The Darwin Information Typing Architecture (DITA) provides a way for documentation authors and architects to create collections of typed topics that can be easily assembled into various delivery contexts. Topic specialization is the process by which authors and architects can define topic types, while maintaining compatibility with existing style sheets, transforms, and processes. The new topic types are defined as an extension, or delta, relative to an existing topic type, thereby reducing the work necessary to define and maintain the new type.

The point of the XML-based Darwin Information Typing Architecture (DITA) is to create modular technical documents that are easy to reuse with varied display and delivery mechanisms, such as helpsets, manuals, hierarchical summaries for small-screen devices, and so on. This article explains how to put the DITA principles into practice with regards to the creation of a DTD and transforms that will support your particular information types, rather than just using the base DITA set of concept, task, and reference.

Topic specialization is the process by which authors and architects define new topic types, while maintaining compatibility with existing style sheets, transforms, and processes. The new topic types are defined as an extension, or delta, relative to an existing topic type, thereby reducing the work necessary to define and maintain the new type.

This document assumes that you already know what DITA is; if you need a basic introduction, see the companion roadmap article, Introduction to the Darwin Information Typing Architecture. The examples used in this paper use XML DTD syntax and XSLT; if you need background on these subjects, see Resources.

Architectural context

In SGML, architectural forms are a classic way to provide mappings from one document type to another. Specialization is an architectural-forms-like solution to a more constrained problem: providing mappings from a more specific topic type to a more general topic type. Because the specific topic type is developed with the general topic type in mind, specialization can ignore many of the thornier problems that architectural forms address. This constrained domain makes
specialization processes relatively easy to implement and maintain. Specialization also provides support for multi-level or hierarchical specializations, which allow more general topic types to serve as the common denominator for different specialized types.

The specialization process was created to work with DITA, although its principles and processes apply to other domains as well. This will make more sense if you consider an example: Given specialization and a generic DTD such as HTML, you can create a new document type (call it MyHTML). In MyHTML you could enforce site standards for your company, including specific rules about forms layout, heading levels, and use of font and blink tags. In addition, you could provide more specific structures for product and ordering information, to enable search engines and other applications to use the data more effectively.

Specialization lets MyHTML be defined as an extension of the HTML DTD, declaring new element types only as necessary and referencing HTML's DTD for shared elements. Wherever MyHTML declares a new element, it includes a mapping back to an existing HTML element. This mapping allows the creation of style sheets and transforms for HTML that operate equally well on MyHTML documents. When you want to handle a structure differently (for example, to format product information in a particular way), you can define a new style sheet or transform that holds the extending behavior, and then import the standard style sheet or transform to handle the rest. In other words, new behavior is added as extensions to the original style sheet, in the same way that new constraints were added as extensions to the original DTD or schema.

**Specializing information types**

The Darwin Information Typing Architecture is less about document types than information types. A document is considered to be made up of a number of topics, each with its own information type. A topic is, simply, a chunk of information consisting of a heading and some text, optionally divided into sections. The information type describes the content of the topic: for example, the type of a given topic might be "concept" or "task."

DITA has three types of topic: a generic topic, or information-typed concept, task, and reference topics. Concept, task, and reference topics can all be considered specializations of topic:

**Figure 1. Three information types, as specializations of topic**

Additional information types can be added to the architecture as specializations of any of these three basic types, or as a peer specialization directly off of topic -- and any of these additional specializations can, in turn, be specialized:
Each new information type is defined as an extension of an existing information type: The specializing type inherits, without duplication, any common structures; and the specializing type provides a mapping between its new elements and the general type’s existing elements. Each information type is defined in its own DTD module, which defines only the new elements for that type. A document that consists of exactly one information type (for example, a task document in a help web) has a document type defined by all the modules in the information type’s specialization hierarchy (for example, task.mod and topic.mod). A document type with multiple information types (for example, a book consisting of concepts, tasks, and reference topics) includes the modules for each of the information types used, as well as the modules for their ancestors (concept.mod, task.mod, reference.mod, plus their ancestor topic.mod).

Because information type declarations are separated into modules, you can define new information types without affecting ancestor types. This separation gives you the following benefits:

- Reduces maintenance costs: Each authoring group maintains only the elements that it uniquely requires
- Increases compatibility: The core information types can be centrally maintained, and changes to the core types are reflected in all specializing types
- Distributes control: Reusability is controlled by the reuser, instead of by the author; adding a new type does not affect the maintenance of the core type, and does not affect other users of different types

Any information-typed topic belongs to multiple types. For example, an API description is, in more general terms, a reference topic.

Specialization example: Reference topic

Consider the specialization hierarchy for a reference topic:
Table 1 expresses the relationship between the general elements in `topic` and the specific elements in `reference`. Within the table, the columns, rows, and cells indicate information types, element mappings, and elements. Table 2 explains the relationships in detail to help you interpret Table 1.

**Table 1. Relationships between topic and a specialization based on it**

<table>
<thead>
<tr>
<th>Topic</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>(topic.mod)</td>
<td>(reference.mod)</td>
</tr>
<tr>
<td>topic</td>
<td>reference</td>
</tr>
<tr>
<td>title</td>
<td></td>
</tr>
<tr>
<td>body</td>
<td>refbody</td>
</tr>
<tr>
<td>simptable</td>
<td>properties</td>
</tr>
<tr>
<td>section</td>
<td>refsyn</td>
</tr>
</tbody>
</table>

**Table 2. How to interpret Table 1.**

<table>
<thead>
<tr>
<th>Structure</th>
<th>Associations</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Columns</strong></td>
<td>The <code>Topic</code> column shows basic <code>topic</code> structure, which comprises a title and body with optional sections, as declared in a DTD module called <code>topic.mod</code>. The <code>Reference</code> column shows a more specialized structure, with <code>reference</code>, <code>refbody</code> replacing body, and <code>refsyn</code> replacing <code>section</code>; these new elements are declared in a DTD module called <code>reference.mod</code>.</td>
</tr>
<tr>
<td><strong>Rows</strong></td>
<td>Each row represents a mapping between the elements in that row. The elements in the <code>Reference</code> column specialize the elements in the <code>Topic</code> column. Each general element also serves as a category for more specialized elements in the same row. For example, <code>reference's refsyn</code> is a kind of <code>section</code>.</td>
</tr>
</tbody>
</table>
| **Cells** | Each cell in a column represents the following possibilities in relation to the cell to its left:  
  - A blank cell: The element in the cell to the left is reused as-is. For example, a `referencetitle` is the same as a `topic title`, and `topic's` declaration of the `title` element can be used by `reference`.  
  - A full cell: An element that is specific to the current type replaces the more general element to the left. For example, in `reference`, `refbody` replaces the more general body.  
  - A split row: Two or more specialized elements replace the more general element to the left. For example, `reference` replaces `section` with the more specific `synsect` (syntax) and `section`. |
• A split row with a blank cell: The new specializations are in addition to the more general element, which remains available in the specialized type. For example, `reference` adds properties as a special type of `simpletable (dl)`, but the general kind of `simpletable` remains available in `reference`.

### The reference type module

Listing 1 illustrates not the actual `reference.mod` content, but a simplified version based on Table 1. The use of entities in the content models support domain specialization, as described in the domain specialization article (see Resources).

**Listing 1**

```xml
<!ELEMENT reference ((%title;), (%prolog;)?, (%refbody;), (%info-types;)* )>
<!ELEMENT refbody (%section; | refsyn | %simpletable; | properties)*>
<!ELEMENT properties ((%sthead;)?, (%strow;)+) >
<!ELEMENT refsyn (%section;)* >
```

Most of the content models declared here depend on elements or entities declared in `topic.mod`. Therefore, if `topic`'s structure is enhanced or changed, most of the changes will be picked up by `reference` automatically. Also, the definition of `reference` remains simple: It doesn't have to redefine any of the content that it shares with `topic`.

### Adding specialization attributes

To expose the element mappings, you can add an attribute to each element that shows its mappings to more general types.

**Listing 2**

```xml
<!ATTLIST reference class CDATA "- topic/topic reference/reference ">
<!ATTLIST refbody class CDATA "- topic/body reference/refbody ">
<!ATTLIST properties class CDATA "- topic/simpletable reference/properties ">
<!ATTLIST refsyn class CDATA "- topic/section reference/refsyn ">
```

Later on, I'll talk about how to take advantage of these attributes when you write an XSL transform. See the appendix for a more in-depth description of the class attribute.

### Creating an authoring DTD

Now that you've defined the type module (which declares the newly typed elements and their attributes) and added specialization attributes (which map the new type to its ancestors), you can assemble an authoring DTD.

**Listing 3**

```xml
<!--Redefine the infotype entity to exclude other topic types-->
ENTITY % info-types "reference">
<!--Embed topic to get generic elements -->
ENTITY % topic-type SYSTEM "topic.mod">
%topic-type;
<!--Embed reference to get specific elements -->
ENTITY % reference-type SYSTEM "reference.mod">
%reference-type;
```
Specialization example: API description

Now, I'll show you how to create a more specialized information type: API descriptions, which are a kind of (and therefore specialization of) reference topic.

Figure 4. A more specialized information type, API description

Table 3 shows part of the specialization for an information type called APIdesc, for API description. As before, each column represents an information type, with specialization occurring from left to right. That is, each information type is a specialization of its neighbor to the left. Each row represents a set of mapped elements, with more specific elements to the right mapping to more general equivalents to the left.

As before, each cell specializes the contents of the cell to its left:

- A blank cell: The element to the left is picked up by the new type unchanged. For example, simpletable and refsyn are available in an API description.
- A full cell: The element to the left is replaced by a more specific one. For instance, APIname replaces title.
- A split row with a blank cell: New elements are added to the elements on the left. For example, the API description adds a usage section as a peer of the refsyn and section elements.

Table 3. Summary of APIdesc specialization

<table>
<thead>
<tr>
<th>Topic</th>
<th>Reference</th>
<th>APIdesc</th>
</tr>
</thead>
<tbody>
<tr>
<td>(topic.mod)</td>
<td>(reference.mod)</td>
<td>(APIdesc.mod)</td>
</tr>
<tr>
<td>topic</td>
<td>reference</td>
<td>APIdesc</td>
</tr>
<tr>
<td>title</td>
<td></td>
<td>APIname</td>
</tr>
<tr>
<td>body</td>
<td>refbody</td>
<td>APIbody</td>
</tr>
<tr>
<td>simpletable</td>
<td>properties</td>
<td>parameters</td>
</tr>
<tr>
<td>section</td>
<td>refsyn</td>
<td></td>
</tr>
<tr>
<td></td>
<td>usage</td>
<td></td>
</tr>
</tbody>
</table>
The APIdesc module

Here you can see that the content for an API description is actually much more restricted than the content of a general reference topic. The sequence of syntax, then usage, then parameters is now imposed, followed by optional additional sections. This sequence is a subset of the allowable structures in a reference topic, which allows any sequence of syntax, properties, and sections. In addition, the label for the usage section is now fixed as usage, taking advantage of the spectitle attribute of section (which is there for exactly this kind of usage): With the spectitle attribute providing the section title, you can also get rid of the title element in usage's content model, making use of the predefined section.notitle.cnt entity.

Listing 4

```xml
<!ELEMENT APIdesc (APIname, (%prolog;)?, APIbody,(%info-types;)* )>
<!ELEMENT APIname (%title.cnt;)*>
<!ELEMENT APIbody (refsyn,usage,parameters,(%section;)*>
<!ELEMENT usage (%section.notitle.cnt;)* >
<!ELEMENT parameters ((%sthead;)?, (%strow;)+)
```

Adding specialization attributes

Every new element now has a mapping to all its ancestor elements.

Listing 5

```xml
<!ATTLIST APIdesc
    class CDATA "- topic/topic reference/reference APIdesc/APIdesc " >
<!ATTLIST APIname
    class CDATA "- topic/title reference/title APIdesc/APIname " >
<!ATTLIST APIbody
    class CDATA "- topic/body reference/refbody APIdesc/APIbody" >
<!ATTLIST parameters
    class CDATA "- topic/simpletable reference/properties APIdesc/parameters ">
<!ATTLIST usage
    class CDATA "- topic/section reference/section APIdesc/usage ">
```

Note that APIdesc explicitly identifies its equivalent in both reference and topic, even though they are the same (title) in both cases. In the same way, usage explicitly maps to section in both reference and topic. This explicit identification makes it easier for processes to keep track of complex mappings. Even if you had a specialization hierarchy 10 levels deep or more, the attributes would still allow unambiguous mappings to each ancestor information type.

Authoring DTDs

Now that you’ve defined the type module (which declares the newly typed elements and their attributes) and added specialization attributes (which map the new type to its ancestors), you can assemble an authoring DTD.
Listing 6

<!--Redefine the infotype entity to exclude other topic types-->
ENTITY % info-types "APIdesc">
<!ENTITY % topic-type SYSTEM "topic.mod">
%topic-type;
<!--Embed topic to get generic elements -->
ENTITY % topic-type SYSTEM "topic.mod">
%topic-type;
<!--Embed reference to get more specific elements -->
ENTITY % reference-type SYSTEM "reference.mod">
%reference-type;
<!--Embed APIdesc to get most specific elements -->
ENTITY % APIdesc-type SYSTEM "APIdesc.mod">
%APIdesc-type;

Working with specialization

After a specialized type has been defined and the necessary attributes have been declared, they can provide the basis for the following operations:

- Applying a general style sheet or transform to a specialized topic type
- Generalizing a topic of a specialized type (transforming it into a more generic topic type)
- Specializing a topic of a general type (transforming it into a more specific topic type -- to be used only when a topic was originally authored in specialized form and has gone through a general stage without breaking the constraints of its original form)

Applying general style sheets or transforms

Because content written in a new information type (such as APIdesc) has mappings to equivalent or less restrictive structures in preexisting information types (such as reference and topic), the preexisting transforms and processes can be safely applied to the new content. By default, each specialized element in the new information type will be treated as an instance of its general equivalent. For example, in APIdesc the <usage> element will be treated as a topic <section> element that happens to have the fixed label "Usage".

To override this default behavior, an author can simply create a new, more specific rule for that element type and then import the default style sheet or transform, thus extending the behavior without directly editing the original style sheet or transform. This reuse by reference reduces maintenance costs (each site maintains only the rules it uniquely requires) and increases consistency (because the core transform rules can be centrally maintained, and changes to the core rules will be reflected in all other transforms that import them). Control over reuse has moved from the author of the transform to the reuser of the transform.

The rest of this section assumes knowledge of XSLT, the XSL Transformations language.

Requirements

This process works only if the general transforms have been enabled to handle specialized elements, and if the specialized elements include enough information for the general transform to handle them.
• **Requirement 1: Mapping attributes** To provide the specialization information, you need to add specialization attributes, as outlined previously. After you include the attributes in your documents, they are ready to be processed by specialization-aware transforms.

• **Requirement 2: Specialization-aware transforms** For the transform, you need template rules that check for a match against both the element name and the attribute value.

### Listing 7

```xml
<xsl:template match="*[contains(@class,' topic/simpletable ')]">
  <!--matches any element that has a class attribute that mentions topic/simpletable-->  
  <!--do something-->  
</xsl:template>
```

### Example: overriding a transform

To override the general transform for a specific element, the author of a new information type can create a transform that declares the new behavior for the specific element and imports the general transform to provide default behavior for the other elements.

For example, an `APIdesc` specialized transform could allow default handling for all specialized elements except `parameters`:

### Listing 8

```xml
<xsl:import href="general-transform.xsl"/>
<xsl:template match="*[contains(@class,' APIdesc/parameters ')]">
  <!--do something-->  
  <xsl:apply-templates/>
</xsl:template>
```

Both the preexisting `referenceproperties` template rule and the new `parameters` template rule match when they encounter a `parameters` element (because the `parameters` element is a specialized type of `referenceproperties` element, and its class attribute contains both values). However, because the `parameters` template is in the importing style sheet, the new template takes precedence.

### Generalizing a topic

Because a specialized information type is also an instance of its ancestor types (an `APIdesc` is a reference topic is a topic), you can safely transform a specialized topic to one of its more generic ancestors. This upward compatibility is useful when you want to combine sets of documentation from two sources, each of which has specialized differently. The ancestor type provides a common denominator that both can be safely transformed to. This compatibility may also be useful when you have to feed topics through processes that are not specialization-aware: For example, a publication center that charges for each document type or uses non-DTD-aware processes could be sent a generalized set of documents, so that they only support one document type or set of markup. However, wherever possible, you should use specialization-aware processes and transforms, so that you can avoid generalizing and process your documents in their more descriptive, specialized form.
To safely generalize a topic, you need a way to map from your information type to the target information type. You also need a way to preserve the original type in case you need round-tripping later.

The class attribute that was introduced previously serves two purposes. It provides:

- The information needed to map
- A way to preserve the information to allow round-tripping

Each level of specialization has its own set of class attributes, which in the end provide the full specialization hierarchy for all specialized elements.

Consider the APIdesc topic in Listing 9.

**Listing 9**

```xml
<APIdesc>
  <APIname>AnAPI</APIname>
  <APIbody>
    <refsyn>AnAPI (parm1, parm2)</refsyn>
    <usage spectitle="Usage">Use AnAPI to pass parameters to your process.</usage>
    <parameters>...
      </parameters>
  </APIbody>
</APIdesc>
```

With the class attributes exposed (all values are provided as defaults by the DTD):

**Listing 10**

```xml
<APIdesc class="- topic/topic reference/reference APIdesc/APIdesc ">
  <APIname class="- topic/title reference/title APIdesc/APIname ">AnAPI
</APIname>
  <APIbody class="- topic/body reference/refbody APIdesc/APIbody ">
    <refsyn class="- topic/section reference/refsyn ">AnAPI(parm1, parm2)</refsyn>
    <usage class="- topic/section reference/section APIdesc/usage ">
      <p class="- topic/p ">Use AnAPI to pass parameters to your process.</p>
    </usage>
    <parameters class="topic/simpletable reference/properties APIdesc/parameters ">
      ...
    </parameters>
  </APIbody>
</APIdesc>
```

From here, a single template rule can transform the entire APIdesc topic to either a reference or a generic topic. The template rule simply looks in the class attribute for the ancestor element name, and renames the current element to match.

After a transform to topic, the code should look something like Listing 11.
Listing 11

```xml
<topic class="- topic/topic reference/reference APIdesc/APIdesc ">
  <title class="- topic/title reference/title APIdesc/APIname ">AnAPI</title>
  <body class="- topic/body reference/refbody APIdesc/APIbody ">
    <section class="- topic/section reference/section APIdesc/usage ">
      <p class="- topic/p ">Use AnAPI to pass parameters to your process.</p>
    </section>
    <simpletable class="topic/simpletable reference/properties APIdesc/parameters ">
      ...
    </simpletable>
  </body>
</topic>
```

Even after generalization, specialization-aware transforms can continue to treat the topic as an APIdesc because the transforms can look in the class attribute for information about the element type hierarchy.

From here, it is possible to round-trip by reversing the transformation (looking in the class attribute for the specializing element name, and renaming the current element to match). Whenever the class attribute doesn't list the target (the first section has no APIdesc value), the element is changed to the last value listed (so the first section becomes, accurately, a refsyn).

However, if anyone changes the structure of the content while it is a generic topic (as by changing the order of sections), the result might not be valid anymore under the specialized information type (which in the APIdesc case enforces a particular sequence of information in the APIbody). So although mapping to a more general type is always safe, mapping back to a specialized type can be problematic: The specialized type has more rules, which make the content specialized. But those rules aren't enforced while the content is encoded more generally.

**Specializing a topic**

It is relatively trivial to specialize a general topic if the content was originally authored as a specialized type. However, a more complex case can result if you have authored content at a general level that you now want to type more precisely.

For example, suppose that you create a set of reference topics. Then, having analyzed your content, you realize that you have a consistent pattern. Now you want to enforce this pattern and describe it with a specialized information type (for example, API descriptions). In order to specialize, you need to first create the target DTD and then add enough information to your content to allow it to be migrated.

You can put the specializing information in either of two places:

- Add it to the class attribute. You need to be careful to get the order correct and include all ancestor type values.
• Give the name of the target element in an `outputclass` attribute, migrate based on that value, and add the `class` attribute values afterward.

In either case, before migration you can run a validation transform that looks for the appropriate attribute, then checks that the content of the element will be valid under the specialized content model. You can use a tool like Schematron to generate both the validating transform and the migrating transform, or you can migrate first and use the specialized DTD to validate that the migration was successful.

### Specializing with schemas

Like the XML DTD syntax, the XML Schema language is a way of defining a vocabulary (elements and attributes) and a set of constraints on that vocabulary (such as content models, or fixed versus implied attributes). It has a built-in specialization mechanism, which includes the capability to restrict allowable specializations. Using the XML Schema language instead of DTDs makes it much easier to validate that specialized information types represent valid subsets of generic types, which ensures smooth processing by generic translation and publishing transforms.

Unlike DTDs, XML schemas are expressed as XML documents. As a result, they can be processed in ways that DTDs cannot. For example, you can maintain a single XML schema and then use XSL to generate two versions:

- An authoring version of it that eliminates any fixed attributes and any overridden elements
- A processor-ready version of it that includes the class attributes that drive the translation and publishing transforms

However, XML schemas are not yet popular enough to adopt wholeheartedly. The main problems are a lack of authoring tools and incompatibilities between the implementations of an evolving standard. These problems should be remedied by the industry over the next year or so, as the standard is finalized and schemas become more widely adopted and supported.

### Summary

You can create a specialized information type by using this general procedure:

1. Identify the elements that you need.
2. Identify the mapping to elements of a more general type.
3. Verify that the content models of specialized elements are more restrictive than their general equivalents.
4. Create a type module file that holds your specialized element and attribute declarations (including the `class` attribute).
5. Create an authoring DTD file that imports the appropriate type modules.

You can create specialized XSL transforms by using this general procedure:

1. Create a new transform for your information type.
2. Import the existing transform that you want to extend.
3. Identify the elements that you need to treat specially.
4. Add template rules that match those elements, based on their class attribute content.

Appendix: Rules for specialization

Although you could create a new element equivalent for any tag in a general DTD, this work is useless to you as an author unless the content models that would include the tag are also specialized. In the APIdesc example, the `parameters` element is not valid content anywhere in topic or reference. For it to be used, you need to create valid contexts for parameters, all the way up to the topic-level container. To expose the `parameters` element to your authors, you need to specialize the following parts:

- A body element, to allow parameters as valid content (giving us APIdesc)
- A topic element, to allow the specialized body (giving us APIdesc)

This domino effect can be avoided by using domain specialization. If you truly just want to add some new variant structures to an existing information type, use domain specialization instead of topic specialization (see "Specializing domains in DITA" in Resources).

To ensure that the specialized elements are more constrained than their general equivalents (that is, that they allow a proper subset of the structures that the general equivalent allows), you need to look at the content model of the general element. You can safely change the content model of your specialized element as shown in Table 4:

Table 4. Summary of specialization rules

<table>
<thead>
<tr>
<th>Content type</th>
<th>Example (Special specializing General)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Required</td>
<td>Rename only</td>
</tr>
<tr>
<td></td>
<td><code>&lt;!ELEMENT General(a)&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;!ELEMENT Special(a.1)&gt;</code></td>
</tr>
<tr>
<td>Optional (?)</td>
<td>Rename, make required, or delete</td>
</tr>
<tr>
<td></td>
<td><code>&lt;!ELEMENT General(a?)&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;!ELEMENT Special(a.1?)&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;!ELEMENT Special(a.1)&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;!ELEMENT Special EMPTY&gt;</code></td>
</tr>
<tr>
<td>One or more (+)</td>
<td>Rename, make required, split into a required element plus others, split into one or more elements plus others.</td>
</tr>
<tr>
<td></td>
<td><code>&lt;!ELEMENT General(a+)&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;!ELEMENT Special(a.1+)&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;!ELEMENT Special(a.1)&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;!ELEMENT Special(a.1,a.2,a.3+,a.4*)&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;!ELEMENT Special(a.1+,a.2,a.3*)&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;!ELEMENT Special(a.1*,a.2?,a.3*)&gt;</code></td>
</tr>
<tr>
<td>Zero or more (*)</td>
<td>Rename, make required, make optional, split into a required element plus others, split into an optional element plus others, split into one-or-more plus others, split into zero-or-more plus others, or delete</td>
</tr>
<tr>
<td></td>
<td><code>&lt;!ELEMENT General(a*)&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;!ELEMENT Special(a.1*)&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;!ELEMENT Special(a.1)&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;!ELEMENT Special(a.1?,a.2,a.3+,a.4*)&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;!ELEMENT Special(a.1?,a.2,a.3*,a.4*)&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;!ELEMENT Special(a.1*,a.2?,a.3*)&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;!ELEMENT Special EMPTY&gt;</code></td>
</tr>
<tr>
<td>Either-or</td>
<td>Rename, or choose one</td>
</tr>
<tr>
<td></td>
<td>`&lt;!ELEMENT General (a</td>
</tr>
<tr>
<td></td>
<td>`&lt;!ELEMENT Special (a.1</td>
</tr>
<tr>
<td></td>
<td><code>&lt;!ELEMENT Special (a.1)&gt;</code></td>
</tr>
</tbody>
</table>
Extended example

You have a general element `General`, with the content model `(a,b?,(c|d+))`. This definition means that a `General` always contains element `a`, optionally followed by element `b`, and always ends with either `c` or one or more `d`s.

Listing A. The content model for the general element `General`

```xml
<!ELEMENT General (a,b?,(c|d+))>
```

When you specialize `General` to create `Special`, its content model must be as or more restrictive: It cannot allow more things than `General` did, or you will not be able to map upward or guarantee the correct behavior of general processes, transforms, or style sheets.

Leaving aside renaming (which is always allowed, and simply means that you are also specializing some of the elements that `Special` can contain), here are some valid changes that you could make to the content model of `Special`, resulting in the same or more restrictive content rules:

Listing B. A valid change to the model `Special`, making `b` mandatory

```xml
<!ELEMENT Special (a,b,(c|d))>
```

`Special` now requires `b` to be present, instead of optional, and allows only one `d`. It safely maps to `General`.

Listing C. A valid change to the model `Special`, making `c` mandatory and disallowing `d`

```xml
<!ELEMENT Special (a,b?,c)>  
```

`Special` now requires `c` to be present, and no longer allows `d`. It safely maps to `General`.

Listing D. A valid change to the model `Special`, making three specializations of `d` mandatory

```xml
<!ELEMENT Special (a,b?,d1,d2,d3)>  
```

`Special` now requires three specializations of `d` to be present, and does not allow `c`. It safely maps to `General`.

Details of the class attribute

Every element must have a `class` attribute. The `class` attribute starts and ends with white space, and contains a list of blank-delimited values. Each value has two parts: The first part identifies a topic type, and the second part (after a `/`) identifies an element type. The `class` attribute value should be declared as a default attribute value in the DTD. Generally, it should not be modified by the author.
Example:

```xml
<appstep class="- topic/li task:step bctask/appstep ">
A specialized
step</appstep>
```

When a specialized type declares new elements, it must provide a class attribute for the new element. The class attribute must include a mapping for every topic type in the specialized type's ancestry, even those in which no element renaming occurred. The mapping should start with topic, and finish with the current element type.

Example:

```xml
<appname class="- topic/kwd task/kwd bctask/appname ">
```

This is necessary so that generalizing and specializing transforms can map values simply and accurately. For example, if `task/kwd` was missing as a value, and I decided to map this `bctask` up to a task topic, then the transform would have to guess whether to map to `kwd` (appropriate if `task` is more general, which it is) or leave as `appname` (appropriate if `task` were more specialized, which it isn't). By always providing mappings for more general values, we can then apply the simple rule that missing mappings must by default be to more specialized values, which means the last value in the list is appropriate. While this example is trivial, more complicated hierarchies (say, five levels deep, with renaming occurring at two and four only) make this kind of mapping essential.

A specialized type does not need to change the `class` attribute for elements that it does not specialize, but simply reuses by reference from more generic levels. For example, since `task` and `bctask` use the `p` element without specializing it, they don't need to declare mappings for it.

A specialized type only declares `class` attributes for the elements that it uniquely declares. It does not need to declare `class` attributes for elements that it reuses or inherits.

**Using the class attribute**

Applying an XSLT template based on `class` attribute values allows a transform to be applied to whole branches of element types, instead of just a single element type.

Wherever you would check for an element name (any XPath statement that contains an element name value), you need to enhance this to instead check the contents of the element's `class` attribute. Even if the element is unrecognized, the `class` attribute can let the transform know that the element belongs to a class of known elements, and can be safely treated according to their rules.

Example:

```xml
<xsl:template match="*[contains(@class,' topic/li ')]">
```

This match statement will work on any `li` element it encounters. It will also work on `step` and `appstep` elements, even though it doesn't know what they are specifically, because the `class` attribute tells the template what they are generally.
This match statement won't work on generic li elements, but it will work on both step elements and appstep elements; even though it doesn't know what an appstep is, it knows to treat it like a step.

Be sure to include a leading and trailing blank in your class attribute string check. Otherwise, you could get false matches (without the blanks, task/step would match on notatask/stepaway when it shouldn't).

The class attribute in domains specialization

When you create a domains specialization, the new elements still need a class attribute, but should start with a "+" instead of a "-". This signals any generalization transforms to treat the element differently: A domains-aware generalization transform may have different logic for handling domains than for handling topic specializations.

Domain specializations should be derived either from topic (the root topic type), or from another domain specialization. Do not create a domain by specializing an already specialized topic type: This can result in unpredictable generalization behavior, and is not currently supported by the architecture.

Notices

The information provided in this document has not been submitted to any formal IBM test and is distributed "AS IS," without warranty of any kind, either express or implied. The use of this information or the implementation of any of these techniques described in this document is the reader's responsibility and depends on the reader's ability to evaluate and integrate them into their operating environment. Readers attempting to adapt these techniques to their own environments do so at their own risk.

Â© Copyright International Business Machines Corp., 2002. All rights reserved.
Resources

- IBM donated DITA to the OASIS standards organization in March of 2004, where it is now managed by the OASIS DITA Technical Committee (http://www.oasis-open.org/committees/dita/). In April of 2005, OASIS approved Version 1.0 of the DITA specification, which consists of the following documents:
  - A consolidated .zip file with all specifications, DTDs, and Schemas is publicly available in the documents section of the OASIS DITA Technical Committee site: http://www.oasis-open.org/committees/download.php/12091/cd2.zip
- A reference implementation toolkit for both the developerWorks and OASIS 1.0 versions of the DITA DTDs/Schemas is available at the DITA Open Toolkit project site on SourceForge: http://dita-ot.sourceforge.net. The DITA Open Toolkit supercedes all previous versions published on developerWorks, the last version of which was commonly called "dita132".
- Find out how to join the discussion in the DITA forum, moderated by Don Day and Michael Priestley.
- Go directly to the DITA forum.
- Download the latest DITA DTDs, stylesheets, and sample documents.
- Need background on XSLT? Read Michael Kay's technical introduction, "What kind of language is XSLT?" (developerWorks, April 2005).
About the author

Michael Priestley

Michael Priestley is an information developer for the IBM Toronto Software Development Laboratory. He has written numerous papers on subjects such as hypertext navigation, singlesourcing, and interfaces to dynamic documents. He is currently working on XML and XSL for help and documentation management. You can reach Michael at mpriestl@ca.ibm.com.

© Copyright IBM Corporation 2001, 2005
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)