Specializing domains in DITA

Feature provides for great flexibility in extending and reusing information types

Erik Hennum

September 28, 2005
(First published May 01, 2002)

In current approaches, DTDs are static. As a result, DTD designers try to cover every contingency and, when this effort fails, users have to force their information to fit existing types. The Darwin Information Typing Architecture (DITA) changes this situation by giving information architects and developers the power to extend a base DTD to cover their domains. This article shows you how to leverage the extensible DITA DTD to describe new domains of information.

The Darwin Information Typing Architecture (DITA) is an XML architecture for extensible technical information. A domain extends DITA with a set of elements whose names and content models are unique to an organization or field of knowledge. Architects and authors can combine elements from any number of domains, leading to great flexibility and precision in capturing the semantics and structure of their information. In this overview, you'll learn how to define your own domains.

Introducing domain specialization

In DITA, the topic is the basic unit of processable content. The topic provides the title, metadata, and structure for the content. Some topic types provide very simple content structures. For example, the concept topic has a single concept body for all of the concept content. By contrast, a task topic articulates a structure that distinguishes pieces of the task content, such as the prerequisites, steps, and results.

In most cases, these topic structures contain content elements that are not specific to the topic type. For example, both the concept body and the task prerequisites permit common block elements such as p paragraphs and ul unordered lists.

Domain specialization lets you define new types of content elements independently of topic type. That is, you can derive new phrase or block elements from the existing phrase and block elements. You can use a specialized content element within any topic structure where its base element is allowed. For instance, because a p paragraph can appear within a concept body or task prerequisite, a specialized paragraph could appear there, too.
Here's an analogy from the kitchen. You might think of topics as types of containers for preparing food in different ways, such as a basic frying pan, blender, and baking dish. The content elements are like the ingredients that go into these containers, such as spices, flour, and eggs. The domain resembles a specialty grocer who provides ingredients for a particular cuisine. Your pot might contain chorizo from the carnicería when you're cooking Tex-Mex or risotto when you're cooking Italian. Similarly, your topics can contain elements from the programming domain when you're writing about a programming language or elements from the UI domain when you're writing about a GUI application.

DITA has broad tastes, so you can mix domains as needed. If you're describing how to program GUI applications, your topics can draw on elements from both the programming and UI domains. You can also create new domains for your content. For instance, a new domain could provide elements for describing hardware devices. You can also reuse new domains created by others, expanding the variety of what you can cook up.

In a more formal definition, topic specialization starts with the containing element and works from the top down. Domain specialization, on the other hand, starts with the contained element and works from the bottom up.

**Understanding the base domains**

A DITA domain collects a set of specialized content elements for some purpose. In effect, a domain provides a specialized vocabulary. With the base DITA package, you receive the following domains:

<table>
<thead>
<tr>
<th>Domain</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>highlight</td>
<td>To highlight text with styles such as bold, italic, and monospace</td>
</tr>
<tr>
<td>programming</td>
<td>To define the syntax and give examples of programming languages</td>
</tr>
<tr>
<td>software</td>
<td>To describe the operation of a software program</td>
</tr>
<tr>
<td>UI</td>
<td>To describe the user interface of a software program</td>
</tr>
</tbody>
</table>

In most domains, a specialized element adds semantics to the base element. For example, the apiname element of the programming domain extends the basic keyword element with the semantic of a name within an API.

The highlight domain is a special case. The elements in this domain provide styled presentation instead of semantic or structural markup. The highlight styles give authors a practical way to mark up phrases for which a semantic has not been defined.
Providing such highlight styles through a domain resolves a long-standing dispute for publication DTDs. Purists can omit the highlight domain to enforce documents that should be strictly semantic. Pragmatists can include the highlight domain to provide expressive flexibility for real-world authoring. A semipragmatist could even include the highlight domain in conceptual documents to support expressive authoring, but omit the highlight domain from reference documents to enforce strict semantic tagging.

More generally, you can define documents with any combination of domains and topics. As shown in Generalizing a domain, the resulting documents can still be exchanged.

## Combining an existing topic and domain

The DITA package provides a DTD for each topic type and an omnibus DTD (`ditabase.dtd`) that defines all of the topic types. Each of these DTDs includes all of the predefined DITA domains. Thus, topics written against one of the supplied DTDs can use all of the predefined domain specializations.

Behind the scenes, a DITA DTD is just a shell. Elements are actually defined in other modules, which are included in the DTD. Through these modules, DITA provides you with the building blocks to create new combinations of topic types and domains.

When you add a domain to your DITA installation, the new domain provides you with additional modules. You can use the additional modules to incorporate the domain into the existing DTDs or to create new DTDs.

In particular, each domain is implemented with two files:

- A file that declares the *entities* for the domain. This file has the `.ent` extension.
- A file that declares the *elements* for the domain. This file has the `.mod` extension.

As an example, suppose that you’re authoring the reference topics for a programming language. You’re a purist about presentation, so you want to exclude the highlight domain. You also have no need for the software or UI domains in this reference. You could address this scenario by defining a new shell DTD that combines the reference topic with the programming domain, excluding the other domains.

A shell DTD has a consistent design pattern with a few well-defined sections. The instructions in these sections perform the following actions:

1. Declare the entities for the domains. In the scenario, this section would include the programming domain entities:

```xml
<!ENTITY % pr-d-dec PUBLIC "-//IBM//ENTITIES DITA Programming Domain//EN" "programming-domain.ent">
%pr-d-dec;
```

2. Redefine the entities for the base content elements to add the specialized content elements from the domains.
This section is crucial for domain specialization. Here, the design pattern makes use of two kinds of entities. Each base content element has an element entity to identify itself and its specializations. Each domain provides a separate domain specialization entity to list the specializations that it provides for a base element. By combining the two kinds of entities, the shell DTD allows the specialized content elements to be used in the same contexts as the base element.

In the scenario, the pre element entity identifies the pre element (which, as in HTML, contains preformatted text) and its specializations. The programming domain provides the pr-d-pre domain specialization entity to list the specializations for the pre base element. The same pattern is used for the other base elements specialized by the programming domain:

```
ENTITY % pre "pre | %pr-d-pre;">
ENTITY % keyword "keyword | %pr-d-keyword;">
ENTITY % ph "ph | %pr-d-ph;">
ENTITY % fig "fig | %pr-d-fig;">
ENTITY % dl "dl | %pr-d-dl;">
```

To learn which content elements are specialized by a domain, you can look at the entity declaration file for the domain.

3. Define the domains attribute of the topic elements to declare the domains represented in the document.

Like the class attribute, the domains attribute identifies dependencies. While the class attribute identifies base elements, the domains attribute identifies the domains available within a topic. Each domain provides a domain identification entity to identify itself in the domains attribute.

In the scenario, the only topic is the reference topic. The only domain is the programming domain, which is identified by the pr-d-att domain identification entity:

```
ATTLIST reference domains CDATA "&pr-d-att;">
```

4. Redefine the infotypes entity to specify the topic types that can be nested within a topic.

In the scenario, this section declares the reference topic:

```
ENTITY % info-types "reference">
```

5. Define the elements for the topic type, including the base topics.

In the scenario, this section includes the base topic and reference topic modules:

```
ENTITY % topic-type PUBLIC
  "-//IBM//ELEMENTS DITA Topic//EN"
  "topic.mod">
%topic-type;
ENTITY % reference-typemod PUBLIC
  "-//IBM//ELEMENTS DITA Reference//EN"
  "reference.mod">
%reference-typemod;
```

6. Define the elements for the domains.

In the scenario, this section includes the programming domain definition module:

```
ENTITY % pr-d-def PUBLIC
  "-//IBM//ELEMENTS DITA Programming Domain//EN"
  "programming-domain.mod">
%pr-d-def;
```

Often, it is easiest to work by copying an existing DTD and adding or removing topics or domains. In the scenario, you can start with reference.dtd and remove the highlight, software, and UI domains as indicated with the bold, highlighted text below.
Creating a domain specialization

For some documents, you might need new types of content elements. In a common scenario, you need to mark up phrases that have special semantics. You can handle such requirements by creating new specializations of existing content elements and providing a domain to reuse the new content elements within topic structures.

As an example, suppose that you’re writing the documentation for a class library. You intend to write processes that will index the documentation by class, field, and method. To support this processing, you need to mark up the names of classes, fields, and methods within the topic content, as in the following sample:

```
<p>The `<classname>String</classname>` class provides the `<fieldname>length</fieldname>` field and the `<methodname>concatenate()</methodname>` method.
</p>
```

You must define new content elements for these names. Because the names are special types of names within an API, you can specialize the new elements from the `apiname` element provided by the programming domain.

The design pattern for a domain requires an abbreviation to represent the domain. A sensible abbreviation for the class library domain might be `cl`. The identifier for a domain consists of the abbreviation followed by `-d` (for domain).

As noted in [Combining an existing topic and domain](#), the domain requires an entity declaration file and an element definition file.

Writing the entity declaration file

The entity declaration file has sections that perform the following actions:

1. **Define the domain specialization entities.**
   A domain specialization entity lists the specialized elements provided by the domain for a base element. For clarity, the entity name is composed of the domain identifier and the base element name. The domain provides domain specialization entities for ancestor elements as well as base elements.
   In the scenario, the domain defines a domain specialization entity for the `apiname` base element as well as the `keyword` ancestor element (which is the base element for `apiname`):
   ```
   <!ENTITY % cl-d-apiname "classname | fieldname | methodname">
   <!ENTITY % cl-d-keyword "classname | fieldname | methodname">
   ```

2. **Define the domain identification entity.**
   The domain identification entity lists the topic type as well as the domain and other domains for which the current domain has dependencies. Each domain is identified by its domain identifier. The list is enclosed in parentheses. For clarity, the entity name is composed of the domain identifier and `-att`.
   In the scenario, the class library domain has a dependency on the programming domain, which provides the `apiname` element:
Writing the element definition file

The element definition file has sections that perform the following actions:

1. **Define the content element entities for the elements introduced by the domain.**
   These entities permit other domains to specialize from the elements of the current domain.
   In the scenario, the class library domain follows this practice so that additional domains can be added in the future. The domain defines entities for the three new elements:

   ```xml
   <!ENTITY % classname  "classname">
   <!ENTITY % fieldname  "fieldname">
   <!ENTITY % methodname "methodname">
   ```

2. **Define the elements.**
   The specialized content model must be consistent with the content model for the base element. That is, any possible contents of the specialized element must be generalizable to valid contents for the base element. Within that limitation, considerable variation is possible. Specialized elements can be substituted for elements in the base content model. Optional elements can be omitted or required. An element with multiple occurrences can be replaced with a list of specializations of that element, and so on.
   The specialized content model should always identify elements through the element entity rather than directly by name. This practice lets other domains merge their specializations into the current domain.
   In the scenario, the elements have simple character content:

   ```xml
   <!ELEMENT classname        (#PCDATA)>
   <!ELEMENT fieldname        (#PCDATA)>
   <!ELEMENT methodname       (#PCDATA)>
   ```

3. **Define the specialization hierarchy for the element with class attribute.**
   For a domain element, the value of the attribute must start with a plus sign. Elements provided by domains should be qualified by the domain identifier.
   In the scenario, specialization hierarchies include the keyword ancestor element provided by the base topic and the apiname element provided by the programming domain:

   ```xml
   <!ATTLIST classname      class CDATA "+ topic/keyword pr-d/apiname">
   <!ATTLIST fieldname      class CDATA "+ topic/keyword pr-d/apiname">
   <!ATTLIST methodname     class CDATA "+ topic/keyword pr-d/apiname">
   ```

The complete element definition file would look as follows:
Writing the shell DTD

After creating the domain files, you can write shell DTDs to combine the domain with topics and other domains. The shell DTD must include all domain dependencies.

In the scenario, the shell DTD combines the class library domain with the concept, reference, and task topics and the programming domain. The portions specific to the class library domain are highlighted below in bold:

```xml
<!ENTITY % pr-d-dec PUBLIC
   "-//IBM//ENTITIES DITA Programming Domain//EN"
   "programming-domain.ent">
%pr-d-dec;
<!ENTITY % cl-d-dec SYSTEM "classlib-domain.ent">
%cl-d-dec;

<!--vocabulary declarations-->
<!ENTITY % pre     "pre     | %pr-d-pre;">
<!ENTITY % keyword "keyword | %pr-d-keyword; | %cl-d-apiname;">
<!ENTITY % ph      "ph      | %pr-d-ph;">
<!ENTITY % fig     "fig     | %pr-d-fig;">
<!ENTITY % dl      "dl      | %pr-d-dl;">
<!ENTITY % apiname "apiname | %cl-d-apiname;">

<!--vocabulary substitution-->  
<!ATTLIST concept    domains CDATA "&pr-d-att; &cl-d-att;">
<!ATTLIST reference  domains CDATA "&pr-d-att; &cl-d-att;">
<!ATTLIST task       domains CDATA "&pr-d-att; &cl-d-att;">

<!--Redefine the infotype entity to exclude other topic types-->
<!ENTITY % info-types "concept | reference | task">

<!--Embed topic to get generic elements -->
<!ENTITY % topic-type PUBLIC
   "-//IBM//ELEMENTS DITA Topic//EN" "topic.mod">
%topic-type;

<!--Embed topic types to get specific topic structures-->
<!ENTITY % concept-typemod PUBLIC
   "-//IBM//ELEMENTS DITA Concept//EN"
   "concept.mod">
%concept-typemod;
<!ENTITY % reference-typemod PUBLIC
   "-//IBM//ELEMENTS DITA Reference//EN"
   "reference.mod">
%reference-typemod;

<!ENTITY % task-typemod PUBLIC
   "-//IBM//ELEMENTS DITA Task//EN"
   "task.mod">
%task-typemod;
```
Notice that the class library phrases are added to the element entity for keyword as well as for apiname. This addition makes the class library phrases available within topic structures that allow keywords and not just in topic structures that explicitly allow API names. In fact, the structures of the reference topic specify only keywords, but it’s good practice to add the domain specialization entities to all ancestor elements.

Considerations for domain specialization

When you define new types of topics or domain elements, remember that the hierarchies for topic specialization and domain specialization must be distinct. A specialized topic cannot use a domain element in a content model. Similarly, a domain element can specialize only from an element in the base topic or in another domain. That is, a topic and domain cannot have dependencies. To combine topics and domains, use a shell DTD.

When specializing elements with internal structure -- including the ul, ol, and dl lists, as well as table and simpletable -- you should specialize the entire content element. Creating special types of pieces of the internal structure independently of the whole content structure usually doesn’t make much sense. For example, you usually want to create a special type of list instead of a special type of li list item for ordinary ul and ol lists.

You should never specialize from the elements of the highlight domain. These style elements do not have a specific semantic. Although the formatting of the highlight styles might seem convenient, you might find you need to change the formatting later.

As noted previously, you should use element entities instead of literal element names in content models. The element entities are necessary to permit domain specialization.

The content model should allow for the possibility that the element entity might expand to a list. When applying a modifier to the element entity, you should enclose the element entity in parentheses. Otherwise, the modifier will apply only to the last element if the entity expands to a list. Similar issues affect an element entity in a sequence:

..., ( %classname; ), ...
..., ( %classname; )? ...
..., ( %classname; )* ...
..., ( %classname; )+ ...
..., | %classname; | ...

The parentheses aren't needed if the element entity is already in a list.
Generalizing a domain

As with topics, a specialized content element can be generalized to one of its ancestor elements. In the previous scenario, a `classname` can generalize to `apiname` or even `keyword`. As a result, documents using different domains but the same topics can be exchanged or merged without having to generalize the topics.

To return to the highlight style controversy mentioned in Understanding the base domains, a pragmatic document authored with highlight domain will contain phrases like the following:

```plaintext
... the <b>important</b> point is ...
```

When the document is generalized to the same topic but without the highlight domain, the pragmatic `b` element becomes a purist `ph` element, indicating that the phrase is special without introducing presentation:

```plaintext
... the <ph class="+ topic/ph hi-d/b">important</ph> point is ...
```

In the previous scenario, the class library authors could send their topics to another DITA shop without the class library domain. The recipients would generalize the class library topics, converting the `classname` elements to `apiname` base elements. After generalization, the recipients could edit and process the class, field, and method names in the same way as any other API names. That is, the situation would be the same as if the senders had decided not to distinguish class, field, and method names and, instead, had marked up these names as generic API names.

As an alternative, the recipients could decide to add the class library domain to their definitions. In this approach, the senders would provide not only their topics but also the entity declaration and element definition files for the domain. The recipients would add the class library domain to their shell DTD. The recipients could then work with `classname` elements without having to generalize.

The recipients can use additional domains with no impact on interoperability. That is, the shell DTD for the recipients could use more domains than the shell DTD for the senders without creating the need to modify the topics.

**Note:** When defining specializations, you should avoid introducing a dependency on special processing that lacks a graceful fallback to the processing for the base element. In the scenario, special processing for the `classname` element might generate a literal "class" label in the output to save some typing and produce consistent labels. After automated generalization, however, the label would not be supplied by the base processing for the `apiname` element. Thus, the dependency would require a special generalization transform to append the literal "class" label to `classname` elements in the source file.

**Summary**

Through topic specialization and domains, DITA provides the following benefits:

- **Simpler topic design:** The document designer can focus on the structure of the topic without having to foresee every variety of content used within the structure.
• **Simpler topic hierarchies:** The document designer can add new types of content without having to add new types of topics.

• **Extensible content for existing topics:** The document designer can reuse existing types of topics with new types of content.

• **Semantic precision:** Content elements with more specific semantics can be derived from existing elements and used freely within documents.

• **Simpler element lists for authors:** The document designer can select domains to minimize the element set. Authors can learn the elements that are appropriate for the document instead of learning to disregard unneeded elements.

In short, the DITA domain feature provides for great flexibility in extending and reusing information types. The highlight, programming, and UI domains provided with the base DITA release are only the beginning of what can be accomplished.

**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.

© Copyright IBM Corporation 2002, 2005

**Trademarks**
(www.ibm.com/developerworks/ibm/trademarks/)