

Extension options:

- Allow (in selected locations) any Element of any type
- Allow selected element (typically called “Extension”) to have any type
- Add extra data (attributes or child elements) to an existing Element
 - Extend using xsi:type
 - Extend using substitution group
- Redefine an existing element or type

Some XML Schema guidelines recommend adding to every complex type an `xsd:anyType` element to contain extensions. Instance documents can be validated against this schema, but the extended data is not being checked.

One solution is to tease out and individually validate against supplemental schemas the many xml fragments that are ignored by `xsd:anyType`. This can be done by XSLT or by clever use of validating parsers. You may need to select a supplemental schema based on the parent element, but even if your extended data is all defined in one schema, you still need specialized code.

Another option is to create an extension schema that regards the `xsd:anyType` elements as the heads of substitution groups. This is only an option if the various `xsd:anyType` elements are globally declared and have different names, depending on the complex type that contains them: if all of your `xsd:anyType` elements are locally defined or named “extraData”, you are sunk. Furthermore this solution is ugly because (presuming we have a straightforward mapping between XML elements and objects) it splits the attributes and associations of one object between two objects: the XML representation is replaced subclassing with an association. These split object mappings become even more annoying if you eventually need to extend the extension.

Yet another option is to validate against a new schema that redefines the original elements to add typed data (rather than `xsd:anyType`). The element and complex type names don’t change, only their definitions change. This seems like a recipe for confusion.

An arguably better way is to use extension. This is analogous to single-inheritance subclassing. Extensions define new complex types. There are two ways to take advantage of these new types:

- Define a new element that can replace the old one: this new element uses the extended type instead of the original type.

- Keep the element name the same, and use the new type anyway. The validator will gag on the extended data, unless you give it a clue you are replacing the type with a new type. That clue is the `xsi:type` attribute.

Superficially, both of these mechanisms appear to suffer from a fatal flaw: what prevents a miscreant from sneaking in a type that you don't want? Actually it's not a problem at all. The whole reason of using extensions instead of `xsd:anyType` is that you want to validate the new data in a way that maps cleanly onto inheritance. That means you have a schema to validate against. This new schema precisely defines both the extra data, and where it can appear. When you are using `xsi:type`, an instance document can not just specify any random type... it needs to specify some type that extends the original, or the validator will flunk the instance. Likewise if you are using a substitute element, the substitute must specify a type that extends the original, or the validator will again reject the instance.

Extension does have some pitfalls you should be aware of:

- If Acme defines a schema that extends a base standard, and their product sends an instance document to a product made by Ace, the Ace device needs to validate against a schema that includes the Acme extensions. That shouldn't be a problem, and it shouldn't be a surprise: we *wanted* to validate this extended data, right? Of course the Ace product is free to ignore the extra (validated) Acme data. Please observe that schemas should not be hardwired anyway: the user should be able to specify which schemas should be used. In particular the user should be able to specify (on both machines) a master schema that includes both the Acme and Ace extensions, as well as the base schema. This way, devices made by Ace send out data with extra information that only another Ace machine can really appreciate, but which both Ace and Acme machines can validate.
- I've been told that Microsoft's validating parser does not currently support `xsi:type`, but the next version is scheduled to do so.
- The base complex type being extended can't use `xsd:all`, it needs to use `xsd:sequence` (a schema validator will tell you if this is a problem).
- Extended schemas can become nondeterministic (a schema validator will tell you). This may be a problem for document-centric XML, but it should not be a problem for schemas based on information models, because role names are unique in any decent information model.

I have provided examples of several methods of extension in separate directories. The schemas in these files have been validated against the IBM Schema Quality Checker (<http://www.alphaworks.ibm.com/tech/xmlsqc>). Instances have been validated against Sun's Multi-schema Validator (<http://www.sun.com/software/xml/developers/multischema/>).

=====`Extension_xsiType`=====

This example shows a Catalog containing Products: it uses `xsi:type` for subclassing within the base schema (Hardware and Software inherit from Product), as well as for extension by an additional schema.

Base.xsd contains the original un-extended schema.

ValidExtension.xsd and InvalidExtension.xsd are very similar. The difference is that InvalidExtension attempts to extend Hardware, which is defined with `final='extension'`.

Both of the extension schemas use `xsd:import`. I think it is a good idea to keep the extensions in a different namespace, but you do not have to do that. Use `xsd:include` instead of `xsd:import` if you crave the excitement stemming from the widespread confusion and hysteria that would result from redefined (and therefore ambiguous) namespaces.

valid_instance.xml is unextended: it can be validated against either Base.xsd or ValidExtension.xsd.

valid_instance_extended.xml and valid_instance_extended_diffNS.xml need to be validated against ValidExtension.xsd. These instances have identical data, but treat namespaces differently:

valid_instance_extended.xml uses Base.xsd as the default namespace, and valid_instance_extended_diffNS.xml uses the extension as the default namespace.

invalid_instance naughtily attempts to use a ShippingInfo as an extension of Product, and even more brazenly attempts to use a Wombat (which isn't even defined in the schema) as an extension of Product. The msv error messages are not terribly helpful, although the line numbers of the problems are correctly identified.

===== Extension_substitutionGroup =====

These files are all very similar, except that the substitution group mechanism is used for extension instead of `xsi:type`.

Base.xsd differs from the previously discussed version in that Hardware and Software no longer inherit from Product: instead a Catalog element is composed of Hardware and Software elements, and the contents of Product have been unpleasantly duplicated in Hardware and Software. Inheritance was removed so that this example would run under tools that don't support `xsi:type`, and so that there would be explicit Hardware and Software elements to act as substitution group heads. Also `final='extension'` was removed from the definition of the Hardware complex type so that substitution group blocking could be clearly illustrated by InvalidExtension.xsd.

ValidExtension.xsd differs from the previously described version only by the additional definition of a Vaporware element that can substitute for a Software element.

InvalidExtension.xsd differs only by the additional definition of a Mineral element that could substitute for a Hardware element, if only Hardware did not have a block='substitution' attribute.

===== Extension_anyType =====

Similar to Extension_substitutionGroup except that the substitution group head is the extension element of type xsd:anyType.