsd:complexType name="ProductType">
    <xsd:sequence>
      <xsd:element name="Type" type="xsd:string" minOccurs="1" maxOccurs="1"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

Person.xsd

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                targetNamespace="http://www.company.org"
                xmlns="http://www.person.org"
                elementFormDefault="qualified">
   <xsd:complexType name="PersonType">
      <xsd:sequence>
        <xsd:element name="Name" type="xsd:string"/>
        <xsd:element name="SSN" type="xsd:string"/>
      </xsd:sequence>
   </xsd:complexType>
</xsd:schema>
```

Company.xsd

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                targetNamespace="http://www.company.org"
                xmlns="http://www.company.org"
                elementFormDefault="qualified">
   <xsd:include schemaLocation="Person.xsd"/>
   <xsd:include schemaLocation="Product.xsd"/>
   <xsd:element name="Company">
      <xsd:complexType>
         <xsd:sequence>
            <xsd:element name="Person" type="PersonType" maxOccurs="unbounded"/>
            <xsd:element name="Product" type="ProductType" maxOccurs="unbounded"/>
         </xsd:sequence>
      </xsd:complexType>
   </xsd:element>
</xsd:schema>
```

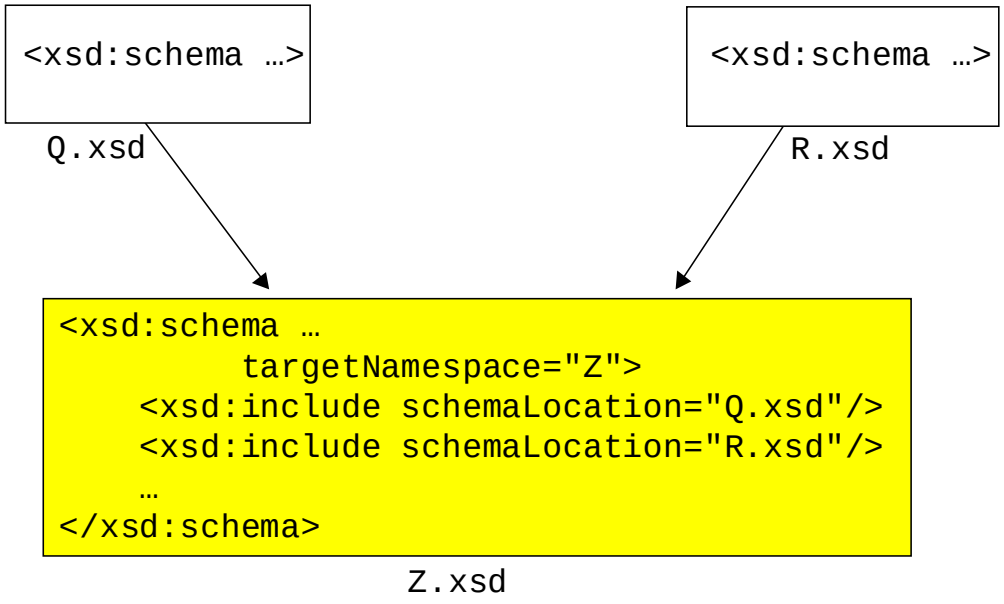
Note that all three schemas have the same targetNamespace:

```
http://www.company.org
```

Also note the mechanism used for accessing components in other schemas which have the same targetNamespace: <include>. When accessing components in a schema with a different namespace the <import> element is used, as we saw above in the Heterogeneous Design.

[3] Chameleon Namespace Design

This design approach says to give the "main" schema a targetNamespace, and the "supporting" schemas have no targetNamespace, e.g.,



In our example, the company schema is the main schema. The person and product schemas are supporting schemas. Below are the three schemas using this design approach:

Product.xsd (no targetNamespace)

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            elementFormDefault="qualified">
  <xsd:complexType name="ProductType">
    <xsd:sequence>
      <xsd:element name="Type" type="xsd:string" minOccurs="1" maxOccurs="1"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

Person.xsd (no targetNamespace)

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                elementFormDefault="qualified">
  <xsd:complexType name="PersonType">
    <xsd:sequence>
      <xsd:element name="Name" type="xsd:string" minOccurs="1" maxOccurs="1"/>
      <xsd:element name="SSN" type="xsd:string" minOccurs="1" maxOccurs="1"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

Company.xsd (main schema, uses the no-namespace-schemas)

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                targetNamespace="http://www.company.org"
                xmlns="http://www.company.org"
                elementFormDefault="qualified">
  <xsd:include schemaLocation="Person.xsd"/>
  <xsd:include schemaLocation="Product.xsd"/>
  <xsd:element name="Company">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Person" type="PersonType" maxOccurs="unbounded"/>
        <xsd:element name="Product" type="ProductType" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

There are two things to note about this design approach:

First, as shown above, a schema is able to access components in schemas that have no targetNamespace, using <include>. In our example, the company schema uses the components in Product.xsd and Person.xsd (and they have no targetNamespace).

Second, note the chameleon-like characteristics of schemas with no targetNamespace:

– The components in the schemas with no targetNamespace get namespace-coerced. That is, the components “take-on” the targetNamespace of the schema that is doing the <include>.

For example, ProductType in Products.xsd gets implicitly coerced into the company targetNamespace.

“Chameleon effect” ... This is a term coined by Henry Thompson to describe the ability of components in a schema with no targetNamespace to take-on the namespace of other schemas. This is powerful!

Impact of Design Approach on Instance Documents

Above we have shown how the schemas would be designed using the three design approaches. Let’s turn now to the instance document. Does an instance document differ depending on the design approach? All of the above schemas have been designed to expose the namespaces in instance documents (as directed by: elementFormDefault=“qualified”). If they had instead all used elementFormDefault=“unqualified” then instance documents would all have this form:

```
<?xml version="1.0"?>
<c:Company xmlns:c="http://www.company.org"
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xsi:schemaLocation=
                        "http://www.company.org
                         Company.xsd">
    <Person>
        <Name>John Doe</Name>
        <SSN>123-45-6789</SSN>
    </Person>
    <Product>
        <Type>Widget</Type>
    </Product>
</c:Company>
```

It is when the schemas expose their namespaces in instance documents that differences appear. In the above schemas, they all specified elementFormDefault=“qualified”, thus exposing their namespaces in instance documents. Let’s see what the instance documents look like for each design approach:

[1] Company.xml (conforming to the multiple targetNamespaces version)

```
<?xml version="1.0"?>
<Company xmlns="http://www.company.org"
                xmlns:per="http://www.person.org"
                xmlns:prod="http://www.product.org"
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xsi:schemaLocation=
                        "http://www.company.org
                         Company.xsd">
    <Person>
        <per:Name>John Doe</per:Name>
        <per:SSN>123-45-6789</per:SSN>
    </Person>
    <Product>
        <prod:Type>Widget</prod:Type>
    </Product>
</Company>
```

Note that:

– there needs to be a namespace declaration for each namespace

– the elements must all be uniquely qualified (explicitly or with a default namespace)

[2] Company.xml (conforming to the single, umbrella targetNamespace version)

```
<?xml version="1.0"?>
<Company xmlns="http://www.company.org"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation=
                  "http://www.company.org
                   Company.xsd">
    <Person>
        <Name>John Doe</Name>
        <SSN>123-45-6789</SSN>
    </Person>
    <Product>
        <Type>Widget</Type>
    </Product>
</Company>
```

Since all the schemas are in the same namespace the instance document is able to take advantage of that by using a default namespace.

[3] Company.xml (conforming to the main targetNamespace with supporting no-targetNamespace version)

```
<?xml version="1.0"?>
<Company xmlns="http://www.company.org"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation=
                  "http://www.company.org
                   Company.xsd">
    <Person>
        <Name>John Doe</Name>
        <SSN>123-45-6789</SSN>
    </Person>
    <Product>
        <Type>Widget</Type>
    </Product>
</Company>
```

Both of the schemas that have no targetNamespace take on the the company targetNamespace (ala the Chameleon effect). Thus, all components are in the same targetNamespace and the instance document takes advantage of this by declaring a default namespace.

<redefine> - only Applicable to Homogeneous and Chameleon Namespace Designs

The <redefine> element is used to enable access to components in another schema, while simultaneously giving the capability to modify zero or more of the components. Thus, the <redefine> element has a dual functionality:

– it does an implicit <include>. Thus it enables access to all the components in the referenced schema

– it enables you to redefine zero or more of the components in the referenced schema, i.e., extend or restrict components

Example. Consider again the Company.xsd schema above. Suppose that it wishes to use ProductType in Product.xsd. However, it would like to extend ProductType to include a product ID. Here's how to do it using redefine:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                targetNamespace="http://www.company.org"
                xmlns="http://www.company.org"
                elementFormDefault="qualified">
  <xsd:include schemaLocation="Person.xsd"/>
  <xsd:redefine schemaLocation="Product.xsd">
    <xsd:complexType name="ProductType">
      <xsd:complexContent>
        <xsd:extension base="ProductType">
          <xsd:sequence>
            <xsd:element name="ID" type="xsd:ID"/>
          </xsd:sequence>
        </xsd:extension>
      </xsd:complexContent>
    </xsd:complexType>
  </xsd:redefine>
  <xsd:element name="Company">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Person" type="PersonType" maxOccurs="unbounded"/>
        <xsd:element name="Product" type="ProductType" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Now the <Product> element in instance documents will contain both <Type> and <ID>, e.g.,

```
<?xml version="1.0"?>
<Company xmlns="http://www.company.org"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xsi:schemaLocation=
                        "http://www.company.org
                         Company.xsd">
      <Person>
            <Name>John Doe</Name>
            <SSN>123-45-6789</SSN>
      </Person>
      <Product>
            <Type>Widget</Type>
            <ID>1001-1-00</ID>
      </Product>
</Company>
```

The <redefine> element is very powerful. However, it can only be used with schemas with the same targetNamespace or with no targetNamespace. Thus, it only applies to the Homogenous Namespace Design and the Chameleon Namespace Design.

Name collisions

When a schema uses Chameleon components those components become part of the including schema's targetNamespace, just as though the schema author had typed the element declarations and type definitions inline. If the schema <include>s multiple no-namespace schemas then there will be a chance of name collisions. In fact, the schema may end up not being able to use some of the no-namespace schemas because their use results in name collisions with other Chameleon components. To demonstrate the name collision problem, consider this example:

Suppose that there are two schemas with no targetNamespace:

```
1.xsd
  A
  B
2.xsd
 A
 C
```

Schema 1 creates no-namespace elements A and B. Schema 2 creates no-namespace elements A, and C. Now if schema 3 <include>s these two no-namespace schemas there will be a name collision:

```
3.xsd
targetNamespace="http://www.example.org"
<include schemaLocation="1.xsd"/>
<include schemaLocation="2.xsd"/>
```

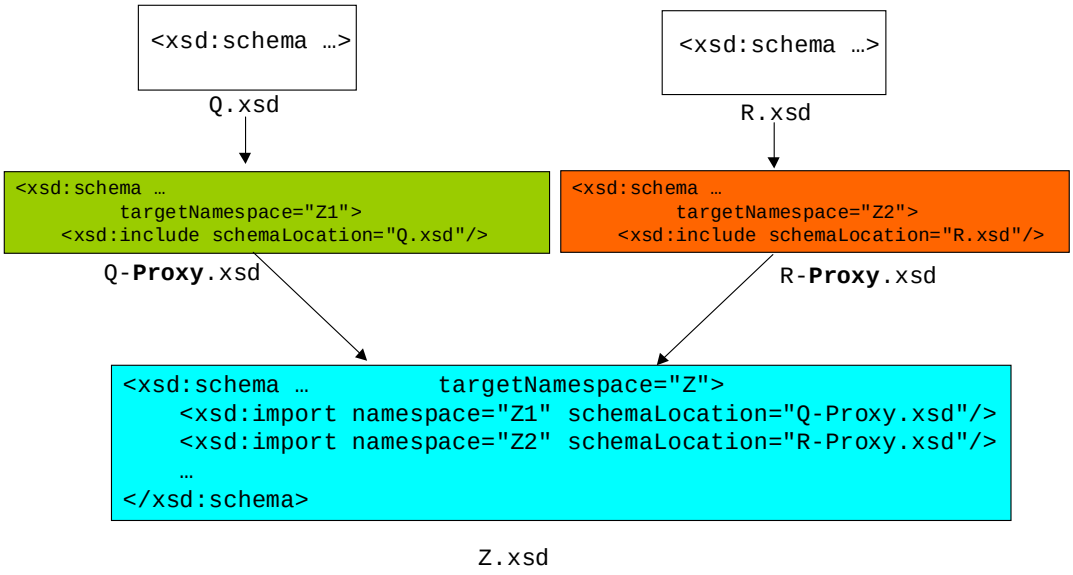
This schema has a name collision - A is defined twice. [Note: it's not an error to have two elements in the same symbol space, provided they have the same type. However, if they have a different type then it is an error, i.e., name collision.]

Namespaces are the standard way of avoiding such collisions. Above, if instead the components in 1.xsd and 2.xsd resided in different namespaces then 3.xsd could have <import>ed them and there would be no name collision. [Recall that two elements/types can have the same name if the elements/types are in different namespaces.]

How do we address the name collision problem that the Chameleon design presents? That's next.

Resolving Namespace Collisions using Proxy Schemas

There is a very simple solution to the namespace collision problem: for each no-namespace schema create a companion namespaced-schema (a "proxy schema") that <include>s the no-namespace schema. Then, the main schema <import>s the proxy schemas.



With this approach we avoid name collisions. This design approach has the added advantage that it also enables the proxy schemas to customize the Chameleon components using <redefine>.

Thus, this approach is a two-step process:

Create the Chameleon schemas
Create a proxy schema for each Chameleon schema

The "main" schema <import>s the proxy schemas.

The advantage of this two-step approach is that it enables *applications* to decide on a domain (namespace) for the components that it is reusing. Furthermore, applications are able to refine/

customize the Chameleon components. This approach requires an extra step (i.e., creating proxy schemas) but in return it provides a lot of flexibility.

Contrast the above two-step process with the below one-step process where the components are assigned to a namespace from the very beginning:

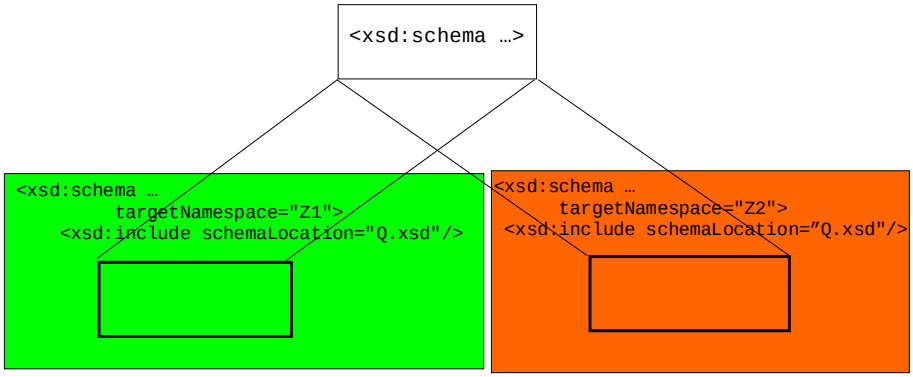
```
1-fixed.xsd
targetNamespace="http://www.1-fixed.org"
  A
  B
2-fixed.xsd
targetNamespace="http://www.2-fixed.org"
 A
 C
main.xsd
targetNamespace="http://www.main.org"
<xsd:import namespace="http://www.1-fixed.org"
      schemaLocation="1-fixed.xsd"/>
<xsd:import namespace="http://www.2-fixed.org"
      schemaLocation="2-fixed.xsd"/>
```

This achieves the same result as the above two-step version. In this example, the components are not Chameleon. Instead, A, B, and C were hardcoded with a namespace from the very beginning of their life. The downside of this approach is that if main.xsd wants to <redefine> any of the elements it cannot. Also, applications are forced to use a domain (namespace) defined by someone else. These components are in a rigid, static, fixed namespace.

Creating Tools for Chameleon Components

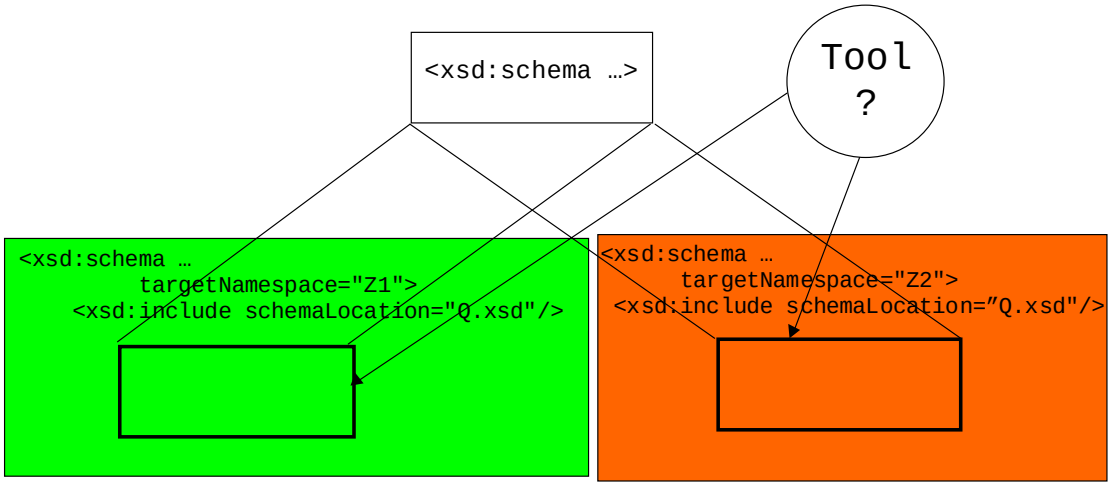
Tools for Chameleon Components

We have seen repeatedly how Chameleon components are able to blend in with the schemas that use them. That is, they adopt the namespace of the schema that <include>s them.



Chameleon components take-on the namespace of the <include>ing schema

How do you write tools for components that can assume so many different faces (namespaces)?



How does a tool identify components that can assume many faces? Certainly not by namespaces.

Consider this no-namespace schema:

```
1.xsd
  A
  B
```

Suppose that we wish to create a tool, T, which must process the two Chameleon components A and B, regardless of what namespace they reside in. The tool must be able to handle the following situation: imagine a schema, main.xsd, which <include>s 1.xsd. In addition, suppose that main.xsd has its own element called A (in a different symbol space, so there's no name collision). For example:

```
main.xsd
targetNamespace="http://www.example.org"
<include schemaLocation="1.xsd"/>
<element name="stuff">
  <complexType>
   <sequence>
     <element name="A" type="xxx"/>
     ...
   </sequence>
  </complexType>
</element>
```

How would the tool T be able to distinguish between the Chameleon component A and the local A in an instance document?

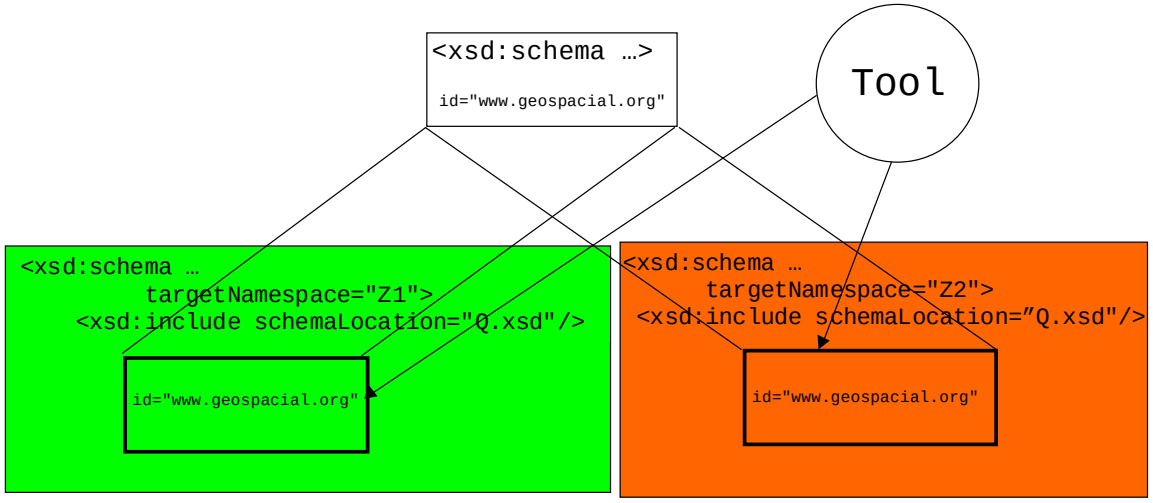
Chameleon Component Identification

One simple solution is that when you create Chameleon components assign them a global unique id (a GUID). The XML Schema spec allows you to add an attribute, id, to all element, attribute, complexType, and simpleType components.

```
<xsd:element name="Lat_Lon"
             id="http://www.geospacial.org"
             …
</xsd:element>
```

Each component (element, complexType, simpleType, attribute) in a schema can have an associated id attribute. This can be used to uniquely identify each Chameleon component, regardless of its namespace.

Note that the id attribute is purely local to the schema. There is no representation in the instance documents. This id attribute could be used by a tool to "locate" a Chameleon component, regardless of what "face" (namespace) it currently wears. That is, the tool can open up an instance document using DOM, and the DOM API will provide the tool access to the id value for all components in the instance document.



A tool can locate the Chameleon component by using the id attribute.

Best Practice

Above we explored the "design space" for this issue. We looked at the three design approaches in action, both schemas and instance documents. So which design is better? Under what circumstances?

When you are reusing schemas that *someone else* created you should <import> those schemas, i.e., use the Heterogeneous Namespace design. It is a bad idea to copy those components into your namespace, for two reasons: (1) soon your local copies would get out of sync with the other schemas, and (2) you lose interoperability with any existing applications that process the other schema's components. The interesting case (the case we have been considering throughout this discussion) is how to deal with namespaces in a collection of schemas that *you* created. Here's our guidelines for this case:

Use the Chameleon Design:

– with schemas which contain components that have no inherent semantics by themselves,

– with schemas which contain components that have semantics only in the context of an <include>ing schema,

– when you don't want to hardcode a namespace to a schema, rather you want <include>ing schemas to be able to provide their own application-specific namespace to the schema

Example. A repository of components - such as a schema which defines an array type, or vector, linked list, etc - should be declared with no targetNamespace (i.e., Chameleon).

As a rule of thumb, if your schema just contains type definitions (no element declarations) then that schema is probably a good candidate for being a Chameleon schema.

Use the Homogeneous Namespace Design

- when all of your schemas are conceptually related
- when there is no need to visually identify in instance documents the origin/lineage of each element/attribute. In this design all components come from the same namespace, so you loose the ability to identify in instance documents that "element A comes from schema X". Oftentimes that's okay - you don't want to categorize elements/attributes differently. This design approach is well suited for those situations.

Use the Heterogeneous Namespace Design

- when there are multiple elements with the same name. (Avoid name collision)
- when there is a need to visually identify in instance documents the origin/lineage of each element/attribute. In this design the components come from different namespaces, so you have the ability to identify in instance documents that "element A comes from schema X".

Lastly, as we have seen, in a schema each component can be uniquely identified with an id attribute (this is NOT the same as providing an id attribute on an element in instance documents. We are talking here about a schema-internal way of identifying each schema component.) Consider identifying each schema component using the id attribute. This will enable a finer degree of traceability than is possible using namespaces. The combination of namespaces plus the schema id attribute is a powerful tandem for visually and programmatically identifying components.