[vertical list of authors]
[Don R. Day]
[Michael Priestley]
[Dave A Schell]

[Michael Priestley; IBM Corporation; Toronto, Canada; mpriestl@ca.ibm.com; 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.]

[Erik Hennum; IBM Corporation; ehennum@us.ibm.com. Erik Hennum works on the design and implementation of User Assistance for the IBM Storage Systems Group.]

[cover art/text goes here]

Contents

Introduction to the Darwin Information Typing Architecture

Search title: Pathway to the future

This document is a roadmap for the Darwin Information Typing Architecture: what it is and how it applies to technical documentation. It is also a product of the architecture, having been written entirely in XML and produced using the principles described here...

Executive summary

The Darwin Information Typing Architecture (DITA) is an XML-based, end-to-end architecture for authoring, producing, and delivering technical information. This architecture consists of a set of design principles for creating "information-typed" modules at a topic level and for using that content in delivery modes such as online help and product support portals on the Web.

At the heart of DITA (Darwin Information Typing Architecture), representing the generic building block of a topic-oriented information architecture, is an XML document type definition (DTD) called "the topic DTD." The extensible architecture, however, is the defining part of this design for technical information; the topic DTD, or any schema based on it, is just an instantiation of the design principles of the architecture.

Background

This architecture and DTD were designed by a cross-company workgroup representing user assistance teams from across IBM. After an initial investigation in late 1999, the workgroup developed the architecture collaboratively during 2000 through postings to a database and weekly teleconferences. The architecture has been placed on IBM's developerWorks Web site as an alternative XML-based documentation system, designed to exploit XML as its encoding format. With the delivery of these signficant updates contains enhancements for consistency and flexibility, we consider the DITA design to be past its prototype stage.

Information interchange, tools management, and extensibility

IBM, with millions of pages of documentation for its products, has its own very complex SGML DTD, IBMIDDoc, which has supported this documentation since the early 1990s. The workgroup had to consider from the outset, "Why not just convert IBMIDDoc or use an existing XML DTD such as DocBook, or TEI, or XHTML?" The answer requires some reflection about the nature of technical information.

First, both SGML and XML are recognized as meta languages that allow communities of data owners to describe their information assets in ways that reflect how they develop, store, and process that information. Because knowledge representation is so strongly related to corporate cultures and community jargon, most attempts to define a ***universal DTD*** have ended up either unused or unfinished. The ***ideal for information interchange*** is to share the semantics and the transformational rules for this information with other data-owning communities.

Second, most companies rely on many delivery systems, or process their information in ways that differ widely from company to company. Therefore any attempt at a ***universal tool set*** also proves futile. The ***ideal for tools management*** is to base a processing architecture on standards, to leverage the contributed experience of many others, and to solve common problems in a broad community.

Third, most attempts to formalize a document description vocabulary (DTD or schema) have been done as information modelling exercises to capture the ***current business practices*** of data owners. This approach tends to encode *legacy* practices into the resulting DTDs or vocabularies. The ***ideal for future extensibility*** in DTDs for technical

information (or any information that is continually exploited at the leading edge of technology) is to build the fewest presumptions about the "top-down" processing system into the design of the DTD.

In the beginning, the workgroup tried to understand the role of XML in this leading edge of information technology. As the work progressed, the team became aware that any DTD design effort would have to account for a plurality of vocabularies, a tools-agnostic processing paradigm, and a legacy-free view of information structures. Many current DTDs incorporate ways to deal with some of these issues, but the breadth of the issues lead to more than just a DTD. To support many products, brands, companies, styles, and delivery methods, the entire authoring-to-delivery process had to be considered. What resulted was a range of recommendations that required us to represent our design, not just as a DTD, but as an information architecture.

Main features of the DITA architecture

As the "Architecture" part of DITA's name suggests, DITA has unifying features that serve to organize and integrate information:

- *Topic orientation.* The highest standard structure in DITA is the topic. Any higher structure than a topic is usually part of the processing context for a topic, such as a print-organizing structure or the helpset-like navigation for a set of topics. Also, topics have no internal hierarchical nesting; for internal organization, they rely on sections that define or directly support the topic.
- *Reuse.* A principal goal for DITA has been to reduce the practice of copying content from one place to another as a way of reusing content. Reuse within DITA occurs on two levels:
    - *Topic reuse.* Because of the non-nesting structure of topics, a topic can be reused in any topic-like context. Information designers know that when they reuse a topic in a new information model, the architecture will process it consistently in its new context.
    - *Content reuse.* The SGML method of declaring reusable external entities is available for XML users, but this has several practical limitations in XML. DITA instead leans toward a different SGML reuse technique and provides each element with a conref attribute that can point to any other equivalent element in the same or any other topic. This referencing mechanism starts with a base element, thus assuring that a fail-safe structure is always part of the calling topic (the topic that contains the element with the conref attribute). The new content is always functionally equivalent to the element that it replaces.
- *Specialization.* The class mechanism in CSS indicates a common formatting semantic for any element that has a matching class value. In the same way, any DITA element can be extended into a new element whose identifier gets added to the class attribute through its DTD. Therefore, a new element is always associated to its base, or to any element in its specialization sequence.
    - *Topic specialization.* Applied to topic structures, specialization is a natural way to extend the generic topic into new information types (or infotypes), which in turn can be extended into more specific instantiations of information structures. For example, a recipe, a material safety data sheet, and an encyclopedia article are all potential derivations from a common reference topic.
    - *Domain specialization.* Using the same specialization principle, the element vocabulary within a generic topic (or set of infotyped topics) can be extended by introducing elements that reflect a particular information domain served by those topics. For example, a keyword can be extended as a unit of weight in a recipe, as a part name in a hardware reference or as a variable in a programming reference. A specialized domain, such as programming phrases, can be introduced by substitution anywhere that the root elements are allowed. This makes the entire vocabulary available throughout all the infotyped topics used within a discipline. Also, a domain can be replaced within existing infotyped topics, in effect hiding the jargon of one discipline

from writers dealing with the content of another. Yet both sets of topics can be appropriate for the same user roles of performing tasks or getting reference information.

- *Property-based processing.* The DITA model provides metadata and attributes that can be used to associate or filter the content of DITA topics with applications such as content management systems, search engines, processing filters, and so on.
    - *Extensive metadata to make topics easier to find.* The DITA model for metadata supports the standard categories for the Dublin Core Metadata Initiative. In addition, the DITA metadata enables many different content management approaches to be applied to its content.
    - *Universal properties.* Most elements in the topic DTD contain a set of universal attributes that enable the elements to be used as selectors, filters, content referencing infrastructure, and multi-language support. In addition, some elements, whose attributes can serve a range of specialized roles, have been analyzed to make sure that their enumerated values provide a rich basis for specialization (which usually constrains values and never adds to them).
- *Taking advantage of existing tags and tools.* Rather than being a radical departure from the familiar, DITA builds on well-accepted sets of tags and can be used with standard XML tools.
    - *Leveraging popular language subsets.* The core elements in DITA's topic DTD borrow from HTML and XHTML, using familiar element names like p, ol, ul, dl within an HTML-like topic structure. In fact, DITA topics can be written, like HTML, for rendering directly in a browser. In more ambitious designs, DITA topics can be written, like SGML, to be normalized through processing into a deliverable, say XHTML or a well-formed XML format targeted for a particular browser's ability to handle XML. Also, DITA makes use of the popular OASIS (formerly CALS) table model.
    - *Leveraging popular and well-supported tools.* The XML processing model is widely supported by a number of vendors. The class-based extension mechanism in DITA translates well to the design features of the XSLT and CSS stylesheet languages defined by the World Wide Web Consortium and supported in many transformation tools, editors and browsers. DITA topics can be processed by a spectrum of tools ranging from shareware to custom tailored products, on almost any operating platform.

Topic as the basic architectural unit

The various information architectures for online deliverables all tend to focus on the idea of topics as the main design point for such information. A topic is a unit of information that describes a single task, concept, or reference item. The information category (concept, task, or reference) is its information type (or infotype). A new information type can be introduced by **specialization** from the structures in the base topic DTD. Typed topics are easily managed within content management systems as reusable, stand-alone units of information. For example, selected topics can be gathered, arranged, and processed within a **delivery context** to provide a variety of deliverables. These deliverables might be groups of recently updated topics for review, helpsets for building into a user assistance application, or even chapters or sections in a booklet that are printed from user-selected search results or "shopping lists."

Benefits of the DITA architecture

Through topic granularity and topic type specialization, DITA brings the following benefits of the object-oriented model to information sets:

- *Encapsulation.* The designer of the topic type only needs to address a specific, manageable problem domain. The author only needs to learn the elements that are specific to the topic type. The implementer of the processing for the topic type only needs to process elements that are special.
- *Polymorphism.* Special topic types can be treated as more generic topic types for common processing.
- *Message passing.* The class attribute preserves at all times the derivation hierarchy

of an element. At any time, a topic may be generalized back to any earlier form, and if the class attributes are preserved, these topics may be re-specialized. One use of this capability would be to allow two separate disciplines to merge data at an earlier common part of the specialization hierarchy, after which they can be transformed into one, the other, or a brand new domain and set of infotyped topics.

DITA can be considered object-oriented in that:

- Data and processors are separated from their environment and can be chunked to provide behaviors similar to object-orientation (such as override transforms that modify or redefine earlier behaviors).
- Classification of elements through a sequence of derivations that are progressively more specific, possibly more constrained, and always rigidly tied to a consistent processing or rendering model.
- Inheritance of behaviors, to the extent that new elements either fall through to behaviors for ancestors in their derivation hierarchy, or can be mapped to modified processors that extend previous behaviors.

With discipline and ingenuity, some of the benefits of topic information sets can be provided through a book DTD. In particular, techniques for chunking can generate topics out of a book DTD. In DITA, the converse approach is possible: a book can be assembled from a set of DITA topics. In both cases, however, the adaptation is secondary to the primary purpose of the DTD. That is, if you are primarily authoring books, it makes the most sense to use a DTD that is designed for books. If you are primarily authoring topics, it makes sense to use a DTD that is designed for topics and can scale to large, processable collections of topics.

DITA overview

The Darwin Information Typing Architecture defines a set of relationships between the document parts, processors, and communities of users of the information.

The Darwin Information Typing Architecture has the following layers that relate to specific design points expressed in its core DTD, *topic*.

Figure1. Layers in the Darwin Information Typing Architecture

Delivery contexts		
helpset	aggregate printing	Web site; information portal

Typed topic structures			
topic	concept	task	reference

Specialized vocabularies (domains) across information types			
Typed topic:	concept	task	reference
Included domains:	highlighting software programming user interface		

Common structures	
metadata	OASIS (CALS) table

A typed topic, whether concept, task, or reference, is a stand-alone unit of ready-to-be-published information. Above it are any processing applications that may be driven by a superset DTD; below it are the two types of content models that form the basis of all specialized DTDs within the architecture. We will look at each of these layers in more detail.

DITA delivery contexts

This domain represents the processing layer for topical information. Topics can be processed singly or within a delivery context that relates multiple topics to a defined deliverable. Delivery contexts also include document management systems, authoring units, packages for translation, and more.

delivery contexts		
helpset	aggregate printing	Web site; information portal

DITA typed topic specializations (infotyped topics)

The typed topics represent the fundamental structuring layer for DITA topic-oriented content. The basis of the architecture is the *topic* structure, from which the *concept*, *task*, and *reference* structures are specialized. Extensibility to other typed topics is possible by further specialization.

typed topic structures			
topic	concept	task	reference

The four information types (topic, concept, task, and reference) represent the primary content categories used in the technical documentation community. Moreover, specialized, information types, based on the original four, can be defined as required.

As a notable feature of this architecture, communities can define or extend additional information types that represent their own data. Examples of such content include product support information, programming message descriptions, and GUI definitions. Besides the ability to type topics and define specific content models therein, DITA also provides the ability to extend tag vocabularies that pertain to a domain. Domain specialization takes the place of what had been called "shared structures" in DITA's original design.

DITA vocabulary specialization (domains)

Commonly, when a set of infotyped topics are used within a domain of knowledge, such as computer software or hardware, a common vocabulary is shared across the infotyped topics. However, the same infotyped topic can be used across domains that have different vocabularies and semantics. For example, a hardware reference topic might refer to diagnostic codes while a software reference topic might refer to error message numbers, with neither domain necessarily needing to expose the other domain's unique vocabulary to its own writers.

Using the same technique as specialization for topics, DITA allows the definition of domains of special vocabulary that can be shared among infotyped topics. Domains can even be elided entirely, to produce typed topics that have only the core elements
In the original design of DITA, all of the shared vocabulary had been made global to all information types by being defined in the topic DTD, which had two undesirable effects:

- new vocabulary could not be added without increasing the size of the core DTD
- certain domain-specific vocabulary could not be prohibited for DTDs specialized for a different domain.

. The vocabulary of a domain can take the form of phrases, special paragraphs, and lists--basically anything allowed within a section, the smallest organizing part of a topic.

specialized vocabularies (domains) across information types			
Typed topic:	concept	task	reference
Included domains:	highlighting software programming user interface		

The basic domains defined as examples for DITA include:

Domain	Elements
highlighting	b, u, i, tt, sup, sub
software	msgph, msgblock, msgnum, cmdname, varname, filepath, userinput, systemoutput
programming	codeph, codeblock, option, var, parmname, synph, oper, delim, sep, apiname, parml, plentry, pt, pd, syntaxdiagram, synblk, groupseq, groupchoice, groupcomp, fragment, fragref, synnote, synnoteref, repsep, kwd
user interface	uicontrol, wintitle, menucascade, shortcut

By following the rules for specializing a new domain of content, you can extend, replace, or remove these domains. Moreover, content specialization enables you to name and extend *any* content element in the scope of DITA infotyped topics for a more semantically significant role in a new domain.

To enable specialized vocabulary, you declare a parameter entity equivalent for every element used in a DTD (such as topic or one of its specializations), and then use the parameter entities instead of literal element tokens within the content models of that DTD. Later, after entity substitution, because an element's parameter entity is redefined to include both the original element and the domain elements derived from that element, anywhere the original element is allowed, the other derived domain elements are also allowed. In effect, a domain-agnostic topic can be easily extended for different domains by simply changing the scope of entity set inclusions in a front-end DTD "shell" that formalizes the vocabulary extensions within that typed topic or family of typed topics

DITA common structures

One of the design points of DITA has been to exploit the reuse of common substructures within the world of XML. Accordingly, the topic DTD incorporates the OASIS table model (known originally as the CALS table model). It also has a defined set of metadata that might be shared directly with the metadata models of quite different DTDs or schemas.

common structures	
metadata	OASIS (CALS) table

The metadata structure defines document control information for individual topics, higher-level processing DTDs, or HTML documents that are associated to the metadata as side files or as database records.

The table structure provides presentational semantics for body-level content. The OASIS/CALS table display model is supported in many popular XML editors.

Elements designed for specialization

DITA provides a rich base for specialization because of the general design of elements used in its archetype-like topic DTD.

For example, a section in the base topic DTD can contain both text and element data. However, a section can be specialized to eliminate PCDATA, yielding an element-only content model similar to the body level of most DTDs. Specialized another way, a section can eliminate most block-like elements and thus be characterized as a description for definitions, field labels, parts, and so forth.

In DITA, an effort has been made to select element names that are popular or that are common with HTML. Some semantic names have been borrowed from industry DTDs that support large SGML libraries, such as IBMIDDoc and DocBook.

The attribute lists within the topic DTD reflect this design philosophy. For example, one of the "universal attributes" (they appear on most elements) is `importance`, which defines values for weightings or appraisals that are often used as properties in specialized elements. This attribute shows up in several elements of the task topic specialization with only two allowed values out of the original set, "optional' and "required." In other domains, the elements are more appropriately ranked as "high" or "low," again values that are provided at the topic level.

The values of specialization

A company that has specific information needs can define specialized topic types. For example, a product group might identify three main types of reference topic: messages, utilities, and APIs. By creating a specialized topic type for each type of content, the product architect can be assured that each type of topic has the appropriate content. In addition, the specialized topics make XML-aware search more useful because users can make fine-grained distinctions. For example, a user could search for "xyz" only in messages or only in APIs, as well as search for "xyz" across reference topics in general.

There are rules for how to specialize safely: each new information type must map to an existing one and must be more restrictive in the content that it allows. With such specialization, new information types can use generic processing streams for translation, print, and Web publishing. Although a product group can override or extend these processes, they get the full range of existing processes by default without any extra work or maintenance.

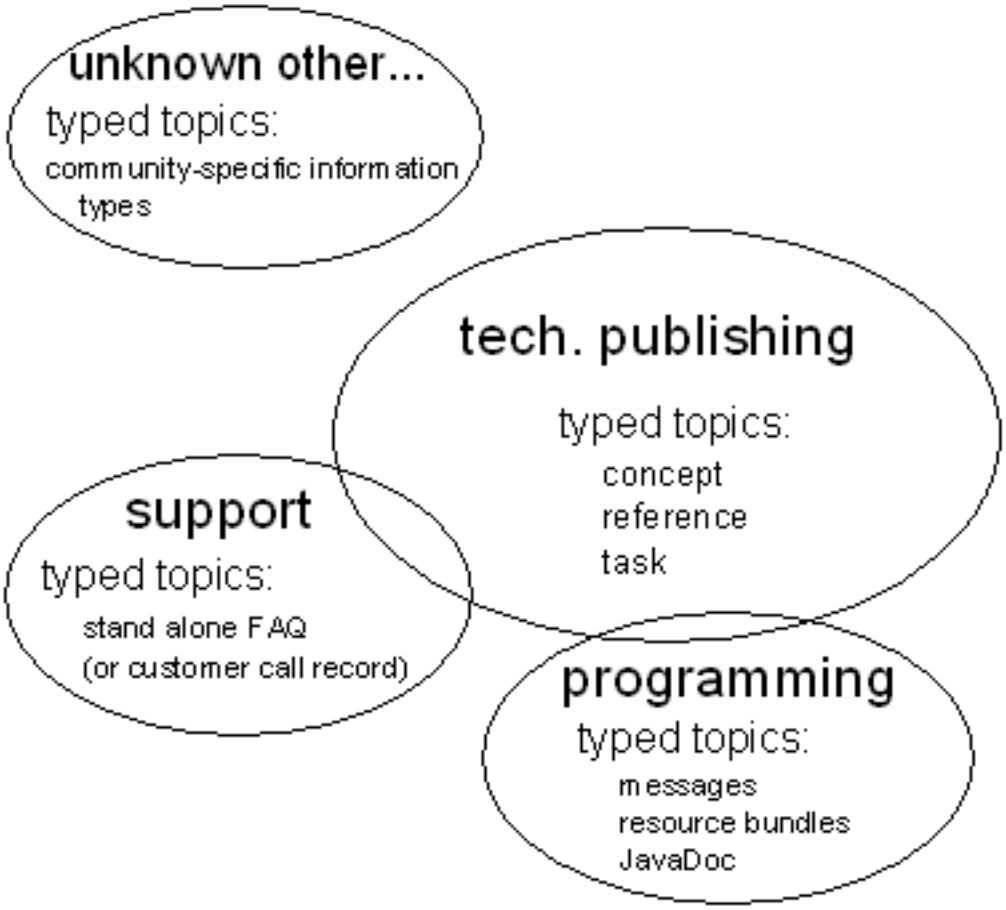
A corporation can have a series of DTDs that represent a consistent set of information descriptions, each of which emphasizes the value of specialization for those new information types.

Role of content communities in the Darwin Information Typing Architecture

The technical documentation community that designed this architecture defined the basic architecture and shared resources. The content owned by specified communities (within or outside of the defining community) can reuse processors, styles, and other features already defined. But, those communities are responsible for their unique business processes based on the data that they manage. They can manage data by creating a further specialization from one of the base types.

The following figure represents how communities, as "content owners at the topic level," can specialize their content based on the core architecture.

Figure1. Relationship of specialized communities to the base architecture



In this figure, the overlap represents the common architecture and tools shared between content-owning communities that use this information architecture. New communities that define typed documents according to the architecture can then use the same tools at the outset, and refine their content-specific tools as needed.

Notices

> 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.

Specializing topic types in DITA

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.

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.

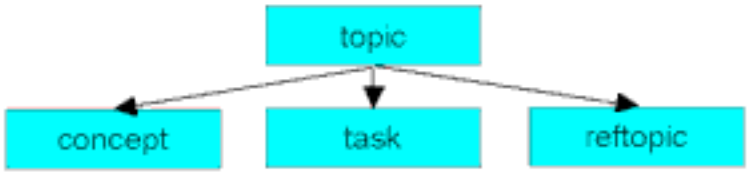
Specialization lets MyHTML be defined as an extension of the HTML DTD, declaring new element types only as necessary and referencing HTML's DTD for shared elements. Wherever MyHTML declares a new element, it includes a mapping back to an existing

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.

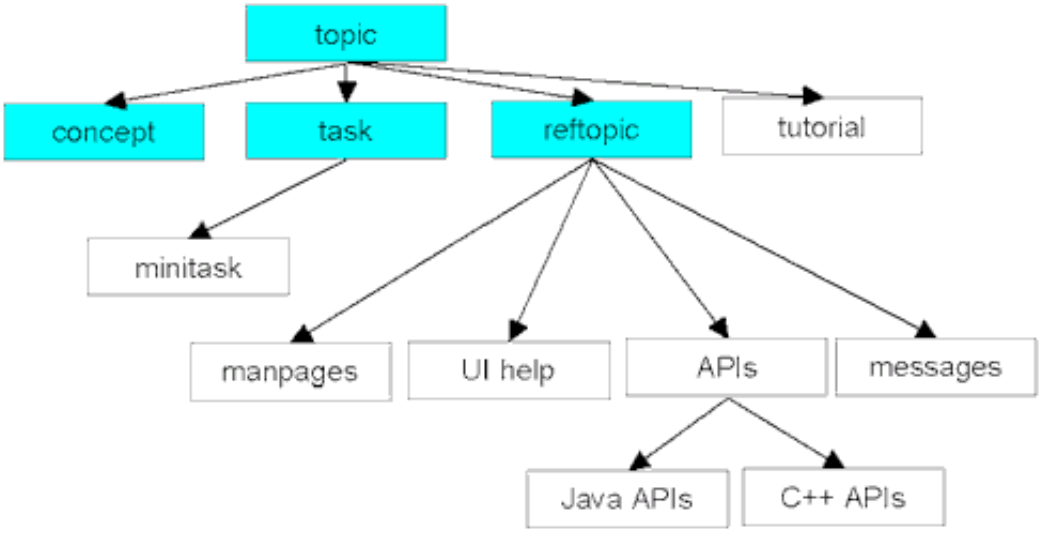
Specializing information types

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:



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 of the separation of information types 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:

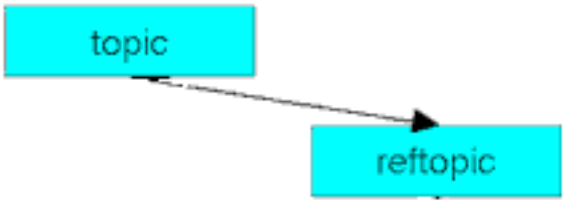


Table 1 expresses the relationship between the general elements in topic and the specific elements in reference. Within the table, the columns, rows, and cells indicate information types, element mappings, and elements. Table 2 explains the relationships in detail to help you interpret Table 1.

Table1. Relationships between topic and a specialization based on it

Topic	Reference
(topic.mod)	(reference.mod)
topic	reference
title	'
body	refbody
simpletable	properties
'	
section	refsyn
'	

Structure

Associations

Columns

The **Topic** column shows basic `topic` structure, which comprises a title and body with optional sections, as declared in a DTD module called `topic.mod` . The **Reference** column shows a more specialized structure, with `reference` replacing `topic`, `refbody` replacing `body`, and `refsyn` replacing `section`; these new elements are declared in a DTD module called `reference.mod` .

Rows

Each row represents a mapping between the elements in that row. The elements in the **Reference** column specialize the elements in the **Topic** column. Each general element also serves as a category for more specialized elements in the same row. For example, `reference`'s `refsyn` is a kind of `section`.

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 `reference` `title` 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 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.

Listing 1. reference.mod

```
<!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 redeclare any of the content that it shares with `topic`.

Adding specialization attributes

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

Listing 2. reference.mod (part 2)

```
<!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, we'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 we'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), we can assemble an authoring DTD.

Listing 3. reference.dtd

```
<!--Redefine the infotype entity to exclude other topic types-->
<!ENTITY % info-types "reftopic">
```

```
<!--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 let's create a more specialized information type: API descriptions, which are a kind of (and therefore specialization of) reference topic:

Figure1. A more specialized information type, API description

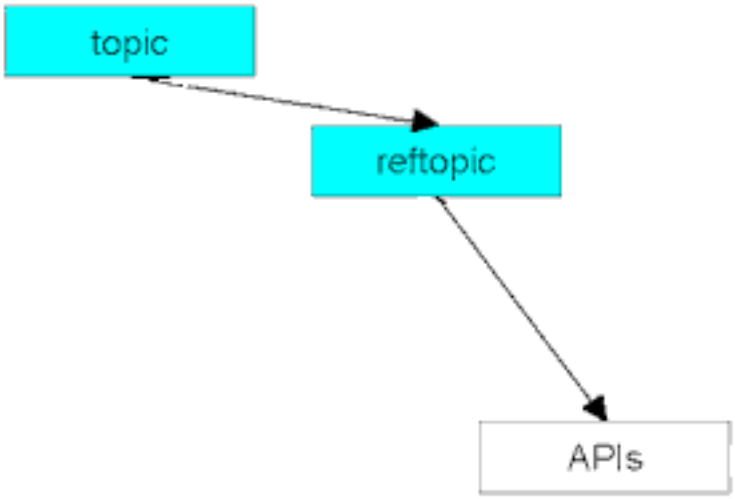


Table 3 shows part of the specialization for an information type called `APIdesc`, for API description. As before, each column represents an information type, with specialization occurring from left to right. That is, each information type is a specialization of its neighbor to the left. Each row represents 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 example, `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.

Table1. Summary of APIdesc specialization

Topic	Reference	APIdesc
(topic.mod)	(reference.mod)	(APIdesc.mod)
topic	reference	APIdesc
title	'	APIname
body	refbody	APIbody
simpletable	properties	parameters
'	'	
section	refsyn	'

Topic	Reference	APIdesc
'	'	
usage		

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, we can also get rid of the title element in usage's content model, making use of the predefined section.notitle.cnt entity.

APIdesc.mod

```
<!ELEMENT APIdesc (APIname, (%prolog;)?, APIbody,(%info-types;)* )>
<!ELEMENT APIname (%title.cnt;)*>
<!ELEMENT APIbody (refsyn,usage,parameters,(%section;)*)>
<!ELEMENT usage (%section.notitle.cnt;)* >
<!ATTLIST usage spectitle CDATA #FIXED "Usage">
<!ELEMENT parameters ((%sthead;)?, (%strow;)+)>
```

Adding specialization attributes

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

APIdesc.mod (part 2)

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

Note that `APIname` 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 we'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), we can assemble an authoring DTD.

APIdesc.dtd

```
<!--Redefine the infotype entity to exclude other topic types-->
<!ENTITY % info-types "APIdesc">
<!--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">
%reftopic-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 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 tranforms 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.

The specialization-aware interface

```
<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`:

A specialized transformation for APIdesc

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

Both the preexisting `reference properties` template rule and the new `parameters` template rule match when they encounter a `parameters` element (because the `parameters` element is a specialized type of `reference properties` 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 per 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 provices:

- 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 11:

A sample topic from APIdesc

```
<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):

The same sample topic from APIdesc, including the class attributes

```
<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 "
  spectitle="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, it should look something like Listing 13:

A transformed topic from APIdesc

```
<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/refsyn ">AnAPI(parm1,
  parm2)</section>
  <section class="- topic/section reference/section APIdesc/usage "
  spectitle="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.
- Or 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 vs. 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 would make 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, we 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 `APIbody`)
- 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).

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 A:

Table1. Summary of specialization rules

Content type	Allowed specialization	Example (Special specializing General)

Required	Rename only	`<!ELEMENT General(a)>`
		
		`<!ELEMENT Special(a.1)>`

Optional (?)	Rename, make required, or delete	`<!ELEMENT General(a?)>`
		
		`<!ELEMENT Special(a.1?)>`
		`<!ELEMENT Special(a.1)>`
		`<!ELEMENT Special EMPTY>`

One or more (+)	Rename, make required, split into a required element plus others, split into one or more elements plus others.	`<!ELEMENT General(a+)>`
		
		`<!ELEMENT Special(a.1+)>`
		`<!ELEMENT Special(a.1)>`
		`<!ELEMENT Special(a.1,a.2,a.3+,a.4*)>`
		`<!ELEMENT Special(a.1+,a.2,a.3*)>`

Zero or more (*)	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	`<!ELEMENT General(a*)>`
		
		`<!ELEMENT Special(a.1*)>`
		`<!ELEMENT Special(a.1)>`
		`<!ELEMENT Special(a.1?)>`
		`<!ELEMENT Special(a.1,a.2,a.3+,a.4*)>`
		`<!ELEMENT Special(a.1?,a.2,a.3+,a.4*)>`
		`<!ELEMENT Special(a.1+,a.2,a.3*)>`
		`<!ELEMENT Special(a.1*,a.2?,a.3*)>`
		`<!ELEMENT Special EMPTY>`

Either-or	Rename, or choose one	`<!ELEMENT General (a|b)>`
		
		`<!ELEMENT Special (a.1|b.1)>`
		`<!ELEMENT Special (a.1)>`

Content type	Allowed specialization	Example (Special specializing General)

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.

The content model for the general element General

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

When you specialize `General` to create `Special`, its content model must be the same 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:

A valid change to the model Special, making b mandatory

```
<!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`.

A valid change to the model Special, making c mandatory and disallowing d

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

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

A valid change to the model Special, making three specializations of d mandatory

```
<!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:

```
<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:

```
<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 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:

```
<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.
<xsl:template match="*[contains(@class,' task/step ')]">
```

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

Specializing domains in DITA

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. DITA changes this situation by giving information architects and developers the power to extend a base DTD to cover their domains.

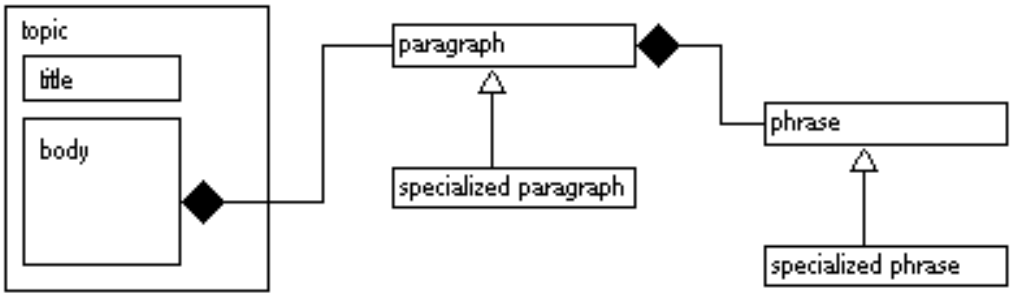
The Darwin Information Typing Architecture (DITA) is an XML architecture for extensible technical information. A domain extends DITA with a set of elements whose names and content models are unique to an organization or field of knowledge. Architects and authors can combine elements from any number of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

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 we'll see 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, let's say we're authoring the reference topics for a programming language. We're purists about presentation, so we want to exclude the highlight domain. We also have no need for the software or UI domains in this reference. We 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:

   ```
   <!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. Where 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 would declare the `reference` topic:

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

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

   In the scenario, this section would include 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 would include the programming domain definition module:

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

Often, it would be easiest to work by copying an existing DTD and adding or removing topics or domains. In the scenario, it would be easiest to start with `reference.dtd` and remove the highlight, software, and UI domains as shown with the underlined text below.

```
<!--vocabulary declarations-->
<!ENTITY % ui-d-dec PUBLIC "-//IBM//ENTITIES DITA User Interface
Domain//EN" "ui-domain.ent">
  %ui-d-dec;
<!ENTITY % hi-d-dec PUBLIC "-//IBM//ENTITIES DITA Highlight Domain//EN"
"highlight-domain.ent">
  %hi-d-dec;
<!ENTITY % pr-d-dec PUBLIC "-//IBM//ENTITIES DITA Programming Domain//EN"
"programming-domain.ent">
  %pr-d-dec;
<!ENTITY % sw-d-dec PUBLIC "-//IBM//ENTITIES DITA Software Domain//EN"
"software-domain.ent">
  %sw-d-dec;

<!--vocabulary substitution-->
<!ENTITY % pre     "pre     | %pr-d-pre;     | %sw-d-pre;">
```

```
<!ENTITY % keyword "keyword | %pr-d-keyword; | %sw-d-keyword; |
%ui-d-keyword;">
<!ENTITY % ph      "ph      | %pr-d-ph;      | %sw-d-ph;       | %hi-d-ph;
| %ui-d-ph;">
<!ENTITY % fig     "fig     | %pr-d-fig;">
<!ENTITY % dl      "dl      | %pr-d-dl;">

<!--vocabulary attributes-->
<!ATTLIST reference  domains CDATA "&ui-d-att; &hi-d-att; &pr-d-att;
&sw-d-att;">

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

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

<!--Embed reference to get specific elements -->
<!ENTITY % reference-typemod PUBLIC "-//IBM//ELEMENTS DITA Reference//EN"
"reference.mod">
  %reference-typemod;

<!--vocabulary definitions-->
<!ENTITY % ui-d-def PUBLIC "-//IBM//ELEMENTS DITA User Interface
Domain//EN" "ui-domain.mod">
  %ui-d-def;
<!ENTITY % hi-d-def PUBLIC "-//IBM//ELEMENTS DITA Highlight Domain//EN"
"highlight-domain.mod">
  %hi-d-def;
<!ENTITY % pr-d-def PUBLIC "-//IBM//ELEMENTS DITA Programming Domain//EN"
"programming-domain.mod">
  %pr-d-def;
<!ENTITY % sw-d-def PUBLIC "-//IBM//ELEMENTS DITA Software Domain//EN"
"software-domain.mod">
  %sw-d-def;
```

Creating a domain specialization

For some documents, you may 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, let's say we're writing the documentation for a class library. We intend to write processes that will index the documentation by class, field, and method. To support this processing, we 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>
```

We must define new content elements for these names. Because the names are special types of names within an API, we 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:

   ```
   <!ENTITY cl-d-att "(topic pr-d cl-d)">
   ```

The complete entity declaration file would look as follows:

```
<!ENTITY % cl-d-apiname "classname | fieldname | methodname">
<!ENTITY % cl-d-keyword "classname | fieldname | methodname">

<!ENTITY cl-d-att "(topic pr-d cl-d)">
```

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:

   ```
   <!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:

```
<!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:

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

The complete element definition file would look as follows:

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

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

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

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:

```
<!--vocabulary declarations-->
<!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 substitution-->
<!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 attributes-->
<!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;

<!--vocabulary definitions-->
<!ENTITY % pr-d-def PUBLIC "-//IBM//ELEMENTS DITA Programming Domain//EN"
"programming-domain.mod">
  %pr-d-def;
<!ENTITY % cl-d-def SYSTEM "classlib-domain.mod"> %cl-d-def;
```

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:

```
... 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:

```
... 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 any 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

How to define a formal information architecture with DITA map domains

The benefits of formal information typing are well known for the content of topics, but collections of topics also benefit from formal organizing structure. Such formal structures guide authors while they assemble collections of topics and ensure consistent large-scale patterns of information for the user. Using DITA map domains, a designer can define a formal information architecture that can be reused in many deliverables.

This article explains the design technique for creating DITA map domains. As an example, the article walks through the definition for assembling a set of topics as a how-to. Such a how-to could be one reusable design component within an information architecture.

Formal information architecture

Information architecture can be summarized as the design discipline that organizes information and its navigation so an audience can acquire knowledge easily and efficiently. For instance, the information architecture of a web site often provides a hierarchy of web pages for drilling down from general to detailed information, different

types of web pages for different purposes such as news and documentation, and so on.

An information architecture is subliminal when it works well. The lack of information architecture is glaring when it works poorly. The user cannot find information or, even worse, cannot recognize or assimilate information when by chance it is encountered. You probably have experience with websites that are poorly organized or uneven in their approach, so that conventions learned in part of the website have no application elsewhere. Extracting knowledge from such information resources is exhausting, and you quickly abandon the effort and seek the information elsewhere.

Currently, information architects work by defining the architecture through guidelines and instructions to the writer. A better approach is to formalize the architecture through an XML design that is validated by the XML editor or parser. This formal approach has the following benefits:

- Authors receive guidance from the markup while working.
- Information with the same purpose is consistent across deliverables.
- Information for a purpose is complete.
- Processing can rely on the structure of the information and operate on the declared semantics of the information.

The following drawings illustrate the gain in clarity and consistency by applying a design to produce a formal information architecture:

In short, the formal design acts as a kind of blueprint to be fullfilled by the writer.

Specializing topics and maps

DITA supports the definition of a formal information architecture through topics and map types. The topic type defines the information architecture within topics (the micro level) while the map type defines the information architecture across topics (the macro level).

The base topic and map types are general and flexible so they can accomodate a wide variety of readable information. You specialize these general types to define the restricted types required for your information architecture.

Topic
: The topic type mandates the structure for the content of a topic. For instance, the DITA distribution includes a task type that mandates a list of steps as part of the topic content. This specialized topic type provides guidance to the author and ensures the consistency of all task topics. Processing can rely on this consistency and semantic precision. For instance, the processing for the task type could format the task steps as checkable boxes.

Map
: The map type mandates the structure for a collection of topics. A map can define the navigation hierarchy for a help system or the sequence and nesting of topics in a book. For instance, the DITA distribution includes a bookmap demo that mandates a sequence of preface, chapter, and appendix roles for the top-level topics. This specialized map type ensures that the collection of topics conforms to a basic book structure.

Without formal types, the information architecture is defined only through editorial guidelines. Different authors may interpret or conform to the guidelines in varying degrees, resulting in inconsistency and impredictability. By contrast, the formal types ensure that the design that can be repeated for many deliverables.

The how-to collection

One typical purpose for a collection of topics is explain how to accomplish a specific goal.

A how-to assembles the relevant topics and arranges them in a typical sequence for one way to reach that goal. A standard design pattern for the how-to collection might consist of an introduction topic, some background concepts, some task and example topics, and a summary.

A help system or book might have several how-tos, for instance, on setting up web authentication, reading a database from a web application, and so on. Or, a web provider might publish an ongoing series of how-to articles on technical subjects. Thus, designing a formal how-to pattern would be useful so that all how-tos are consistent regardless of the writer.

Note that formalizing a collection doesn't prevent topic reuse but, instead, guides topic reuse so that appropriate types of topics are used at positions within the collection. For example, in the how-to, concept topics will appear only as background before the tasks rather than in the middle of the how-to.

Map specialization

Among the many capabilities added to maps by DITA 1.3 is specialization through map domains. Instead of packaging specializations of elements for topic content, however, you specialize elements for map content, typically the`topicref`. The specialized `topicref`element lets authors specify semantics or constraints on collections of topics. By packaging the `topicref` specializations as a map domain rather than as a map type, you can reuse the formal collection design in many different map types.

A specialized `topicref`can be used for the following purposes:

- To restrict the references to topics of a specialized type. For instance, a `conceptref` refers only to concept topics (including specialized concepts).
- To assign a topic a topic to a role within a collection. For instance, the topic identified by a `summaryref` could provide the concluding explanation for a collection.
- To restrict the contents of the collection, requiring specific topic types or requiring topics to act in specific roles at specified positions within the collection.

Drawing on all of these capabilities, we can define a formal structure for a how-to collection.

Implementing a map domain

A map domain uses the same DTD design pattern as a topic domain. See specializing domains for the details on the domain design pattern, which aren't repeated here. Instead, this article summarizes the application of the domain DTD design pattern to maps.

1. Create a domain entities file to declare the elements extending the `topicref` element.
2. Create a domain definition module to define the elements including their element entities, content and attribute definitions, and the architectural class attribute.
3. Create a shell DTD that assembles the base map module and the domain entities file and definition module.
4. Create map collections from the shell DTD.

Declaring the map domain entities

The entities file for the how-to domain defines the howto, conceptref, taskref, and exampleref extensions for the topicref element and defines the how-to domain declaration for the domain attributes entity:

```
<!ENTITY % howto-d-topicref "howto">
<!ENTITY howto-d-att "(map howto-d)">
```

Defining the map domain module

The definition module for the how-to domain starts with the element entities so the new elements could, in turn, be extended by subsequent specializations. Of these new elements, only `howto` has been declared in the entities file because the other new elements should only appear in the child list of the `howto` element. (In fact, reference typing elements such as `conceptref` and `taskref` might also be defined in the entities file for reuse in other specialized child lists.)

```
<!ENTITY % howto       "howto">
<!ENTITY % conceptref  "conceptref">
<!ENTITY % taskref     "taskref">
<!ENTITY % exampleref  "exampleref">
<!ENTITY % summaryref  "summaryref">
```

The definition module goes on to define the elements. The definition for the `howto` element restricts the content list for the collection to the metadata for the topic, references to any number of concept topics, references to task topics and optional example topics, and a topic acting in the role of a concluding summary. In addition, the `howto` element refers to the topic that provides an overview of the contents.

```
<!ELEMENT howto ((%topicmeta;)?, (%conceptref;)*, ((%taskref;),
(%exampleref;)?)+,
      (%summaryref;))>
<!ATTLIST howto
  navtitle     CDATA     #IMPLIED
  id           ID        #IMPLIED
  href         CDATA     #IMPLIED
  keyref       CDATA     #IMPLIED
  query        CDATA     #IMPLIED
  conref       CDATA     #IMPLIED
  copy-to      CDATA     #IMPLIED
  %topicref-atts;
  %select-atts;>
```

The `conceptref` and `taskref` elements have a restricted type, meaning that validating processing is obligated to report an error if the referenced topic doesn't have the declared type (or a specialization from the declared type):

```
<!ELEMENT conceptref ((%topicmeta;)?, (%conceptref;)*)>
<!ATTLIST conceptref
  href         CDATA     #IMPLIED
  type         CDATA     "concept"
  ...>
<!ELEMENT taskref    ((%topicmeta;)?, (%taskref;)*)>
<!ATTLIST taskref
  href         CDATA     #IMPLIED
  type         CDATA     "task"
  ...>
```

The `exampleref` and `summaryref` elements don't restrict the type but, instead, assign roles to the referenced topics. Because the content list of the `howto` collection topic allows a topic to act as an example and requires a topic to act as a summary, the author is prompted to create topics in those roles, and the roles can be used in processing, for instance, to add a lead-in word to the emitted topic titles.

```
<!ELEMENT exampleref  ((%topicmeta;)?, (%exampleref;)*)>
<!ATTLIST exampleref
  ...>
<!ELEMENT summaryref  ((%topicmeta;)?)>
<!ATTLIST summaryref
  ...>
```

On closer investigation, either or both of these particular roles may turn out to reflect a

persistent topic structure or semantic, in which case it would be appropriate to define topic types and limit the corresponding `topicref` specialization to topics of those types. The general technique, however, of assigning a role to a topic in the context of a collection remains valid.

Finally, the definition module sets the class attribute to declare that the new elements derive from `topicref` and are provided by the howto package:

```
<!ATTLIST howto %global-atts;
    class CDATA "- map/topicref howto/howto ">
<!ATTLIST conceptref %global-atts;
    class CDATA "- map/topicref howto/conceptref ">
...
```

Assembling the shell DTD

As with topic domains, a shell DTD assembles the base map module with the entities file and definition module for the how-to domain:

```
<!--vocabulary declarations-->
<!ENTITY % howto-d-dec PUBLIC "-//IBM//ENTITIES DITA How To Map
Domain//EN" "howto.ent">
  %howto-d-dec;
...

<!--vocabulary substitution (one for each extended base element,
    with the names of the domains in which the extension was declared)-->
<!ENTITY % topicref  "topicref | %mapgroup-d-topicref; |
%howto-d-topicref;">

<!--vocabulary attributes (must be declared ahead of the default
definition) -->
<!ENTITY included-domains "&mapgroup-d-att; &howto-d-att;">

<!--Embed map to get generic elements -->
<!ENTITY % map-type PUBLIC "-//IBM//Elements DITA Map//EN"
"../../dtd/map.mod">
  %map-type;

<!--vocabulary definitions-->
...

<!ENTITY % howto-d-def PUBLIC "-//IBM//ELEMENTS DITA How To Map
Domain//EN" "howto.mod">
  %howto-d-def;
```

Creating a collection with the domain

Using the shell DTD, a map could include one or more how-to collections, as in the following example:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE map PUBLIC "-//IBM//DTD DITA How To Map//EN"
  "howtomap.dtd">
<map>
  <!-- how-to clusters can appear anywhere in a map hierarchy but always
       follow a consistent information pattern within the how to -->
  <howto href="dita-mapdomains.xml">
    <conceptref href="informationArchitecture.xml"/>
    <conceptref href="mapBackground.xml"/>
    <conceptref href="formalCollection.xml"/>
    <conceptref href="mapSpecialization.xml"/>
    <taskref href="implementDomain.xml"/>
    <exampleref href="declareEntities.xml"/>
    <exampleref href="domainModule.xml"/>
    <exampleref href="assembleDTD.xml"/>
    <exampleref href="domainInstance.xml"/>
    <summaryref href="summary.xml"/>
  </howto>
</map>
```

In fact, this example is the map for the article that you're reading right now. That is, as

you may well have noticed, this article conforms to the formal pattern for a how-to collection. Here's the list of topics in this how-to article but with the addition of the topic type or role and title:

- howto: How to define a formal information architecture with DITA map domains
  - concept: Formal information architecture
  - concept: Specializing topics and maps
  - concept: The how-to collection
  - concept: Map specialization
  - task: Implementing a map domain
  - example: Declaring the map domain entities
  - example: Defining the map domain module
  - example: Assembling the shell DTD
  - example: Creating a collection with the domain (this topic)
  - summary: Summary

While this article contains only a how-to collection, a how-to collection could be part of a larger deliverable. For instance, a help system could include multiple how-tos as part of a navigation hierarchy. Similarly, how-to collections could be used in books by creating a new shell DTD that combines the bookmap map type with the how-to map domain.

As you explore collection types, you'll find that, in addition to topics, a collection can aggregate smaller collections. For instance, you could create domains for a how-to collection, a case study collection, and a reference set collection. A product information collection could then require a product summary topic and at least one of each of these subordinate collections in that order.

You'll also find that, to represent a high-level relationship with a collection, you can create a relationship to the root topic for the collection branch. As the introduction and entry point for the collection, the root topic should provide the most statement of the content of the collection. That is, you can treat the set of topics as a collective content object, using the root topic to represent the collection as a whole for navigation and cross references.

Summary

In this article, you've learned how to specialize the `topicref` element to mandate a specific collection of topics. For complete, single-purpose collections such as functional specifications and quick reference guides, you might package these specialized `topicref` elements with a new map type. For building-block collections (such as how-tos or case studies) that can appear within a large deliverable, especially when different designers might create different collection types, you might want to package the specialized `topicref` elements as a map domain.

By specializing a DITA map in this way, you can implement a formal information architecture not just at the micro level within topics but at the macro level across topics. By defining such large-scale collective content objects, you can provide guidance to authors and declare semantics for processors with the end result that users have consistent and complete information deliverables.