XML Schema best practices

By David Stephenson

December 2004

Introduction

The XML Schema language is a complex maze of constructs that overlap each other. Creating an XML schema is hard to get right; creating several interlocking schemas which can be extended and versioned is even harder. This document provides you with a map that allows you to navigate the maze with confidence.

This document consists of the following:

- An overview of the common requirements that are important for all schemas
- A list of best practices for designing, writing, and coding to XML Schema

There is also an appendix that discusses the rationale behind these best practices.

Meta requirements for XML schemas

The design and specification of XML schemas should be as rigorous an activity as designing and developing code or designing database schemas. As such, when creating an XML schema you should be working within a development process and working to a set of design guidelines and coding standards (when writing the XML schema file). XML schemas should be reviewed for accuracy and compliance with guidelines and standards. Each of the requirements in the list below is a general requirement for all XML schemas that are intended to be used in a deployed system.

- **Understandable:** XML schemas should be clear, consistent and unambiguous. They should contain human readable documentation and, where appropriate, links to requirements or design documents.
- **Semantically complete:** An XML schema should define, for one or more target XML documents, each and every element and attribute that is understood by your solution when processing target documents. For example, if your application uses the value of an attribute or element then a definition for that item should be included in the XML schema.
- This may sound obvious or even necessary for correct validation of the target document. It isn't in many cases. When creating an extensible XML schema as described in this document it is surprisingly easy to fail to describe every single field that is used by your application.
- **Constraining:** An XML schema is a contract published so that both the creator of an XML document and the recipient of an XML document can verify that the instance document obeys the contract. When designing an XML schema you should constrain the values for all the elements and attributes that the application uses and relies on to the set of values that the application can handle. A valid document should imply valid data within the limits of what can be specified by the XML Schema language. It is important to understand that we are ensuring that the full application required state is defined in the XML schema, but not restricting the document to hold the application state (see extensibility below).
- **Non-redundant:** XML schemas should import and include other XML schema files rather than duplicating types and elements locally.
- **Reusable:** XML schemas should be specified in such a way that types and elements can be leveraged by other XML schemas. Every type defined in an XML schema that is the content type of an attribute or an element should be defined globally (i.e., at the top level in the Schema). Types that are defined globally can be reused in other XML schemas.
- **Extensible:** Schemas should be designed to be extensible—that is, new elements and attributes can be inserted throughout the document. Extension points can be made explicit if the schema is being designed to be enhanced. Mechanisms that can be used to enable extensibility include: attribute and element wild cards, substitution groups, and type substitution. Note that this requirement seems

incompatible with the requirement for constraining above, and they do often produce some tension in the design process, but both are important features of a complete schema.

- **Non-modifying:** XML schemas should not specify default values for attributes and element content. This is because default values may cause XML Schema validation to produce heisenbugs. Validating against an XML schema must not change any values in the Infoset.

XML Schema may also be used at run time to check that XML instance documents conform to their XML schema. As validating an XML document against its XML schema is an expensive operation it is usual to omit XML Schema validation in a deployed system.

Schema design best practices

This section contains condensed best practices for XML schemas. It does not contain the rationale behind the best practice (the appendix contains the background rationale) nor does it contain an explanation of XML Schema features or how they work. There are plenty of good books and articles on XML Schema available in order to learn XML Schema.

Namespaces URI, URL vs. URN

When creating an XML schema, should you specify a `targetNamespace`?

Yes always.

When creating an XML schema you need to specify a `targetNamespace` URI, what type of URI should you choose: a URN or a URL?

The URI used as the target namespace in an XML schema must be a valid URI; it should be unique and "owned" by the entity that creates the XML schema.

The URI used should be an `http:` URL that points to a web page owned by the entity creating the XML schema. For example: `http://www.w3.org/2003/XInclude`.

The URI should resolve to an HTML web page that describes the namespace, probably referencing the XML schema[s] documents that define that XML namespace. See the best practices for versioning (below) for more detail on the form of the URL and what should be included in the web page.

Should I set the default namespace to the XML schema namespace or the `targetNamespace`?

This is a human readability issue as it does not affect the semantics of the XML schema at all. The cleanest and simplest (but not the most compact) approach is to not use the default namespace and map both the XML schema namespace and the `targetNamespace` using prefixes. For example:

```
<xsd:schema
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        targetNamespace="www.example.com/2003-11/examples1"
        xmlns:tns="www.example.com/2003-11/examples1" >
```

If your schema has a suggested prefix (like `wsdl:` for the WSDL namespace) then use that prefix otherwise use "tns" which stands for Target Name Space.

Attributes vs. elements

When specifying an XML schema you often have the choice of placing a value in an attribute or an element, which should you use?

For example:

```
<foo bar='42'/>
```

Or:

```
<foo><bar>42</bar></foo>
```

There is no real consensus on when to use attributes or elements. Here are best practices to help you choose when writing your XML schema.

a. Use attributes for metadata about the parent element (foo is the parent element in the example above).
b. Use attributes for data that is semantically tied to the enclosing element.
c. Use elements for data that have a meaning separate from the enclosing element.
d. Use attributes when the value will be frequently present in order to improve the human readable form of an XML instance document or reduce its size.
e. If you don't know which to use, then use the element form (which is more extensible).

Qualified element names

Should local element names be qualified or unqualified?
Element names should always be qualified.

When you write an XML schema always place the following attribute `elementFormDefault="qualified"` in the `<xsd:schema>` element.

Qualified attribute names

Should all attributes in a document be namespace qualified?
No, `attributeFormDefault` should not be set; it should be left to default to "unqualified".

When should I declare and use a global attribute?
If an attribute is declared at the top level in an XML schema file then it is a global definition and that attribute will always be namespace qualified wherever it is seen.

You should use global attributes in the following cases only:

- The attribute is used (or is being designed to be used) across several disjoint XML schemas and always has the same meaning.
- You need to introduce a new attribute into another XML schema in order to extend that schema.

A good example of a global attribute is `xml:lang` which is used in many XML documents but always has the same meaning.

I have an attribute that is used several times in different elements in my XML schema. What should I do?
Create an attribute group to wrap the attribute in, and reference that attribute group from the complex types that contain the attribute.

Global elements vs. local elements

When should you define a global vs. a local element?
Always define elements globally.

Elements within model groups (`choice`, `sequence`) should always use the `ref=` form and never the `type=` form.

Named types vs. anonymous types

When should you define named complex types vs. inline anonymous complex types?

Complex types should always be named and never anonymous. Global element declarations should always use the `type=` form.

When should you define named simple types vs. inline anonymous simple types?

Elements and attributes should always use the `type=` form and never contain anonymous types.

Simple type definitions may themselves contain inline anonymous simple type definitions. Simple types should not be inline anonymous types inside an attribute or element.

Extensible content models, wildcards, `anyAttribute`

Should I allow extension to my content model?

This depends on many factors including the overall reusability and extensibility requirement for the XML schema you are creating. Some XML schemas are defined to be unextendable; in others extensibility is a strong requirement. Reasonable approaches include:

a. Non extendable—no attribute or element wildcards and blocking of element and type substitution. This may seem a rational choice but in that case why use XML in the first place?
b. Targeted element extension and general attribute extension—this is used in many schemas including the XML Schema Schema. (See the appendix for further explanation.)
c. General extensibility—the XML Schema allows general attribute extension as above but also includes wildcards for external elements where they make sense in the document structure.

Should I allow arbitrary extension to my content model via `anyAttribute`?

Yes, it is nearly always reasonable to include an `anyAttribute` with `namespace="##other"` and `processContent="lax"`, which would allow attributes defined in other XML schemas. It may be reasonable to set `namespace="##any"` in which case, the attributes may be in any namespace or none, which would include local attributes.

Should I allow arbitrary extension to my content model via `<xs:any>`?

Depending on your overall extension strategy for your XML schema, you will need to make a judgment on whether you should include an element wildcard in each content model. The longer lived and more widely used your XML schema is going to be, the more important it is to include extensibility in your XML schema. It is very hard to guess how people are going to want to extend your schema in the future, it is better to err on the side of too many wildcards than too few.

Placing an `<xs:any>` at the end of your complex type content is a good way of adding extensibility to your XML schema. As with `anyAttribute`, `namespace="##other"` and `processContent="lax"` are good settings. Wildcard elements should be able to repeat, so set `maxOccurs="unbounded"`.

Counter examples when you should NOT add an `<xs:any>` include:

a. Types which are abstract can omit the `<xs:any>` as it will be added by the derived type
b. Container types, lists, and sets as they are inherently extendable

Should I use `lax`, `strict`, or `skip` for the `processContent` attribute?

These values only affect the extent of validation performed on foreign elements and attributes. We can use the `processContent` attribute value to indicate some semantics about the criticality of the element or attribute.

For attributes always use **`lax`**.

For elements:

- If you use **`lax`** then this will imply that the element is outside the remit of a target application and it will very likely be ignored.
- If you use **`strict`** then this will imply that the element is a required part of the semantics of the XML document and must be understood by a target implementation. If it is not understood, the document will not be processed. The use of `strict` makes sense when the element is not optional, i.e., `minOccurs` is greater than zero.

Constraints: max, min, length, pattern, etc.

Should I constrain my simple type to the exact data expected using facets, or should I leave room for extensibility?

A simple rule is: if a value absolutely must be in some value set for normal processing of a document then constrain the values to that set, otherwise you may add some flexibility (and some extra documentation!).

What you often want to be able to do is specify the expected value while leaving open the possibility of other values. To do this you can use a union simple type as follows:

```
<xs:simpleType name="zipType">
  <xs:restriction base="xs:integer">
    <xs:maxInclusive value="0"/>
    <xs:maxExclusive value="99999"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="genericZipType">
  <xs:union memberTypes="tns:zipType xs:string"/>
</xs:simpleType>
```

Determining if the type should be constrained, open, or somewhere in-between depends on the specific situation, usage, and domain in which the Schema is being defined.

Should I set `minOccurs` and `maxOccurs` so that the data is as expected or should I allow a wider range of values to leave room for extensibility?

Generally, set the `minOccurs` and `maxOccurs` to what could reasonably make sense and document it.

Setting the `minOccurs` and `maxOccurs` depends on what it would mean to have more or less elements than expected. If you can describe how implementations should handle more or less elements than expected, then you can extend the range of legal values (and document it!); if you can't think how it would ever make sense, then restrict the values.

Graph data structures

If my data structure is a graph how should I represent that in my XML document and my XML schema?

XML documents can only represent a tree, if you need to include further structure in your XML document then you will need to link elements in an XML document using values (attributes or subelements) in these elements. You can use `key` and `keyref` constructs in XML Schema to describe these relationships.

Should I use `ID` and `IDREF`?

No, these DTD features have problems that prohibit general usage.

Should I use `unique`, `key`, and `keyref`?

Yes, these XML Schema constructs are excellent for describing relationships between values in your schema. You should understand how to use these in your XML schema (see XML Schema Part 0: Primer Second Edition <http://www.w3.org/TR/xmlschema-0/>).

Identifiers in XML schemas

How should you name elements and attributes?

Pick a descriptive name without it being excessively long. Use camel case, e.g., "applePearOrange". You should be consistent in your use of upper or lower camel case, i.e., "ApplePearOrange" vs. "applePearOrange". If you are extending an XML schema, follow the form used in the original XML schema.

If you are creating an XML schema from scratch, use lower camel case, i.e., "applePearOrange" as that is the most widespread form of XML document in use today (e.g., XHTML, XML Schema, XSLT, etc.).

How should you name simple and complex types?

XML Schema has separate symbol space for elements and types so they may have the same name. The best practice is to append the word "Type" to all simple and complex type names to aid in human readability and comprehension. So if you wanted to create a simple type for zipCodes element then it could be called "zipCodeType".

Default/fixed attributes value and default element values

Should I use default values for attributes and elements?

No, a better alternative is to indicate the value that should be used if the element or attribute is absent in the human readable documentation. XML Schema validation should not change any values in the Infoset or heisenbugs may occur.

Should I use fixed attributes?

No, for the reason given above. Fixed attributes act like default attributes but if present must have a fixed value. It is reasonable to define an attribute as both required and fixed but such an attribute has little or no worth.

Versioning

How should I version my XML schemas?

When versioning any type of asset there are generally two levels of version tracking:

- Major—completely different structure and semantics, most likely not backward compatible, new versions of applications are written to use new asset version
- Minor—backward compatible changes which introduce new features without removing or changing the semantics of existing structures

The best practice for encoding these two version numbers in an evolving XML schema is to encode the major version inside the target XML namespace of the XML schema and to encode the minor version in the schema version attribute.

How should I indicate the version in the namespace URI?

The best practice for a namespace URIs is as follows:

```
http://$domain/$groupSpecifier/$namespaceTitle/$year-$month
```

The `groupSpecifier` can include several levels, e.g., `/HPS/EMEA/UK/sales/`.

An example namespace could be:

```
http://www.example.org/DFG/tech/schemas/foobar/2003-11
```

As stated previously, the URL should resolve to a web page which describes the schema and includes the group/person who owns the XML schema. The page might include other information, for example, references to products or code that work with that version of the XML schema.

The namespace should have a main page which can be accessed using the namespace URI minus the year-month part (e.g., `http://www.example.org/DFG/tech/schemas/foobar/`). The main page should reference the latest version of the schema and the support status of the previous versions.

Can I reuse a namespace for a new version of the schema?

The namespace name may be reused in any minor version change update of the XML schema which is made for the purpose of clarification, bug fixes, or limited backward-compatible extensions. These changes will be constrained so that they do not:

a. Change the meaning or validity of existing documents written using the XML schema, or
b. Affect the operation of existing software written to process such documents, or
c. Increase or decrease the size of the set of documents that will validate against the XML schema

If you need to break any of the rules above, you should use a new namespace (major version change).

The first version of an XML schema for a namespace should have the version attribute on the `<xsd:schema` element set to `"1"`; subsequent versions should increment that value. When a namespace is reused, the new XML schema version always replaces the previous version of the XML schema.

The rules above do leave some scope for adding new attributes and elements (if the wildcards have been correctly placed) without changing the namespace. Great care should be taken when adding to an existing schema namespace.

Mixed content

Should I enable mixed content in my `complexType` Element?

Generally no. One good exception is where you are defining an HTML-like XML file structure where textual content is designed to be intermixed with markup. It is a bad idea to enable mixed content purely for extensibility reasons.

Simple type code lists

How should I define a simple type which defines a possibly extensible set of enumeration values?

If an enumeration is going to be extensible then its values should be drawn from either a hierarchical set or a probabilistically unique set. The simplest solution is to use a URI or a QName. Using QNames as enumeration values is a best practice as it provides easy extensibility, a compact representation, validation and implicit linkage to documentation. Below is an example attribute that holds one of several values:

```
<xs:attributeGroup name="widgetVarient">
  <xs:attribute name="widgetVarient" type="xs:QName">
    <xs:annotation>
      <xs:documentation>
      The varient of the widget in question.
      The following QNames should be understood:
        <dl>
          <dt>www.example.com/2003-11/examples1:spangle</dt>
          <dd>an ugly widget</dd>
          <dt>www.example.com/2003-11/examples1:rangle</dt>
          <dd>annoying unrequested widget</dd>
          <dt>www.example.com/2003-11/examples1:wangle</dt>
          <dd>argumentative widget</dd>
        </dl>
      </xs:documentation>
    </xs:annotation>
  </xs:attribute>
</xs:attributeGroup>
```

Complex type code lists

Code lists that are held in a simple type have some limitations; if you can hold your enumeration value in a complex type then you can avoid the limitations and allow for extensible code lists with schema validation. The basic idea is that you specify an element which contains a single element of some external type. The form of the external element can be whatever the code list owner requires, but will often be a simple type. The external schema that defines the code list can contain whatever level of validation is required (tight/loose).

You define a code list type in a separate schema as follows:

```
<xs:simpleType name="CodeContentType">
  <xs:extension base="token">
    <xs:enumeration value="FOO"/>
    <xs:enumeration value="BAR"/>
  </xs:extension>
</xs:simpleType>
<xsd:complexType name="CodeType">
  <simpleContent>
    <xs:extension base="CodeContentType">
      <xs:attribute name="SomeMetaData" type="someMetaDataType"
use="optional"/>
    </xs:extension>
  </simpleContent>
</xsd:complexType>
```

Then you use this type in your schema as follows:

```
<xsd:complexType name="MyCodeContainerType">
  <xsd:choice>
    <xsd:element ref="xxx:CodeType"/> <!- externally defined code list
type -->
    <xsd:element ref="yyy:CodeType"/> <!- externally defined code list
type -->
  </xsd:choice>
</xsd:complexType>
<xsd:element name="CodeContainer" type="tns:MyCodeContainerType"/>
```

The element `CodeContainer` will contain a `CodeType` element from one of the code list schemas xxx or yyy.

Generic containers

Object containers abound in object-oriented (OO) programming—there is often a need to map some polymorphic list or hash table into an equivalent XML structure. Even if your list is currently made up of known elements you may want to enable extension to the types of elements that can be placed in the container.

How should I specify an element which is a container of some set of elements?

If the child elements of the container share some common semantics (i.e., common attributes and subelements) that are known at the time you create the XML schema, then you should use type substitution, for example:

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

The above example forces the use of type substitution via the `abstract="true"` on the complex type.

If the members of the container are defined in disjoint XML schemas (i.e., no type linkage possible), use an <any> wildcard, for example:

```
<xs:element name="Catalogue">
  <xs:complexType>
    <xs:sequence maxOccurs="unbounded">
      <xs:any processContents="strict" namespace="##other" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

If the members are known at design time but you want to allow extensibility to add other elements, populate the above choice with the known elements. For example:

```
<xs:element name="Catalogue">
  <xs:complexType>
    <xs:choice maxOccurs="unbounded">
      <xs:element ref="tns:foo"/>
      <xs:element ref="tns:bar"/>
      <xs:any processContents="strict" namespace="##other"/>
    </xs:choice>
  </xs:complexType>
</xs:element>
```

If you are unsure which method to use then go with the type substitution as it allows for a stronger linkage between the two XML schemas.

Note that `processContents` are set to **`strict`** to indicate that the wildcard elements are part of the recognized semantic information for that XML document. If the elements were not part of the semantic content (for example, a miscellaneous container), the standard `lax` wildcard would be sufficient.

Model groups `<sequence>`, `<choice>`, and `<all>`

When should I use <all> model group?

Never. The `<all>` model groups' limited applicability and unexpected extension semantics should be avoided. Use a <sequence> instead.

DTD

Should I create a DTD to go with my XML schema?

No. Tools support for XML Schema is widespread and DTDs do not support namespaces and do not support embedding one XML document with a DTD in another XML document with a DTD (note that DTDs are also not carried by SOAP messages).

Should I use DTD replicated types in the XML schema datatypes, e.g., `xsd:NOTATION`?

No. You should normally avoid using (or restricting from) NOTATION, ID, IDREF, IDREFS, ENTITY and ENTITIES. The only reasonable use is when converting an existing DTD to an XML schema. (Note that all these types do not have the exact same semantics as their DTD counterparts, so beware.)

Extension vs. restriction for complex types

Should I use complex type restriction in my XML schemas?

No. The restriction of complex types has no analog in object-oriented programming languages or in database schemas. Restriction causes the redefinition of the complex types elements and attributes, which leads to fragility in your XML schema; any change to a super type will require corresponding changes in all restricting subtypes, even those in other schemas.

Type hierarchies vs. composition

Should I build up element content via multilevel subclassing or using composition?

A good rule of thumb is that your `complexType` hierarchy should be simple and small, rarely reaching to 3 or 4 levels deep. You should primarily construct your content models using composition.

Must understand attribute

Should I define a global attribute that will indicate to implementation the criticality of extension elements?

No. Generally it is simpler to follow the best practice of treating wildcard elements and attributes as optional and ignorable information. Elements which are marked as abstract (i.e., forced heads of substitution groups) and elements that have abstract complex types are explicit signals that the actual element or type in the instance document must be understood by target applications or they may reject the document.

Nil

Should I make my elements nillable?

No, just make them optional (`minOccurs=0`).

Elements with simple content

How should I define an element that is going to contain only simple content?

Elements that contain only text may either be defined with that simple type as follows:

```
<xs:element name="salary" type="xs:float"/>
```

Or as a complex type with simple content as follows:

```
<xs:complexType name="salaryType">
  <xs:simpleContent>
    <xs:extension base="xs:float">
      <xs:anyAttribute namespace="##other" processContents="lax"/>
    </xs:extension >
  </xs:simpleContent>
</xs:complexType>
<xs:element name="salary" type="tns:salaryType"/>
```

The second form allows for attributes to be added to the element. In future versions of your XML schema new attributes could be placed in the complex type definition.

For maximum extensibility use the `complexType` with `simpleContent` form.

Schema header

It is good practice to place human/computer readable documentation about the schema in the schema document, for example, owner, change log, etc. This information can be held in a special schema header element which would go in an `annotation/appinfo` element. There is no industry standard for such an element.

Import/include

How should I handle a very large schema document?

The best practice is to break up your schema into logical sections and `<include>` these sections into the main XML schema file. Other namespaces should be `<imported>`.

How can I indicate support for extension schemas?

You can either directly import the new extension schema into the latest version of your XML schema or create a new top level Schema document which includes your original XML schema and imports the extension XML schema[s].

Supplemental schema constraints

I can't express all the constraints that I need using XML Schema language, what should I do?

The XML Schema language is large and complex but does not allow one to specify every constraint that one might like, for example, you can't currently describe:

- Relationships between values of attributes and elements
- Relationships between presence of attributes and elements
- Relationships between some value and presence of some attribute or element

There are other schema languages other than XML Schema that have different semantic coverage. There is widespread use of Schematron <http://www.schematron.com/> and XSL <http://www.w3.org/Style/XSL/> for these purposes, but it is often done in incompatible or ad hoc ways that do not leverage the XML Schema type system to enable reusing and sharing of the constraints.

There is no currently agreed best practice for extending XML Schema. Watch this space.

Summary

This article has enumerated the best practices for creating XML schemas. Many of the XML Schema best practice documents available on the web list several equivalent practices, which leaves the non expert reader with more than enough rope to hang themselves. The best practices listed here are a consistent and reasonable subset of the many good practices in use in the industry. XML schemas that follow these rules will be modular, versionable, extensible, and simple. The appendix contains extensive further comments and explanations on many of the topics covered in this document.

Appendix: rationale behind the best practices

Best practice area	**Comments**
Namespaces: URI, URL vs. URN	XML documents that don't use an XML namespace sit in a global space and will collide with all other documents that don't use an XML namespace. Much like Java™ code in the global namespace, it is used for prototype or testing purposes only and any real application will use a namespace. XML documents use URIs as namespace identifiers. There is no expectation that a URI used as an XML namespace actually resolves to anything. Given all the possible types of URI schemes that could be used for your URI, which one should you use? URNs are the closest semantic fit for namespaces with their emphasis on durability and location independence. The problem with URNs is that none of the standardized URN namespaces (see Official IANA Registry of URN Namespaces <http://www.iana.org/assignments/urn-namespaces>) are a good fit for XML namespaces. The often-referenced and sometimes used `urn:uuid` form is not actually a standard URN namespace! The advantage of using a URL specifically a http URL are as follows: 1. Well known and understood form 2. Easy to ensure uniqueness (you own the web site at that location) 3. If the standards change and an XML namespace does actually need to point to something, we already have an addressable namespace For more insights on the issue read XML Namespace Name: URN or URL? <http://www.xfront.com/URLversusURN.pdf> (PDF, 22 KB).
Attributes vs. elements	The choice of using an attribute vs. an element for holding a text value is similar to the argument over where to place the open curlies "{" in C-style programming languages. What are the facts? 4. Programs don't care about attribute vs. element. It's a human readability thing. 5. Attributes cannot repeat but can contain a list of values. 6. Elements can repeat and can also contain a list of values! 7. Attribute values are "normalized" by the XML processor (see Attribute-Value Normalization). 8. Elements can contain attributes but not vice versa. 9. All attributes (except namespace declarations) could have been elements but not vice versa. XML Schema best practice documents often make strong statements about where attributes should and should not be used (for example, "use attributes only for metadata"). These statements are based on so-called wide usage in the industry. This is demonstrably false as it can be verified by looking at a range of schemas.

Qualified element names	XML Schema has a "feature" which allows the locally-defined elements in an XML schema to omit the namespace qualifier. For example:
	`<pre:foo xmlns:pre="http://www.example.com">`
	`<bar>42</bar></pre:foo>`
	If an XML schema `<bar>` was defined within the definition of `<foo>` (i.e., a local definition),

Best practice area	Comments
	and the `elementFormDefault` value is "unqualified" (or unspecified), and the default namespace in the instance document has not been set, then `<bar>` would be validated against its definition in the XML schema. Some facts: • Like element vs. attributes, this is not something programs care about. • Unqualified local elements are in no namespace. • You can't reuse a local element declaration. • Local elements can't be the head of a substitution group. • You can't use the default namespace declaration in instance documents. • You can't tell from a document fragment of an unqualified local element what kind of element it is. • The same element name can have multiple different local definitions in a single XML schema document. As this best practice document calls for always declaring elements as global, you should never be creating local elements nor worrying about `elementFormDefault`.
Global elements vs. local element	Local elements are not sharable, reusable or extendable (via substitution groups). For maximal reusability and extendibility always define elements globally. There are two added benefits for always defining your elements globally: firstly, your XML schemas become more regular, predictable, and readable; secondly, you will never need to worry about qualifying element names and they will always require qualification whatever the value of `elementFormDefault`.
Named types vs. anonymous types	If complex types or simple types are anonymous, they cannot be the target for type derivation and cannot be reused by later versions of the schema. If elements or attributes contain anonymous simple types, those types cannot be reused or derived. As local attributes cannot be reused, it is doubly important to ensure that their types can be reused.
Russian dolls, salami slices and Venetian blinds	If you read any of the XML Schema best practice documents available on the Internet you may encounter the terms "Russian doll", "salami slice" and "Venetian blind" these refer to forms that you can use to define elements and types in an XML schema. This article contains rules that cause a schema writer to use the "Garden of Eden" form, for more information see Schema Design Rules for UBL...and Maybe for You <http://www.idealliance.org/papers/xml02/dx_xml02/papers/05-01-02/05-01-02.html>.
Extensible content models, wildcards, `any`, constraints (max, min, length, etc.)	The "targeted element extension and general attribute extension" refers to schemas that allow arbitrary elements at certain points. For example, the `xs:appinfo` and `xs:documentation` elements in XML Schema. XML Schema also allows any external attributes on any element in the Schema. You could create you own XML schema that provides a similar style of extensibility. It is less of a good idea to interleave `<xs:any` between elements in your content model, because any extension elements placed at these points cannot be added into an XML schema using extension. This is because when complex types are extended new elements are placed at the end of the content and never in the middle. Further learning: W3C XML Schema Design Patterns: Dealing With Change <http://www.xml.com/pub/a/2002/07/03/schema_design.html>. A good write up on this topic: Creating Extensible Content Models <http://www.xfront.com/ExtensibleContentModels.html>.
Constraints (max, min, length, pattern, etc.)	An XML schema describes what are the valid set of values for a simple type. If you make the set of valid values larger than is strictly necessary, any program which reads XML documents complying with the schema must be able to handle (possibly as an error) the full set of values. If you set `minOccurs` to zero, programs need to be able to handle the absence of the element suitably.

Best practice area	Comments
Graph data structures	Most XML serialization frameworks that *respect XML schemas* do not support the automatic serialization and deserialization of object graphs. You will need to write some application logic to wire up your data structure after it has been deserialized from XML form. Do not use `ID` and `IDREF` as they are inferior to `key/keyref` constructs: Why You Should Favor key/keyref/unique Over ID/IDREF For Identity Constraints <http://www.xml.com/pub/a/2002/11/20/schemas.html?page=3>.
Don't use default/fixed attributes or element values (non-modifying)	The principle here is that validation should not change the Infoset, which can cause all sorts of hard-to-track-down bugs. See Attribute and element defaults <http://www.brics.dk/~amoeller/XML/schemas/xmlschema-defaults.html>.
Mixed content	Very broadly speaking XML schemas are used to specify two different categories of document: data centric and document centric. Data-centric XML documents are those designed to contain structured data that might otherwise be held in objects (in an OO program) or tables (inside a relational database). Document-centric XML documents are those that are designed to contain human language text that is "marked up", using elements and attributes, to indicate formatting or semantic information above and beyond the textual information. The best example of document-centric XML is XHTML but other document-centric schemas do exist for the legal and professional domains.
Code list	A good write-up of how these code list schemes work, can be found in the OASIS Position Paper: Code Lists <http://www.oasis-open.org/committees/ubl/ndrsc/pos/p-maler-codelists-09.doc> document.
Generic container	If you don't know what a generic container is or what you might use one for then I suggest reading Creating Variable Content Container Elements <http://www.xfront.com/VariableContentContainers.pdf> as a primer. Advantages of type substitution over substitution groups: 1. XPath expressions will work with all subtypes that use extension, including those in `key/keyref` 2. Much simpler to write style sheets for your XML document 3. Better implementation support in existing frameworks, e.g., .NET The disadvantage of using the `<any>` container is that you have no way of linking a new element to the container in the original schema but the corresponding advantage is that you can hold elements that are specified separately from your schema.
Model groups `<sequence>`, `<choice>`, & `<all>`	The `<all>` model group is not useful as it has limited backward compatibility when redefining or extending the complex types. Elements added in complex type extensions are not permitted in random order but must be placed at the end. If a later version of the XML schema wants to change an existing member of the `<all>` to have `maxOccurs > 1`, backward compatibility is greatly complicated as the new complex type can't use the `<all>` construct any more.
Nil	It is not a good idea to try and use the `tri` value semantics of both optional and nillable elements as this is generally not support by XML serialization frameworks.
Type hierarchies vs. composition	Design by subclassing is important in XML Schema design as it is in OO languages, and in a similar way designers have moved away from complex multi-leveled type hierarchy to a simple 2-3 level hierarchy where most of the structural assembly is done by composition. A good write-up on this issue can be found in the Composition versus Subclassing <http://www.xfront.com/composition-versus-subclassing.html> guidelines.
Schema header	The use of an agreed header in an XML schema is a good idea and helps with running an automated schema repository.

Best practice area	Comments
Import/include	Extension schemas are XML schemas that extend your namespace by providing (in another namespace) types, elements, and attributes designed to provide extra features and work with your schema.

Background reading and alternative rationales

- W3C XML Schema Design Patterns: Avoiding Complexity <http://www.xml.com/pub/a/2002/11/20/schemas.html>

Acknowledgement

I would like to acknowledge Brian Magick, for his careful and thought-provoking review.

About the author

David Stephenson is an HP employee. Previously, he worked for four years on the product generation side in the area of enterprise middleware solutions technology and has extensive experience of web services and Java. Before that David worked for HP Laboratories on distributed systems technology.

For more information

- Dev Resource Central (HP Software Developer Program portal) <http://devresource.hp.com/drc/index.jsp>
- XML fundamentals checklist < http://devresource.hp.com /drc/resources/xmlfundCklist/index.jsp>
- Extensible Markup Language <http://www.w3.org/XML/> (W3C)

Call to action

Visit HP's Invent Online < http://devresource.hp.com /drc/invent/invent.jsp> to view information on educational webcasts delivered by HP experts and partners. Webcasts target developers and present information on web services development, application management, identity management, and a variety of related technologies, including XML, WSDL, and SOAP.

Enhance your ability to create web services and managed applications by signing up to receive the HP Developer News newsletter. Subscribers receive links to white papers, info on the latest tools and downloads, and technical tips all in one monthly e-mail message. Subscribe <http://www.hpdev.com/.docs/pg/5> to get the next newsletter.