Darwin Information Typing Architecture (DITA) Specification 1.0

First Edition (February 2005)

This edition applies to version 1.0 of the Darwin Information Typing Architecture (DITA) and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

Chapter 1. About the DITA Specification

The Darwin Information Typing Architecture (DITA) specification defines both a) a set of document types for authoring and organizing topic-oriented information; and b) a set of mechanisms for combining and extending document types using a process called specialization.

The specification consists of:

- The DTDs and schemas that define DITA markup for the base DITA document types, as well as catalog files
- The language reference that provides explanations for each element in the base DITA document types
- This document, which comes in three parts:
  - an introduction, which provides background concepts and an overview of the architecture
  - the DITA markup specification, which provides an overview of DITA's base document types
  - the DITA specialization specification, which provides details of the mechanisms DITA provides for defining and extending DITA document types.

This document is part of the technical specification for the DITA architecture. While the specification does contain some introductory information, it is not intended as an introduction to DITA nor as a users guide. The intended audience of this specification consists of implementers of the DITA standard, including tool developers and specializers.

Chapter 2. An introduction to DITA

DITA is an architecture for creating topic-oriented, information-typed content that can be reused and single-sourced in a variety of ways. It is also an architecture for creating new topic types and describing new information domains based on existing types and domains.

The process for creating new topic types and domains is called specialization. Specialization allows the creation of very specific, targeted document type definitions while still sharing common output transforms and design rules developed for more general types and domains, in much the same way that classes in an object-oriented system can inherit methods of ancestor classes.

DITA topics are XML conforming. As such, they are readily viewed, edited, and validated with standard XML tools, although some features such as content referencing and specialization may benefit from customized support.

Definitions and background concepts

The following terms have specific meanings in DITA which should be understood before reading either the DITA markup specification or the DITA specialization specification.

Basic concepts

The following are basic concepts used in DITA.

"What are topics?" on page 7
A topic is a unit of information with a title and content, short enough to be specific to a single subject or answer a single question, but long enough to make sense on its own and be authored as a unit.

"What are maps?" on page 14
DITA maps are documents that collect and organize references to DITA topics to indicate the relationships among the topics. They can also serve as outlines or tables of contents for DITA deliverables and as build manifests for DITA projects.

"What is specialization?" on page 29
Specialization allows you to define new kinds of information (new structural types or new domains of information), while reusing as much of existing design and code as possible, and minimizing or eliminating the costs of interchange, migration, and maintenance.

"Structural versus domain specialization" on page 30
Structural specialization defines new types of structured information, such as new topic types or new map types. Domain specialization creates new markup that can be useful in multiple structural types, such as new kinds of keywords, tables, or lists.

"Integration" on page 39
Each domain specialization or structural specialization has its own design module. These modules can be combined to create many different document types. The process of creating a new document type from a specific combination of modules is called integration.

"Customization" on page 46
When you just need a difference in output, you can use DITA customization to override the default output without affecting portability or interchange, and without involving specialization.

"Generalization" on page 36
Specialized content can be generalized to any ancestor type. The generalization process can preserve information about the former level of specialization to allow round-tripping between specialized and unspecialized forms of the same content.

Terminology

DITA uses a number of terms in particular or unique ways. Within the scope of this specification, the following terms are used when talking about DITA models, DITA declarations, and DITA instances.

Model terminology

DITA can be understood at the level of an abstract model without reference to particular DTDs, schemas, or actual XML documents. When discussing DITA concepts at this level, the following terminology is used.

Element type
Defines the structure and semantics for a fragment of content.

Specialized element type
Defines an element type as a semantic refinement of another element type. The content allowed by the specialized element type must be a subset of or identical to the content allowed by the original element type.

Topic type
An element type that defines a complete unit of content. The topic type provides the root element for the topic and, through contained element types, substructure for the topic instances. The root element of the topic type is not necessarily the same as the root element of a document type: document types may nest multiple topic types, and may also declare non-DITA wrapper elements as the root element for compatibility with other processes.

Map type
An element type that defines a set of relationships for topic instances. The map type provides the root element and, through contained element types, substructure for the map instances. The map substructure provides hierarchy, group, and matrix organization of references to topic instances.

Structural type
A topic type or map type.

Domain
A set of elements that support a specific subject area. Elements in a domain can be integrated with topic or map types to enhance their semantic support for particular kinds of content. For example, the structural type <topic> declares the <keyword> element; when integrated with a domain for describing user interfaces, new keyword specializations (such as <wintitle>) become available wherever <keyword> was allowed in the original structural type.

Document type
The full set of element types defined in the modules that are integrated by the document type shell. A DITA document type may support authoring multiple topic types or multiple map types, but not a mix of the two. The

structural types can be augmented with elements from domains. The term "document type" is used for compatibility with existing standards, since this is the point at which DITA's set of topic, domain, and map types are assembled into a document type that is functionally equivalent to a traditional non-modularized document type.

Declaration terminology

When the model is expressed in a DTD or schema, the various element types are declared. When referring to these declarations, the following terminology is used.

Element declaration
The representation within a schema technology (such as DTD, XML Schema, or Relax NG) for an element type.

Type module
The representation within a schema technology for the element types uniquely defined by a topic type, map type, or domain.

Topic module
The representation within a schema technology for the element types uniquely defined by a topic type.

Map module
The representation within a schema technology for the element types uniquely defined by a map type.

Structural module
A topic or map module.

Domain module
The representation within a schema technology for the element types uniquely defined by a domain.

Document type shell
The representation within a schema technology for a shell that declares no element types itself but points to and assembles topic, map, and domain modules.

Document type declaration
The representation within a schema technology for a document type. The document type declaration includes the declaration modules assembled by the document declaration shell.

Instance terminology

When actual documents, topics, and elements are created based on a DITA document type, the following terminology is used.

Element instance
An occurrence of an element type in a document.

Topic instance
An occurrence of a topic type in a document.

Map instance
An occurrence of a map type in a document.

Structural type instance
An occurrence of a topic type or a map type in a document.

Document instance
A document whose meaning and validity are determined by a document type declaration.

Naming conventions and file extensions

The following naming conventions and file extensions are in use by DITA.

DITA topics
*.xml, *.dita

DITA maps
*.ditamap

DTD structural type files
typename.mod

DTD domain type files
typename.mod

typename.ent

Schema structural type files
*typename*_mod.xsd

*typename*_group.xsd

Schema domain type files
typename-domain.xsd

CSS override files
typename.css

customization-purpose.css

XSLT override files
typename.xsl

customization-purpose.xsl

Chapter 3. DITA markup

The two main units of authoring in DITA are topics and maps. Each can be extended into new structural types and domains through specialization.

DITA topics

DITA topics are the basic units of DITA content. Each topic should be organized around a single subject.

What are topics?

A topic is a unit of information with a title and content, short enough to be specific to a single subject or answer a single question, but long enough to make sense on its own and be authored as a unit.

In DITA, a topic is the basic unit of authoring and of reuse. A document may contain one topic or multiple topics, and a document type may support authoring one or many kinds of topics. But regardless of where they occur, all topics have the same basic structure and capabilities. Books, PDF files, Websites, and help sets, for example, can all be constructed from the same set of underlying topic content, although there may be some topics that are unique to a particular deliverable, and the organization of topics may differ to take advantage of the unique capabilities of each delivery mechanism.

Reference information is inherently topic-oriented, since it requires information to be modular and self-contained for the sake of retrievability.

Topic-oriented authoring for conceptual and task information has its roots in Minimalism, an instructional design technique first espoused by John Carroll. The minimalist approach to information design focusses on identifying the smallest amount of instruction that allows for the successful completion of a task, or that provides basic knowledge of a concept. Readers have goals, and they want to achieve those goals as quickly as possible. Generally, readers don't want to read information just for the pleasure of reading. They are reading to learn or to do something.

Some of the key principles of Minimalism are:

- Support actions. Let people act as they learn, and let them pursue the goals they want to accomplish.
- Document tasks, not tools or functions.
- Help readers anticipate and avoid errors.
- Let readers explore. They don't need explained what they can discover for themselves.

While DITA's topic-oriented approach has its roots in instructional design, the topic-based approach can be useful for any information that has human readers and a consistent structure.

Why topics?

Topics are the basis for high-quality information. They should be short enough to be easily readable, but long enough to make sense on their own.

By organizing content into topics, authors can achieve several goals simultaneously:

- Content is readable even when accessed from an index or search, not just when read in sequence as part of a chapter. Since most readers don't read information end-to-end, it's good information design to make sure each unit of information can be read on its own to give just-in-time help.
- Content can be organized differently for online and print purposes. Authors can create task flows and concept hierarchies for online orientation, and still have a print-friendly combined hierarchy that helps people who do want an organized reading flow.
- Content can be reused in different collections. Since the topic is written to make sense when accessed randomly (as by search), it should also make sense when included as part of different product deliverables, so authors can refactor information as needed, including just the topics that apply for each reuse scenario.

Topics are small enough to provide lots of opportunities for reuse, but large enough to be coherently authored and read. While DITA supports reuse below the topic level, this requires considerably more thought and review, since topics assembled out of smaller chunks often require editing to make them flow properly. By contrast, since topics are already organized around a single subject, authors can organize a set of topics logically and get an acceptable flow between them, since transitions from subject to subject don't need to be as seamless as the explanations within a single subject.

Information typing

Information typing is the practice of identifying types of topics that contain distinct kinds information, such as concepts, tasks, and reference information. Topics that answer different kinds of questions can be categorized as different information types. The base topic types provided by DITA (a generic topic, plus concept, task, and reference) provide a usable starter set that can be adopted for immediate authoring.

Classifying information by type helps authors:

- Design new information more easily and consistently.
- Ensure the right design gets used for the kind of information (retrieval-oriented structures like tables for reference information, simple sequences of steps for task information)
- Focus on tasks.
- Factor out supporting concepts and reference information into other topics, where they can be read if required and ignored if not.
- Eliminate unimportant or redundant information. Identify common or reusable subjects.

Information typing is part of the general authoring approach called structured writing, which is used across the technical authoring industry to improve information quality. It is based on extensive research and experience, including Robert Horn's Information Mapping, and Hughes Aircraft's STOP (Sequential Thematic Organization of Proposals).

Information types in DITA are expressed as topic types. The base topic types provided by DITA can be used as a base for further specialization. New information types that require different structures and semantics are directly

supported by topic type modules, each of which defines the specific markup and structural rules required to describe a particular type of topic. These modules can then be integrated into document types to support authoring information-typed topics.

Topic structure

All topics have the same basic structure, regardless of topic type: title, description, prolog, and body.

All DITA topics must have an ID, a title, and a body. Topic structures can consist of the following parts:

Topic element
Required *id* attribute, contains all other elements

Title The subject of the topic.

Alternate titles
Titles specifically for use in navigation or search. When not provided, the base title is used for all contexts.

Short description
A short description of the topic. Used both in topic content, in generated summaries that include the topic, and in links to the topic. While short descriptions aren't required, they can make a dramatic difference to the usability of an information set, and should generally be provided for all topics.

Prolog Container for various kinds of topic metadata, such as change history, audience, product, and so on.

Body The actual topic content: paragraphs, lists, sections - whatever the information type allows.

Related links
Links to other topics. When an author creates a link as part of a topic, the topic becomes dependent on the other topic being available. To reduce dependencies between topics and thereby increase the reusability of each topic, authors can use DITA maps to define and manage links between topics, instead of embedding links directly in each related topic.

Nested topics
Topics can be defined inside other topics. Nesting can result in complex documents that are less usable and less reusable, and should be used carefully. It is more often appropriate for reference information, which can support longer documents organized into multiple topics for scanning and retrieval.

Topic content

All topics, regardless of topic type, build on the same common structures.

Topic bodies
While all topic types have the same elements for title, short description, and prolog, they each allow different content in their body.

Sections and examples
Sections and examples can be contained only by the body of a topic. They cannot nest. They can contain block-level elements like paragraphs, phrase-level elements like API names, or text.

Block-level elements
Paragraphs, lists, and tables are kinds of ″block″ elements. As a class of content, they can contain other blocks, phrases, or text, though the rules vary for each structure.

Phrases and keywords
Authors can intermix markup with text when they need to identify part of a paragraph or even part of a sentence as having special significance. Phrases can usually contain other phrases and keywords as well as text. Keywords can only contain text.

Images
Authors can insert images using the image element. Images can be used at the block level, for example to show screen captures or diagrams, or at the phrase level, for example to show what icons or toolbar buttons look like.

Multimedia
Authors can create multimedia for online information using the object element, for example to display SVG diagrams that can be rotated and explored.

Topic modules

There are three basic modules in topic: for tables, for metadata, and for everything else.

tbl_xml.mod
Defines the elements for authoring tables, based on the CALS table model but with some DITA-specific extensions.

meta_xml.mod
Defines metadata elements. Also used by DITA maps, where metadata can be defined for multiple topics at once.

topic.mod
Defines the rest of the elements in a topic.

Concepts

DITA concept topics answer ″What is...″ questions. They include a body-level element with a basic topic structure, including sections and examples.

Why concepts?

Concepts provide background that helps readers understand essential information about a product, interface, or task. Often, a concept is an extended definition of a major abstraction such as a process or function. Conceptual information may explain a product and how it fits into its category of products. Conceptual information helps users to map their existing knowledge to tasks and other essential information about a product or system.

Concept structure

The <concept> element is the top-level element for a DITA concept topic. Every concept contains a <title> and a <conbody> and optional <titlealts>, <shortdesc>, <prolog>, and <related-links>.

The <conbody> element is the main body-level element for a concept. Like the body element of a general topic, <conbody> allows paragraphs, lists, and other elements as well as sections and examples. But <conbody> has a constraint that a

section or an example can be followed only by other sections or examples.

Here is an example of a simple concept topic.

```
<concept id="concept">
 <title>Bird Calling</title>
 <conbody>
  <p>Bird calling attracts birds.</p>
  <example>
   <p>Bird calling requires learning:</p>
   <ul>
    <li>Popular and classical bird songs</li>
    <li>How to whistle like a bird</li>
   </ul>
  </example>
 </conbody>
</concept>
```

Modules

dtd\concept.mod, schema\concept.mod

Tasks

Task topics answer ″How do I?″ questions, and have a well-defined structure that describes how to complete a procedure to accomplish a specific goal.

Why tasks?

Tasks are the essential building blocks for providing procedure information. A task topic answers the ″How do I?″ question by providing precise step-by-step instructions detailing what to do and the order in which to do it. The task topic includes sections for describing the context, prerequisites, expected results, and other aspects of a task.

Task structure

The <task> element is the top-level element for a task topic. Every task topic contains a <title> and a <taskbody> and optional <titlealts>, <shortdesc>, <prolog>, and <related-links>.

The <taskbody> element is the main body-level element inside a task topic. A task body has a very specific structure, with the following elements in this order: <prereq>, <context>, <steps>, <result, <example> and <postreq>. Each of the body sections is optional.

<prereq>
Describes information needed before starting the current task.

<context>
Provides background information for the task. This information helps the user understand what the purpose of the task is and what they will gain by completing the task. This section should be brief and does not replace or recreate a concept topic on the same subject, although the context section may include some conceptual information.

<steps>
Provides the main content of the task topic. A task consists of a series of steps that accomplish the task. The <steps> section must have one or more <step> elements, which provide the specifics about each step in in the task.

The <step> element represents an action that a user must follow to accomplish a task. Each step in a task must contain a command <cmd> element which describes the particular action the user must do to accomplish the overall task. The step element can also contain information <info>, substeps <substeps>, tutorial information <tutorialinfo>, a step example <stepxmp>, choices <choices> or a stepresult <stepresult>, although these are optional.

<result>
Describes the expected outcome for the task as a whole.

<example
Provides an example that illustrates or supports the task.

<postreq>
Describes steps or tasks that the user should do after the successful completion of the current task. It is often supported by links to the next task or tasks in the <related-links> section.

Here's an example of a task topic.

```
<task id="ertx">
 <title>Creating an ERTX file</title>
 <taskbody>
  <context>Each morning before breakfast you need to create a fresh ERTX file.</context>
  <steps>
   <step><cmd>Start ERTX.</cmd></step></steps>
   <step><cmd>Click New ERTX File.</cmd></step></steps>
  </steps>
  <result>You now have your ERTX file for today!</result>
 </taskbody>
</task>
```

Modules

dtd\task.mod, schema\task.mod

Reference

Reference topics describe regular features of a subject or product, such as commands in a programming language.

Why reference?

In technical information, reference topics are often used to cover subjects such as the commands in a programming language. Reference topics can hold anything that has regular content, such as ingredients for food recipes, bibliographic lists, catalogues, and the like. Reference topics provide quick access to facts. Information needed for deeper understanding of a reference topic or to perform related procedures should be provided in a concept or task topic.

Reference structure

The <reference> element defines a top-level container for a reference topic. Reference topics have the same high-level structure as the other core DITA topic types, with a title, short description, and body. Within the body, reference topics organize content into one or more sections, property lists, or tables.

The <refbody> element holds the main content of the reference topic. Reference topics limit the body structure to tables (both simple and standard), property lists, syntax sections, and generic sections and examples.

All of the elements of <refbody> are optional and may appear in any sequence and number.

<section>
Represents an organizational division in a reference topic. Sections organize subsets of information within a larger topic. You can only include a simple list of peer sections in a topic; sections cannot be nested. A section may have an optional title.

<refsyn>
Contains syntax or signature content (for example, a command-line utility's calling syntax, or an API's signature). The <refsyn> contains a brief, possibly diagrammatic description of the subject's interface or high-level structure.

<example>
Provides containing examples that illustrate or support the current topic. The <example> element has the same content model as <section>.

<table>
Organizes information according into a tabular rows and columns structure. Table markup also allows for more complex structures, including spanning rows and columns, as well as table captions.

<simpletable>
Holds information in regular rows and columns and does not allow for a caption.

<properties>
Lists properties and their types, values, and descriptions.

Here's an example of a reference topic.

```
<reference id = "boldproperty">
<title>Bold property</title>
<shortdesc>(Read-write) Whether to use a bold font for the specified
text string.</shortdesc>
<refbody>
  <refsyn>
    <synph>
      <var>object</var><delim>.</delim><kwd>Font</kwd><delim>.</delim>
      <kwd>Bold</kwd><delim> = </delim><var>trueorfalse</var>
    </synph>
  </refsyn>
  <properties>
    <property>
      <proptype>Data type</proptype>
      <propvalue>Boolean</propvalue>
    </property>
    <property>
      <proptype>Legal values</proptype>
      <propvalue>True (1) or False (0)</propvalue>
    </property>
  </properties>
</refbody>
</reference>
```

Modules

dtd\reference.mod, schema\reference.mod

Domains

A DITA domain defines a set of elements associated with a particular subject area or authoring requirement regardless of topic type.

The elements in a domain are defined in a domain module which can be integrated with a topic type to make the domain elements available within the topic type structure. Currently the following domains are provided:

Table 1. DITA domains

Domain	Description	Short name	Module name
Typographic	For highlighting when the appropriate semantic element doesn't exist yet	hi-d	highlight-domain.mod
Programming	For describing programming and programming languages	pr-d	programming-domain.mod
Software	For describing software	sw-d	software-domain.mod
User interfaces	For describing user interfaces	ui-d	ui-domain.mod
Utilities	For providing imagemaps and other useful structures	ut-d	utilities-domain.mod

DITA maps

Maps organize topics for output to a specific deliverable, including generating navigation files and links to related topics.

What are maps?

DITA maps are documents that collect and organize references to DITA topics to indicate the relationships among the topics. They can also serve as outlines or tables of contents for DITA deliverables and as build manifests for DITA projects.

DITA maps represent the architecture of an information set – what topics are needed, in what order or relationships, to support a particular set of user goals or other requirements.

Maps describe the context in which the topics will be read – the audience, platform, relationships, requirements of the information set. In this way, the topics themselves become relatively context-free, and can be more easily used and reused in many different contexts, as defined by maps.

Maps draw on a rich set of existing best practices and standards for defining information models, such as hierarchical task analysis. They also support the definition of non-hierarchical relationships, such as matrices and groups, which provide a set of capabilities that has some similarities to RDF and ISO topic maps.

A map file references one or more DITA topic files using <topicref> elements. The <topicref> elements can be nested or otherwise organized to reflect the desired relationships between the referenced topics. Map files need to have a file extension of .ditamap to be processed properly

Why DITA maps?

Maps allow scalable reuse of content across multiple contexts. They can be used by information architects, writers, and publishers to plan, develop, and deliver content.

Among the specific uses that maps support:

Defining an information architecture
The map can be used to define what topics are required for a particular audience and user goals, even before the topics themselves exist.

Providing an authoring interface
The map can be used as a starting point for authoring new topics and integrating existing ones.

Defining what topics to build for a particular output
Maps point to topics that are included in output processing. Authors or publishers can use maps to specify a set of topics to transform at the same time, instead of transforming each topic individually.

Defining online navigation
Maps can define the online navigation or table of contents for the topics it points to.

Defining what topics to print
Maps can define a hierarchy that will determine how topics will be combined and nested for printing.

Defining related links
Maps define relationships among the topics they reference; on output, these relationships can be expressed as related links among the topics in each relationship.

Common DITA map attributes and metadata

DITA maps have many of the same common attributes as DITA content, but also have some additional ones for controlling the way relationships are interpreted for different output purposes.

Because DITA maps may encode structures that are wholly or partially specific to a particular medium or kind of output (for example, hyperlinked web pages or printed books), DITA maps contain attributes to help processors interpret the map for each kind of output. These attributes are not available in DITA content: individual topics, once separated from the high-level structures and dependencies associated with a particular kind of output, should be entirely reusable across multiple media.

collection-type, linking

The containment structure in a map can be used to generate related links or references on output. The author can annotate the containment structure to identify a particular set of siblings as being part of a specific type of collection, such as a family or sequence. The collection-type value for a group of siblings can indicate whether to generate links among the siblings, and what kind of links to generate (for example, next and previous links for a sequence). The collection-type attribute can also indicate how the parent topic should link to its children (for example, showing the child links as a numbered list when the collection-type is sequence).

By default, relationships between topics in a map are reciprocal: children link to parents and vice versa; next and previous topics in a sequence link to each other; topics in neighboring table cells link to each other, and so on. This default behavior can be modified using the linking attribute, which lets a topic modify how it participates in a relationship:

- A topic reference with linking="none" does not exist in the map for the purposes of calculating links
- linking="sourceonly" means that the topic will link to its related topics but not vice versa
- linking="targetonly" means that the related topics will link to it, but not vice versa
- linking="normal" is the default, and means that linking will be reciprocal (the topic will link to related topics, and they will link back to it)

```
<topicref href="A.dita" collection-type="sequence">
  <topicref href="A1.dita"/>
  <topicref href="A2.dita"/>
</topicref>
<reltable>
  <relrow>
    <relcell>A.dita</relcell>
    <relcell>B.dita</relcell>
  </relrow>
</reltable>
```

A links to A1, A2, A3 as children

links to B as related

A1 links to A as a parent

links to A2 as next in the sequence

A2 links to A as a parent

links to A1 as previous in the sequence

B links to A as related

Figure 1. Simple linking example

```
<topicref href="A.dita" collection-type="sequence">
  <topicref href="B.dita" linking="none"/>
  <topicref href="A1.dita"/>
  <topicref href="A2.dita"/>
</topicref>
<reltable>
  <relrow>
    <relcell>A.dita</relcell>
    <relcell linking="sourceonly">B.dita</relcell>
  </relrow>
</reltable>
```

A links to A1, A2, A3 as children

(no links to B as a child, no links to B as related)

A1 links to A as a parent

links to A2 as next in the sequence

(no links to B as previous)

A2 links to A as a parent

links to A1 as previous in the sequence

B links to A as related

Figure 2. Linking example with the linking attribute

toc, navtitle, locktitle

Authors can exclude entries from navigation output (such as an online table of contents, or a Web site map) using the toc attribute. By default, hierarchies are included in navigation output, and tables are excluded.

Authors can provide a shorter version of the title for use in the navigation using the navtitle attribute. By default the navtitle attribute is ignored, and used only to help the author keep track of the target topic's title. The locktitle attribute can be set to ensure that the navtitle takes effect and overrides any title values in the target topic, or defined elsewhere in the topic reference metadata.

print, search

You can set attributes on a topic to indicate whether it should be included in printed output and search indexes.

chunk, copy-to

When a set of topics is transformed using a map, multi-topic files can be broken into smaller files, and multiple individual topics can be combined into a single larger file, using the chunk attribute.

New topic versions can be created using the copy-to attribute. The copied topic will have a new file name, and the map can override the default title and shortdesc by providing values for them in the map.

Shared attributes

DITA maps use the same metadata and reuse attributes that DITA topics use:

- product, platform, audience, otherprops, rev, status, importance, xml:lang, translate

- id, conref

DITA maps also use many of the same attributes that are used with link or xref elements in DITA content:

- format, scope, href, keyref, type, query

Shared metadata elements, and the lockmeta attribute

You can associate topic metadata with a topic or branch of topics in a map. By default metadata in the map supplements or overrides metadata in the topic. If the lockmeta attribute is set to ″no″, then the metadata in the map will not take precedence over the metadata in the topic, and conflicts will be resolved in favor of the topic.

The metadata elements in a map are the same as those in a topic, although they may be in a separate order. The map also includes a short description and alternate titles, which can override their equivalents in the content. In sum, the map can override or supplement everything about a topic except its content (in the topic's body element).

DITA map structure

Maps organize topics into hierarchies, tables, and groups, and have special elements for referencing other maps.

topicref elements are the basic elements of a map. A topicref can point to a DITA topic, map, or to any other resource that can be processed or linked to.

topicref elements can be nested to create a hierarchy, which can be used to define print output, online navigation, and parent/child links. The topichead element can be used for nodes in the hierarchy that provide containers without equivalent topics: they are equivalent to topicref elements with a navtitle but no href or equivalent referencing attribute.

Relationship tables are defined with the reltable element. Relationship tables can be used to define relationships among the topics in different cells of the same row. In a relationship table, the columns define common attributes or metadata for the topics in that column. The rows define relationships, with each cell representing a different role in the relationship. For example, a table with different columns for concepts, tasks, and reference topics could be used to define the relationship between a task and the topics that support it.

Both hierarchies and tables can be annotated using the collection-type attribute to define sets of siblings that are part of a particular kind of collection, for example a set of choices, a sequence, or a family. These collection-types can affect link generation, and may be interpreted differently for different outputs.

Groups or collections outside of a hierarchy or table can be defined with the topicgroup element, which is equivalent to a topicref with no referencing attributes or titles. Groups can be combined with hierarchies and tables, for example by including a group within a table cell or within a set of siblings in a hierarchy.

Example of a simple relationship table

```
<reltable>
  <relheader>
    <relcolspec type="concept"/>
    <relcolspec type="task"/>
```

```
    <relcolspec type="reference"/>
  </relheader>
  <relrow>
    <relcell>
      <topicref href="A.dita"/>
    </relcell>
      <topicref href="B.dita"/>
    </relcell>
      <topicref href="C1.dita"/>
      <topicref href="C2.dita"/>
    </relcell>
  </relrow>
</reltable>
```

type="concept"	type="task"	type="reference"
A	B	C1 C2

A links to B, C1, C2

B links to A, C1, C2

C1, C2 link to A, B

Inheritance of attributes and metadata

Some of the attributes and metadata in a map can be inherited based on the structures in the map.

Inheritance is additive except where this would cause a conflict. When there is a conflict, the value defined closest to the topicref takes effect.

The following attributes and metadata elements are inheritable:

Attributes

audience, platform, product, otherprops, rev

linking, toc, print, search

chunk, format, scope, type

Elements

author, source, publisher, copyright, critdates, permissions

audience, category, keywords, prodinfo, othermeta

Attributes and metadata can be defined at the root level (attributes on the map element itself, topicmeta as a direct child of the map element) to apply them to the entire map. They can also be applied at any point in a hierarchy, group, or table. Tables can be particularly useful for attribute and metadata management, since they can be applied to entire columns or rows as well as individual cells.

DITA map modules

Maps have the same module structure as topics, and share some of the same modules for defining metadata.

map.mod defines the base map structures.

mapgroup.mod adds topicgroup and topichead as specialized variants of topicref.

Common metadata elements

The same metadata elements are available in both DITA topic types and DITA map types. This allows the metadata assigned to a topic when it is created to be supplemented or overridden when the topic is included in a collection.

Publication metadata elements

These elements provide standard information about the topic as a publication.

Some content providers might choose to provide such information only in the map or the initial topic for a deliverable.

author The person or organization who created the content. This element is equivalent to the Dublin Core Creator.

publisher
The organization who provides and distributes the content. This element is equivalent to the Dublin Core Publisher.

copyright
The legal ownership for the content. This element is equivalent to the Dublin Core Rights.

Management metadata elements

These elements provide a basis for managing the publication process for topics.

The management elements might get updated by workflow processes or provide input for such processes:

source An identifier or name for the original form of the content. This element is equivalent to Dublin Core Source.

critdates
Milestones in the publishing cycle. This element is equivalent to Dublin Core Date.

permissions
Specification of the level of entitlement needed to access for content.

resourceid
The identifier associated with the topic when provided to the specified application.

Metadata qualification elements

These elements qualify the topic for processes such as flagging, filtering, or retrieval.

The metadata elements apply to an entire topic, and can also be used in a map to apply metadata to multiple topics at a time. Metadata elements can expand on the values used in metadata attributes. (See metadata attributes.) For example, the audience element in a topic's prolog can define an audience in terms of type, job, and experience level, and give it a name; when there is content within the topic's body that applies only to that audience, that content can identify its audience by the same name used in the prolog.

When metadata is expressed in a map, it supplements any metadata expressed in the topics it references. When metadata in a map and a topic conflict (for example,

both define a publisher), by default the value in the map takes precedence, on the assumption that the author of the map has more knowledge of the reusing context than the author of the topic.

audience
The type, job, experience level, and other characteristics of the reader for the topic. Many of these characteristics have enumerated values, but the enumeration can be extended through associated attributes. For instance, the audience type enumeration can be extended through an othertype attribute. The audience element can elaborate values used by audience attributes.

category
A classification of the topic content. Such classifications are likely to come from an enumerated or hierarchical set. This element is equivalent to both Dublin Core Coverage and Dublin Core Subject.

keywords
Terms from a controlled or uncontrolled subject vocabulary that apply to the topic.

prodinfo
The definition of the product or platform for the topic. The prodinfo element can elaborate values used by the product and platform attributes.

othermeta
A name-value pair specifying other metadata about the topic.

Topic properties in topics and maps

The properties of a topic can be specified in the topic itself or on references to the topic within maps.

Within a topic, properties can be expressed using metadata attributes on the topic element or using publication, management, or metadata elements in the topic prolog.

Within a map, the same properties can be expressed on the topicref element that refers to the topic. That is, the topicref attributes and the topicref subelements within the topicmeta container apply to the referenced topic. In addition, the metadata properties map or topicref element set the default properties for nested topicref elements within the map hierarchy. Because the topics in a branch of the navigation hierarchy typically have common subject or properties, this mechanism provides a convenient way to set the properties for a set of topics.

If a property is set in both the map and topic, the map properties are additive if the property (such as the audience type) takes a list of values. If, instead, the property (such as the importance) takes a single value, the map property overrides the topic property.

Example of audience metadata in prolog and body

The practice of providing full metadata in the prolog and referencing it from attributes when a subset of metadata applies is not a best practice. Prolog metadata and attribute metadata can be used and expressed independently. The coordination shown here is possible but is not required.

```
<prolog>
  <metadata>
    <audience name="AdminNovice"
```

```
                type="administrator"
                job="customizing"
                experiencelevel="novice">

  </metadata>
</prolog>
....
<p audience="AdminNovice ProgrammerExp">This paragraph applies to both
novice administrators and expert programmers</p>
```

Common attributes

The following attributes are common across most DITA elements.

Identity attribute

The DITA identity attribute provides mechanisms for identifying content for retrieval or linking.

The id attribute assigns a unique identifier to an element so the element can be referenced. The scope of uniqueness for the id attribute depends on the role of the element within the DITA architecture:

- Because topics are the basic units of information within DITA, the id attribute for the topic must be unique within the document instance.

  A topic architecture assembles topics into a deliverable by reference. To ensure that topics can be referenced, the id attribute is required on the topic element.

  The complete identifier for a topic consists of the combination of the URI for the document instance, a separating hash character, and the topic id (as in `http://some.org/some/directory/topicfile.xml#topicid`).URIs are described in RFC 2396. As is typical with URIs, a relative URI can be used as the identifier for the document instance so long as it is resolvable in the referencing context. For instance, within a file system directory, the filename of the document instance suffices (as in `some/directory/topicfile.xml#topicid`).Within the same document, the topic id alone suffices (as in `#topicid`). Where the topic element is the root element of the document instance, contexts outside the document instance may omit the topic id when referring to the topic element (as in `topicfile.xml`).

  The topic id can be referenced by topicrefs, links, xrefs, or conrefs to the topic as well as indirectly as part of references to the topic content.

  The id attribute for DITA topics is of type ID in XML.

- Because topic content is always contained within a topic, the id attribute for a topic content element must be unique only within the topic. This approach ensures maintainable references to content because the identifier remains valid so long as the document instance, topic, and content exist. The position of the content within the topic and the position of the topic within the document instance can change without invalidating the content identifier. In addition, this approach avoids the need to rewrite topic content ids to avoid naming collisions when aggregating topics.

  The id is optional and need be added only to make the content referenceable.

  The complete identifier for topic content consists of the combination of the complete identifier for the topic, a separating solidus (/), and the topic content id (as in `http://some.org/some/directory/topicfile.xml#topicid/contentid`). As noted before, the topic identifier portion can use a relative URI for the document instance in contexts where the relative URI can be resolved (as in `some/directory/topicfile.xml#topicid/contentid`).

The containing topic id must always be included when referencing an element id. Otherwise, a reference to another topic couldn't be distinguished from a reference to an element within the same topic. For references within the same document instance, the identifier for the document instance can be omitted altogether (as in `#topicid/contentid`).

The id attribute for elements within DITA topics is not of type ID and is not required to be unique.

- For a map, the id of a map, topicref, or anchor must be unique within the document instance. This approach ensures that these elements can be referenced outside the map without qualification by the map id.

  For the anchor element, which exists only to identify a position within a map as a target for references, the id attribute is required. For the other elements, the id attribute is optional.

  As with a topic, the complete identifier consists of the combination of the absolute URI for the map document instance and the element id (as in `http://some.org/some/directory/mapfile.xml#topicrefid`).

  The id attribute for maps, topicrefs, and anchors is of type ID.
- The id for a relationship table element must be unique only within the map.

  As with topic content, the full identifier consists of the combination of the absolute URI for the map and the id for the relationship table element (as in `http://some.org/some/directory/mapfile.xml#mapid/reltableid`).

  The id attribute for reltable elements is not of type ID and is not required to be unique.

Content reference attribute

The DITA conref attribute provides a mechanism for reuse of content fragments. The conref attribute stores a reference to another element and is processed to replace the referencing element with the referenced element.

The element containing the content reference acts as a placeholder for the referenced element. The identifier for the referenced element must be either absolute or resolvable in the context of the referencing element. (See "Identity attribute" on page 22 for the details on identifiers.)

More formally, the DITA conref attribute can be considered a transclusion mechanism. In that respect, conref is similar to XInclude as well as HyTime value references. DITA differs from these mechanisms, however, by comparing the constraints of each context to ensure the ongoing validity of the replacement content in its new context. In other words, conref validity does not apply simply to the current content at the time of replacement, but to the ranges of possible content given the constraints of the two document types. A valid conref processor does not allow the resolution of a reuse relationship that could be rendered invalid under the rules of either the reused or reusing content.

If the referenced element is the same type as the referencing element and the list of domains in the referenced topic instance (declared on the domains attribute) is the same as or a subset of the list of domains in the referencing document, the element set allowed in the referenced element is guaranteed to be the same as, or a subset of, the element set allowed in the placeholder element. In the preferred approach, a processor resolving a conref should tolerate specializations of valid elements and generalize elements in the content fragment as needed for the referencing context.

Replacement of the placeholder occurs after parsing of the document but prior to any styling or other transformational or presentational operations on the full topic.

The target of the conref may be substituted based on build-time or runtime conditions. For example, content such as product names or install paths can be separated out from topic content since they change when the topic is reused by other products; the reusing product can substitute their own targets for the conref to allow resolution to their own product name and install paths, and so on.

Metadata attributes

The metadata attributes express qualifications on the content. These qualifications can be used to modify the processing of the content.

One typical use of the metadata attributes is to filter content based on their values. Another typical use is to flag content based on their values, for example by highlighting the affected text on output. Typically audience, platform, product, and otherprops are used for filtering, and the same attributes plus rev are used for flagging. Status and importance are used for tool-specific or transform-specific behavior, for example marking steps in a task as optional or required.

In general, a metadata attribute provides a list of one or more qualification values, separating those values with whitespace. For instance, an audience attribute of `administrator programmer` qualifies the content as applying to administrators and programmers.

For a topic, the audience, platform, and product metadata can be expressed with attributes on the topic element or with elements within the topic prolog. While the metadata elements are more expressive, the meaning of the values is the same, and can be used in coordination: for example, the prolog elements can fully define the audiences for a topic, and then metadata attributes can be used within the content to identify parts that apply to only some of those audiences.

audience

The values from the enumerated attributes of the audience metadata element have the same meaning when used in the audience attribute of a content element. For instance, the "user" value has the same meaning whether appearing in the type attribute of the audience element for a topic or in the audience attribute of a content element. The principle applies to the type, job, and experience level attributes of the audience element.

The values in the audience attribute may also be used to reference a more complete description of an audience in an audience element. Use the name of the audience in the audience element when referring to the same audience in an audience attribute.

The audience attribute takes a blank-delimited list of values, which may or may not match the name value of any audience elements.

platform

The platform might be the operating system, hardware, or other environment. This attribute is equivalent to the platform element for the topic metadata.

The platform attribute takes a blank-delimited list of values, which may or may not match the content of a platform element in the prolog.

product
The product or component name, version, brand, or internal code or number. This attribute is equivalent to the prodinfo element for the topic metadata.

The product attribute takes a blank-delimited list of values, which may or may not match the value of the prodname element in the prolog.

importance
The degree of priority of the content. This attribute takes a single value from an enumeration.

rev The identifier for the revision level.

status The current state of the content. This attribute takes a single value from an enumeration.

otherprops
A catchall for metadata qualification values about the content. This attribute is equivalent to the othermeta element for the topic metadata.

The product attribute takes a blank-delimited list of values, which may or may not match the values of othermeta elements in the prolog.

The attribute can also take labelled groups of values. A labelled group consists of a string value followed by an open parenthesis followed by one or more blank-delimited values followed by a close parenthesis. The simple format is sufficient when an information set requires only one additional metadata axis, in addition to the base metadata attributes of product, platform, and audience. The full format is useful when an information set requires two or more additional metadata axes. A process can detect which format is in use by the presence of parentheses in the attribute.

For example, a simple otherprops value list: `<codeblock otherprops="java cpp">`

For example, a complex otherprops value list: `<codeblock otherprops="proglang(java cpp) commentformat(javadoc html)">`

Miscellaneous Attributes

The xml:lang attribute identifies the language of a topic or content fragment. The outputclass attaches a classifying label to an element.

Miscellaneous attributes of DITA elements include the following

xml:lang
The xml:lang attribute's behavior is described in detail in the XML specification: http://www.w3.org/TR/REC-xml/#sec-lang-tag The attribute identifies a language by means of the standard language and country codes (as described in RFC 3066). For instance, French Canadian would be identified by the value `fr-ca`. As is usual, the language applies to the contained content and attributes of the current element and contained elements, other than fragments that declare a different language.

outputclass
The outputclass attribute provides a label on one or more element instances, typically to specify a role or other semantic distinction. As the outputclass attribute doesn't provide a formal type declaration or the structural consistency of specialization, it should be used sparingly, often only as a temporary measure while a specialization is developed. For example, <uicontrol> elements that define button labels could be

distinguished by adding an outputclass: <uicontrol outputclass="button">Cancel</uicontrol>. The outputclass value could be used to trigger XSLT or CSS rules, as well as providing a mapping to be used for future migration to a more specialized set of UI elements.

Architectural attributes

DITA provides some attributes to provide type information to processors instead of qualifications or properties of content.

Ordinarily, architectural attributes don't appear in the source files for document instances. Instead, architectural attributes appear in document instances through defaults set in the DTD or Schema declaration. This practice ensures that the creation of document instances cannot produce invalid values for the architectural attributes. These attributes are as follows:

class This attribute identifies the specialization module for the element type as well as the ancestor element types and the specialization modules to which they belong. Every DITA element has a class attribute.

domains
This attribute identifies the domain specialization modules used in a topic and, for each domains module, its module dependencies. Every topic and map element has a domains attribute.

DITAArchVersion
This attribute identifies the version of the DITA architecture used by the DTD or schema.Every topic and map element has a DITAArchVersion attribute. The attribute is declared in a DITA namespace to allow namespace-sensitive tools to detect DITA markup.

To make the document instance usable without the DTD or Schema declaration, a normalization process can instill the architectural attributes in the document instance.

Conditional processing

Conditional processing is the filtering or flagging of information based on processing-time criteria

DITA tries to implement conditional processing in a semantically meaningful way: rather than allowing arbitrary values to accumulate in a document over time in a general-purpose processing attribute, with meaning only to the original author, we encourage the authoring of metadata using specific metadata attributes on content. These metadata values can then be leveraged by any number of processes, including filtering, flagging, search, and indexing, rather than being suitable for filtering only.

There are four attributes intended for conditional processing, available on most elements:

- product: the product that is the subject of the discussion.
- platform: the platform on which the product is deployed.
- audience: the intended audience of the text
- rev: the revision or draft number of the current document (typically used for flagging only, not for filtering)
- otherprops: anything else

Using metadata attributes

Each attribute takes zero or more space-delimited string values. For example, you can use the product attribute to identify that an element applies to two particular products.

```
<p audience="administrator">Set the configuration options:
 <ul>
  <li product="extendedprod">Set foo to bar</li>
  <li product="basicprod extendedprod">Set your blink rate</li>
  <li>Do some other stuff</li>
  <li platform="Windows">Do a special thing for Windows</li>
 </ul>
</p>
```

Figure 3. Example source

Processing metadata attributes

At processing time, you specify the values you want to exclude and the values you want to flag. For example, a publisher producing information for a mixed audience using the basic product could choose to flag information that applies to administrators, and exclude information that applies to the extended product:

```
<prop att="audience" val="administrator" action="flag" use="ADMIN"/>
<prop att="product"  val="extendedprod"  action="exclude"/>
```

The format shown here for identifying values for filtering and flagging is not normative, and is shown purely for the sake of illustrating the expected processing logic.

At output time, the paragraph is flagged, and the first list item is excluded (since it applies to extendedprod), but the second list item is still included (even though it does apply to extendedprod, it also applies to basicprod, which was not excluded).

The result should look something like:

ADMIN Set the configuration options:

- Set your blink rate
- Do some other stuff
- Do a special thing for Windows

Filtering logic

When deciding whether to exclude a particular element, a process should evaluate each attribute, and then evaluate the set of attributes:

- If all the values in an attribute have been set to ″exclude″, the attribute evaluates to ″exclude″
- If any of the attributes evaluate to exclude, the element is excluded.

For example, if a paragraph applies to three products and the publisher has chosen to exclude all of them, the process should exclude the paragraph; even if the paragraph applies to an audience or platform that you aren't excluding. But if the paragraph applies to an additional product that has not been excluded, then its content is still relevant for the intended output and should be preserved.

Flagging logic

When deciding whether to flag a particular element, a process should evaluate each value. Wherever a value that has been set as flagged appears in its attribute (for example, audience="ADMIN") the process should add the flag. When multiple flags apply to a single element, multiple flags should be output, typically in the order they are encountered.

Flagging could be done using text (for example, bold text against a colored background) or using images. When the same element evaluates as both flagged and filtered (for example, flagged because of an audience attribute value and filtered because of its product attribute values), the element should be filtered.

Chapter 4. DITA specialization

Specialization is the process by which new designs are created based off existing designs, allowing new kinds of content to be processed using existing processing rules.

Specialization provides a way to reconcile the needs for centralized management of major architecture and design with the needs for localized management of group-specific and content-specific guidelines and behaviors. Specialization allows multiple definitions of content and output to co-exist, related through a hierarchy of types and transforms. This hierarchy lets general transforms know how to deal with new, specific content, and it lets specialized transforms reuse logic from the general transforms. As a result, any content can be processed by any transform, as long as both content and transform are specialization-compliant, and part of the same hierarchy. Specializers get the benefit of specific solutions, but also get the benefit of common standards and shared resources.

Content	Processing	Result
Unspecialized	Unspecialized	Base processing, expected output
Unspecialized	Specialized	Base processing, specialized overrides are ignored, expected output
Specialized	Unspecialized	Base processing, specialized content treated as general, output may fall short of expectations
Specialized	Specialized	Specialized processing, expected output
Specialized	Differently specialized	Some specialized processing, specialized content treated as nearest common denominator, output may fall short of expectations

The following topics provide an overview of specialization, some recommendations for use, and detailed rules for its mechanisms.

What is specialization?

Specialization allows you to define new kinds of information (new structural types or new domains of information), while reusing as much of existing design and code as possible, and minimizing or eliminating the costs of interchange, migration, and maintenance.

Specialization is used when new structural types or new domains are needed. DITA specialization can be used when you want to make changes to your design for the sake of increased consistency or descriptiveness or have extremely specific needs for output that cannot be addressed using the current data model. Specialization is not recommended for simply creating different output types as

DITA documents may be transformed to different outputs without resorting to specialization (see “Customization” on page 46).

There are two kinds of specialization hierarchy: one for structural types (with topic or map at the root) and one for domains (with elements in topic or map at their root). Structural types define topic or map structures, such as concept or task or reference, which often apply across subject areas (for example, a user interface task and a programming task may both consist of a series of steps). Domains define markup for a particular information domain or subject area, such as programming, or hardware. Each of them represent an “is a” hierarchy, in object-oriented terms, with each structural type or domain being a subclass of its parent. For example, a specialization of task is still a task; and a specialization of the user interface domain is still part of the user interface domain.

Use specialization when you are dealing with new semantics (new, meaningful categories of information, either in the form of new structural types or new domains). The new semantics can be encoded as part of a specialization hierarchy, that allows them to be transformed back to more general equivalents, and also ensures that the specialized content can be processed by existing transforms.

Why specialization?

Specialization can have dramatic benefits for the development of new document architectures.

Among the benefits:

- No need to reinvent the base vocabulary - Create a module in 1/2 day with 10 lines vs. 6 months with 100s of lines; automatically pick up changes to the base
- No impact from other designs that customize for different purposes - Avoid enormous, kitchen-sink vocabularies; Plug in the modules for your requirements
- Interoperability at the base type - Guaranteed reversion from special to base
- Reusable type hierarchies - Share understanding of information across groups, saving time and presenting a consistent picture to customers
- Output tailored to customers and information - More specific search, filtering, and reuse that is designed for your customers and information not just the common denominator
- Consistency - Both with base standards and within your information set
- Learning support for new writers - Instead of learning standard markup plus specific ways to apply the markup, writers get specific markup with guidelines built in
- Explicit support of different product architectural requirements - Requirements of different products and architectures can be supported and enforced, rather than suggested and monitored by editorial staff

Structural versus domain specialization

Structural specialization defines new types of structured information, such as new topic types or new map types. Domain specialization creates new markup that can be useful in multiple structural types, such as new kinds of keywords, tables, or lists.

Structural types define structures for modules of information, such as concept or task or reference, which often apply across subject areas (for example, a user interface task and a programming task may both consist of a series of steps). When

new elements are introduced through structural specialization, the elements that contain the new elements must be specialized as well; and the new container elements must have their containers specialized in turn, all the way to the root element for the module (for example, the <topic> element or <map> element).

Domains typically define markup for a particular domain or subject area, such as programming, or hardware. Domain elements become available wherever their ancestor elements are allowed once the domains are integrated with the structural specializations in a document type.

Both structural specialization hierarchies and domain specialization hierarchies are "is a" hierarchies, in object-oriented terms, with each structural type or domain being a subclass of its parent. For example, a specialization of task is still a task; and a specialization of the programming domain is still concerned with programming.

Structural and domain hierarchies must share a common base module in order to be integrated together. For example, domains for use across topic types must ultimately be specialized off of elements in <topic>.

With the exception of the common base module, a domain cannot be specialized from a structural type. For example, a domain cannot be specialized from elements in <task>, only from the root structural modules for <topic> or <map>. This rule ensures that domains can be integrated and document types can be generalized predictably. The rule may be relaxed in future versions of DITA if a mechanism is added for tracking dependencies between structural and domain specializations in use by a document type.

Elements created by specialization are scoped by the name of the structural type or domain in which they were declared. For structural types, the name is the same as the root element: for example, task is the name of the structural type whose root element is <task>. For domains, the name is not shared with any element, but is assigned by the developer of the specialization. By convention, domain names end with "-d" and are kept short; for example, ui-d for the user interface domain and pr-d for the programming domain.

Limits of specialization

There are times when a new structural or domain type appears not to fit into the existing hierarchy, based on the semantics of the existing types and the restrictions of the specialization process. In these cases there are a variety of options to consider.

The basic specialization mechanism used by the DITA document types can also be used for non-DITA document types in order to provide the same re-use, specialization, and interoperation benefits that one can get from the DITA document types, but restricted to the specific domain within which the new document types apply. Note that even if one uses the DITA-defined types as a starting point, any change to those base types not accomplished through specialization defines a completely new document type that has no meaningful or normative relationship to the DITA document types and cannot be considered in any way to be a conforming DITA application. In other words, the use of DITA specialization from non-DITA base types does not produce DITA-compliant document types.

However, given the substantial benefits of building from the common DITA base classes (including the ability to generalize to a common format, use of standards-compliant tools and processes, and reuse of content across document types through DITA maps and conref) there are some techniques to consider before complete departure from the DITA content architecture.

Specialize from generic elements

The first option to consider is to choose more generic base elements from the available set. For example, if you want to create a new kind of list but cannot usefully do so specializing from <ul>, <ol>, <sl>, or <dl>, you can create a new set of list elements by specializing nested <ph> elements. This new list structure will not be semantically tied to the other lists by ancestry, and so will require specialized processing to receive appropriate output styling. However, it will remain a valid DITA specialization, with the standard support for generalization, content referencing, conditional processing, and so forth.

The following base elements in <topic> are generic enough to support almost any structurally valid specialization:

topic any content unit that has a title and associated content

section
any non-nesting division of content within a topic, titled or not

p any non-titled block of content below the section level

fig any titled block of content below the section level

ul, ol, dl, sl, simpletable
any structured block of content that consists of listed items in one or more columns

ph any division of content below the paragraph level

keyword
any non-nesting division of content below the paragraph level

You should always specialize from the semantically closest match whenever possible. When some structural requirement forces you to pick a more general ancestor, please inform the technical committee: over time a richer set of generic elements should become available.

Customized subset document types for authoring

DITA markup is organized into domain and topic type modules so that authoring groups can easily select the markup subset they require by creating a new document type shell. However, when an authoring group requires a subset of markup rules that does not follow the boundaries of the type modules (for example, global removal of certain attributes or elements), you can if necessary create a customized document type for the sake of enforcing these rules at authoring time, as long as the document types are validated using a standards-compliant document type at processing time.

A customized subset document type should be created without editing of the type modules. The document type shell can override entities in the module files, including attributes and content models, by providing a new definition of the entity before importing the module files.

Customized subset document types are not compliant with the DITA standard, and may not be supported by standards-compliant tools. However, customized subset document types can help limit the quantity and mitigate the consequences of non-standard design in a customized implementation.

Map from customized document type to DITA during preprocessing

While specialization can be used to adapt document types for many different authoring purposes, there are some authoring requirements that cannot be met through specialization - particularly splitting or renaming attributes, and simple renaming of elements. In these cases, where the new document type can be straightforwardly and reliably transformed to a standard document type, the authoring group may be best served by a customized document type that is transformed to a standard document type as part of the publishing pipeline. For example, if an authoring group requires additional metadata attributes, and finds authoring multiple metadata axes in one attribute (otherprops) unusable, the document type could be customized to add metadata attributes and then preprocessed to push those values into otherprops before feeding the documents into a standard publishing process.

A customized document type should be created without editing of the type modules. The document type shell can override entities in the module files, including attributes and content models, by providing a new definition of the entity before importing the type module files.

Customized document types are not compliant with the DITA standard, and will not be supported by standards-compliant tools. Preprocessing can ensure compatibility with existing publishing processes, but does not ensure compatibility with DITA-supporting authoring tools or content management systems. However, when an implementation is being heavily customized in any case, a customized document types can help isolate and control the implications of non-standard design in a customized implementation.

Specialization in content

Specialization is expressed in content through the use of two attributes: the class attribute and the domain attribute. These are not typically present in the document instance, but are provided by default values expressed in a DTD or schema.

Why specialization in content?

Specialization attributes let processes and tools know what set of rules your markup conforms to. This allows reuse of tools and processes for unfamiliar markup.

The class attribute

Each element declared in the DITA architecture has a class attribute. This attribute provides a mapping between the element's current name and its more general equivalents. The more specialized the element type, the longer its class attribute value.

For example, the class attribute for the task topic type's step element is:

```
<!ATTLIST step        class  CDATA "- topic/li task/step ">
```

This tells us that the step element is equivalent to the li element in a generic topic. It also tells us that step is equivalent to a step in a task topic, which we already

knew, but it's worth noting this in the attribute because it enables round-trip migration between upper level and lower level types without loss of information. For example, if a user runs a ″generalize″ transform that maps all elements to their first class value, but preserves their content and attribute values, then the user can follow it up with a ″specialize″ transform that maps all elements to their last class value (preserving content and attribute values), and provide a full round trip for all content between the two document types, using nothing but two generic transforms and the information in the class attribute.

The class attribute tells a processor what general classes of elements the current element belongs to. It's something like an architectural forms attribute, except that it contains multiple mappings in a single attribute, instead of one mapping per attribute. Also, DITA scopes values by module type (for example topic type, domain type, or map type) instead of document type, which lets us combine multiple topic types in a single document without complicating transform logic.

Combining the mappings into a single attribute gives us the following benefits:

- preservation of sequence: you can tell by looking at the order of values which one is the most general and which one is the most specific. This is especially important for ″specializing″ transforms, where you can apply a general rule that says: if the element doesn't have a mapping to the target topic type, simply use the last value of the class attribute (and assume that the specialized topic type is reusing some general element declarations, which only have mappings for the level at which they were declared).
- mapping persistence through migration: when you migrate to a higher-level element, you can preserve its more specialized history in the class attribute. If you were declaring a new attribute for each new mapping (as in architectural forms), then when you migrated to the higher-level type the declaration for the mapping attribute would disappear, and roundtripping would be considerably more problematic.

Class attribute syntax

The class attribute has a particular syntax that must be followed for it to be processed correctly.

Every element must have a class attribute. The class attribute starts with a ″-″ if it is declared in a structural module, or a ″+″ if it is declared in a domain module. After the starting token are one or more blank-delimited values, ending with a blank. Each value has two parts: the first part identifies a module package, for example a topic type or domain package name, and the second part (after a /) identifies an element type. Structural names are taken from the root element for the topic type or map type. Domain names are defined in the domain package.

Typically, the class attribute value should be declared as a default attribute value in the DTD or schema rather than directly in the document instance. The class attribute should not be modified by the author.

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

Figure 4. Example structural type element with class attribute

```
<wintitle class="+ topic/keyword ui-d/wintitle ">A specialized keyword</wintitle>
```

Figure 5. Example domain element with class attribute

When the class attribute is declared in the DTD or schema, it must be declared with a default value. In order to support generalization round-tripping (generalizing specialized content into a generic form and then returning it to the specialized form) the default value must not be fixed. This allows the generalization process to overwrite the default values in a general document type with specialized values taken from the document being generalized.

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 structural type or domain in the specialized type's ancestry, even those in which no element renaming occurred. The mapping should start with the value for the base type (for example topic or map), and finish with the current element type.

```
<windowname class="- topic/kwd task/kwd guitask/windowname ">
```

Figure 6. Example attribute with intermediate value

Intermediate values are necessary so that generalizing and specializing transforms can map values simply and accurately. For example, if task/kwd was missing as a value, and a user decided to generalize this guitask up to a task topic, then the transform would have to guess whether to map to kwd (appropriate if task is more general than guitask, which it is) or leave as windowname (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 than the one we are generalizing to, which means the last value in the list is appropriate. For example, when specializing to <task>, if a <p> element has no target value for <task>, we can safely assume that <p> does not specialize from <task> and should not be generalized.

While this example is trivial, more complicated hierarchies (say, five levels deep, with renaming occurring at two and four only) make explicit intermediate values 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, bctask, and guitask 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.

The domains attribute

The domains attribute lists the names of the domains in use by the current document type, and the ancestry for each domain. The domains attribute is declared on the root element for each topic type.

Each domain in use contributes a string in parentheses that gives the names of each ancestor domain plus the name of the contributing domain. Within each set of parentheses, the domain and its ancestry should be listed starting with the most

distant ancestor (the root type off of which the domain hierarchy is based) and finishing with the name of the domain in use.

Example: task with three domains

```
<task id="mytask" class="- topic/topic task/task "
 domains="(topic ui-d) (topic sw-d) (topic pr-d cpp-d)">
...
</task>
```

In this example, the task allows the use of tags for describing user interfaces (ui-d), software (sw-d), and also C++ programming (cpp-d).

Specialization validity

When you specialize one element from another the new element must obey certain rules in order to be a valid specialization.

- The new element must have a content model that is equivalent to or more restrictive than its parent.
- The new element must have attributes that are equivalent to or a subset of the attributes of its parent.
- The new element's attributes must have values or value ranges that are equivalent to or a subset of the parent's attributes' values or value ranges.
- The new element must have a properly formed class attribute.

Generalization

Specialized content can be generalized to any ancestor type. The generalization process can preserve information about the former level of specialization to allow round-tripping between specialized and unspecialized forms of the same content.

The generalization can either be for the purpose of migration (for example, when retiring an unsuccessful specialization) or for temporary round-tripping (for example, when moving content through a process that is not specialization aware and has only been enabled for instances of the base structural type). When generalizing for migration, the class attribute and domains attribute should be absent from the generalized instance document so that the default values in the general DTD or schema will be used. When generalizing for round-tripping, the class attribute and domains attribute should retain the original specialized values in the generalized instance document.

Any DITA document can contain a mix of markup from at least one structural type and zero or more domains. The structural types and domains allowed in a particular document type are defined by the document type shell.

When generalizing the document, the generalizer may choose to leave a structural type or domain as-is, or may choose to generalize that type or domain to any of its ancestors.

The generalizer can supply the source and target for each generalization: for example, generalize from reference to topic. The generalizer can specify multiple targets in one pass: for example, generalize from reference to topic and from ui-d to topic. When the source and target are not supplied, generalization is assumed to be from all structural types to the base (topic or map), and no generalization for domains.

The generalizer can also supply the target document type. When the target document type is not supplied, the generalized document will not contain a DTD or schema reference. At some time in the future it may be possible to automatically generate a document type shell and target document type based on the class and domains attributes in the generalized document.

The generalization process should be able to handle cases where it is given just sources for generalization (in which case the designated source types are generalized to topic or map), just targets for generalization (in which case all descendants of the target are generalized to that target), or both (in which case only the specified descendants of the target are generalized to that target).

For each structural type instance, the generalization process checks whether the structural type instance is a candidate for generalization, or whether it has domains that are candidates for generalization. It is important to be selective about which structural type instances to process: if the process simply generalizes every element based on its class attribute values, an instruction to generalize ″reference″ to ″topic″ could leave an APIReference topic with an invalid content model, since any elements it reuses from ″reference″ would have been renamed to topic-level equivalents.

The class attribute for the root element of the structural type is checked before generalizing structural types:

Target and source	Source unspecified	Source specified
Target unspecified	Generalize this structural type to its base ancestor	Check whether the root element of the topic type matches a specified source; generalize to its base ancestor if it does, otherwise ignore the structural type instance unless it has domains to generalize.
Target specified	Check whether the class attribute contains the target; generalize to the target if it does, otherwise skip the structural type instance unless it has domains to generalize.	If the root element matches a specified source but its class attribute does not contain the target, emit an error message. If the root element matches a specified source and its class attribute does contain the target, generalize to the target. Otherwise ignore the structural type instance unless it has domains to generalize.

The domains attribute for the root element of the structural type is checked before generalizing domains:

Target and source	Source unspecified	Source specified
Target unspecified	Do not generalize domain specializations in this structural type.	Check whether the domains attribute lists the specified domain; proceed with generalization if it does, otherwise ignore the structural type instance unless it is itself a candidate for generalization.
Target specified	Check whether the domains attribute contains the target; generalize to the target if it does, otherwise skip the structural type instance unless it is itself a candidate for generalization.	If the domains attribute matches a specified source but the domain value string does not contain the target, emit an error message. If the domains attribute matches a specified source and the domain value string does contain the target, generalize to the target. Otherwise ignore the structural type instance unless it is itself a candidate for generalization.

For each element in a candidate structural type instance:

Target and source	Source unspecified	Source specified
Target unspecified	If the class attribute starts with "-" (part of a structural type) rename the element to its base ancestor equivalent. Otherwise ignore it.	Check whether the last value of the class attribute matches a specified source; generalize to its base ancestor if it does, otherwise ignore the element.
Target specified	Check whether the class attribute contains the target; rename the element to the value associated with the target if it does contain the target, otherwise ignore the element.	If the last value in the class attribute matches a specified source but the previous values do not include the target, emit an error message. If the last value in the class attribute matches a specified source and the previous values do include the target, rename the element to the value associated with the target. Otherwise ignore the element.

When renaming elements during round-trip generalization, the generalization process should preserve the values of all attributes. When renaming elements during one-way or migration generalization, the process should preserve the values of all attributes except the class and domains attribute, both of which should be supplied by the target document type.

Specialization in design

Specialization in design enables reuse of design elements, just as specialization in content allows reuse of processing rules. These rules involve the creation and management of markup modules as separate reusable units.

Why specialization in design?

Following the rules for specialization design enables reuse of design elements, just as following the rules for specialized content enables reuse of content

By using standard schemes for developing design modules, a specializer enables:

- Reuse of their design modules by others, allowing shared development of specific parts of a document type
- Faster integration of their designs with other specializations, allowing quicker deployment of new design elements and quicker adoption of new markup standards
- Better management of differences between authoring groups in the same organization: each group can create specific document types that integrate just the modules they require.

Modularization and integration of design

Specialization hierarchies are implemented as sets of module files that declare the markup and entities that are unique to each specialization. The modules must be integrated into a document type before they can be used.

The separation of markup into modules, as with the XHTML modularization initiative, (http://www.w3.org/TR/xhtml-modularization/), allows easy reuse of specific parts of the specialization hierarchy, as well as allowing easy extension of the hierarchy (since new modules can be added without affecting existing document types). This makes it easy to assemble design elements from different sources into a single integrated document type.

Integration

Each domain specialization or structural specialization has its own design module. These modules can be combined to create many different document types. The process of creating a new document type from a specific combination of modules is called integration.

Integration is accomplished using a document type shell, which defines the modules to be integrated and how they will be integrated. Integration defines both what topic types and domains will be allowed in the document type, and how the topic types will be allowed to nest.

The module for a specific type should contain only the declarations for elements that are unique to that type, and should not embed any other modules. The shell should contain no markup declarations, and should directly reference all the modules it requires. Nesting shells or nesting modules (having shells that embed other shells, or modules that embed other modules) is discouraged since it adds complexity and may break some tools. Sharing between document types should be accomplished through shared modules, not through direct reference to any other document type. Dependencies between modules should be satisfied by the integrating shell, not through the module itself.

Modularization in DTDs

To support extensibility and pluggability, DITA requires that a DTD implementation of structural and domain specialization modules conform to well-defined design patterns.

This section describes those design patterns. These design patterns realize the specialization architecture with the capabilities and within the limitations of the DTD grammar.

Structural specialization pattern: Each structural type must be defined in a separate DTD module with a name consisting of the topic element name and the mod extension. To see an example, look at the concepts.mod module for the concept topic type.

The structural type module must conform to the following design pattern.

Default element entities

Each element defined in the module must have a corresponding entity whose default value is the name of the element. The following example comes from the definition for the concept topic.

```
<!ENTITY % conbody "conbody">
```

The document type shell can predefine an element entity to add domain specialized elements into every context in which the base element occurs.

Default included domains entity

The module must define the `included-domains` entity with a default empty that is empty as in the following example:

```
<!ENTITY included-domains "">
```

The document type shell can predefine the `included-domains` entity to list domains added to the document type.

Default nested topics entity

Topic type modules must define an info-types entity that is named with a prefix of the topic element name and a suffix of `-info-types.` This entity can default to a list of element entities if the topic has default subordinate topics. If the topic doesn't have default subordinate topics, the entity can default to the value of the `info-types` entity as in the following example:

```
<!ENTITY % concept-info-types "%info-types;">
```

The document type shell can then control how topics are allowed to nest by redefining the*topictype*-info-types entity for each topic type, or quickly create common nesting rules by redefining the main info-types entity.

Structural type's root element content model

As with all specializations, the root element of a structural specialization must have a content model that restricts or conserves the content model of the element it specializes. In addition, for topic types, the last position in the content model must be the nested topics entity as in the following example:

```
<!ELEMENT concept        ((%title;), (%titlealts;)?, (%shortdesc;)?,
        (%prolog;)?, (%conbody;), (%related-links;)?,
        (%concept-info-types;)* )>
```

Attributes

As with all specializations, the root element's attributes must restrict or conserve the attributes of the element it specializes. In particular, the topic

must set the DITAArchVersion attribute to the DITAArchVersion entity and the domains attribute to the included-domains entity.

```
<!ATTLIST concept         id ID #REQUIRED
                          ...
                          DITAArchVersion CDATA #FIXED "&DITAArchVersion;"
                          domains CDATA "&included-domains;"
>
```

These attributes give processes a reliable way to check the architecture version and look up the list of domains available in the document type.

Element and attribute definitions

The module defines every specialized element used as substructure within the topic. The specialized elements must follow the rules of the architecture in defining content models and attributes. Content models must use element entities instead of literal element names.

In particular, the module defines a class attribute for every specialized element. The class attribute must include the value of the class attribute of the base element and append the element name qualified by the topic element name with at least one leading and trailing space. The class attribute for an element introduces by a structural specialization must start with a minus sign.

Domain specialization pattern: Each domain specialization must have two files:

- A DTD entity declaration file with a name consisting of the domain name and the ent extension.
- A DTD definition module with a name consisting of the domain name and the mod extension.

To see an example, look at the highlight-domain.ent and highlight-domain.mod files.

Domain entity declaration file

The domain entity declaration file must conform to the following design pattern:

Element extension entity

The declaration file must define an entity for each element extended by the domain. The contents of the entity must be the list of specialized elements for the extended element. The name of the entity has a prefix of the abbreviation for the domain and an extension of the name of the extended element. In the following example, the highlight domain (abbreviated as hi-d) extends the ph element.

```
<!ENTITY % hi-d-ph "b | u | i | tt | sup | sub">
```

Domain declaration entity

The declaration file must define one entity for the document type shell to register the domain. The name of the entity has a prefix of the abbreviation for the domain and an att extension. The value of the entity must list the dependencies of the domain module in order of dependency from left to right within enclosing parentheses, starting with the topic module and listing domain dependencies using their abbreviations (including the defining domain as the last item in the list). The following example declares the dependency of the highlight domain on the base topic module.

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

Domain definition module

The domain definition module conforms to the following design pattern:

Default element entities
As in a topic module, the domain definition module must declare a default entity for each element defined by the domain so that other domains can extend the elements.

```
<!ENTITY % b "b">
```

Element and attribute definitions
As in a topic module, the domain definition module must define each specialized element and its attributes. As with any specialization, the domain element must restrict the base element. The class attribute of the domain element must start with a plus sign but, otherwise, follows the same rules as the class attribute for an element introduced by a topic specialization.

Document type shell pattern: The document type shell must conform to the following design pattern. To see an example, look at the concepts.dtd module for the concept document type.

Domain entity inclusions
The document type shell starts by including the domain entity declaration files. The entity for the domain declaration consists of the domain name prefix with the `dec` suffix, as in the following example:

```
<!ENTITY % hi-d-dec PUBLIC
    "-//OASIS//ENTITIES DITA Highlight Domain//EN" "highlight-domain.ent">
    %hi-d-dec;
```

Element extension redefinitions
For each element extended by one or more domains, the document type shell redefines the entity for the element to a list of alternatives including the literal name of the element and the element extension entity from each domain that is providing specializations.

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

Topic nesting redefinitions
For each topic type, the document type shell can control nesting of subtopics by redefining the nested topics entity to the literal element name for any of the topics included in the document type. The document type shell can also simply define the `info-types` entity to set the default for most topic types. Here is an example:

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

Domain declaration redefinition
The document type shell redefines the `included-domains` entity to list the domains included in the document type as in the following example:

```
<!ENTITY included-domains
    "&ui-d-att; &hi-d-att; &pr-d-att; &sw-d-att; &ut-d-att;">
```

Structural definition inclusions
The document type shell includes the definitions for the structural type modules used in the document type. The entity for the structural definition consists of the structural type's name with the `type` suffix, as in the following example:

```
<!ENTITY % topic-type PUBLIC
    "-//OASIS//ELEMENTS DITA Topic//EN" "topic.mod">
    %topic-type;
```

Domain definition inclusions

The document type shell includes the domain definitions for the domains used in the document type. The entity for the domain definition consists of the domain name prefix with the `def` suffix, as in the following example:

```
<!ENTITY % hi-d-def PUBLIC
    "-//OASIS//ELEMENTS DITA Highlight Domain//EN" "highlight-domain.mod">
    %hi-d-def;
```

Modularization in schemas

To support extensibility and pluggability, DITA requires that an XML schema implementation of structural and domain specialization modules conform to well-defined design patterns.

This section describes those design patterns. These design patterns realize the specialization architecture with the capabilities and within the limitations of the XML schema grammar.

Structural specialization pattern:

For each structural type, the document type shell document collects the schema documents, parent structural type modules, domain type modules, and content models needed to implement new topic type specializations. Each new structural type requires three files. To see an example, look at the concept_shell.xsd document type shell document for the concept topic type.

1. Each structural type must define a separate module schema document with a name consisting of the root structural element name and _mod.xsd
2. Each structural type must define a separate model group definition schema document with a name consisting of the root structural element name and _grp.xsd

The default values for the `domains` attributes in the base root structural element and the specialized root structural elements must be defined using the XML Schema redefine to populate the `domains` attribute. It identifies the domains used in the structural type. This attribute give processes a reliable way to look up the list of domains available in the document type. The list the domains is included in the document type as in the following example:

```
<xs:redefine schemaLocation="topic_mod.xsd" >
  <xs:complexType name="topic.class">
     <xs:complexContent>
      <xs:extension base="topic.class">
          <xs:attribute
name="domains" type="xs:string" default="(topic ui-d)
(topic hi-d) (topic sw-d) (topic pr-d) (topic ut-d)"/>
       </xs:extension>
     </xs:complexContent>
   </xs:complexType>
</xs:redefine>
```

In the case of topic types, the head schema document can control nesting of subtopics by redefining the nested topics to the literal element names the document type author wishes to allow nested in the document type.

```
<xs:group name="info-types">
   <xs:choice>
    <xs:group ref="concept-info-types"/>
   </xs:choice>
</xs:group>
```

The module schema document must define an info-type model group that is named with a prefix of the topic element name and a suffix of `-info-types.` Here is an example of a info-types model group that is defined in concept_mod.xsd:

```
<xs:group name="concept-info-types">
  <xs:choice>
     <xs:group ref="concept"/>
  </xs:choice>
</xs:group>
```

The module schema document defines every specialized element used as substructure within the structural type. The specialized elements must follow the rules of the architecture in defining content models and attributes. The naming convention for content models must use the root structural element name and.class.

In particular, the module schema document defines a class attribute for every specialized element. The class attribute must include the value of the class attribute of the base element and append the element name qualified by the root structural element name or domain name with at least one leading and trailing space. The class attribute for an element introduced by structural specialization must start with a minus sign.

The model group schema document defines model groups for each new specialized element in a structural type. Each structural type and domain must have a model group schema document . The model group schema document is an essential part of the specialization.

The new file is needed to mimic substitutionGroups in XML Schema without using the inheritance model in W3C XML Schema 1.0 specification. The process is very similar to the DITA DTD design pattern. For a structural type the name of the schema document consists of the root structural element name and _grp.xsd extension. To see an example of a model group schema document, look at the file concept_grp.xsd :

```
 <xs:group name="concept">
    <xs:sequence>
      <xs:element ref="concept"/>
    </xs:sequence>
</xs:group>
```

Domain specialization pattern:

A domain type schema document with a name consisting of the domain name and the -domain.xsd extension.

As in a structural module, the domain module must define each specialized element, its attributes and its model groups. As with any specialization, the domain element must restrict the base element. The class attribute of the domain element must start with a plus sign but, otherwise, follows the same rules as the class attribute for an element introduced by a topic specialization.

For each element extended by one or more domains, the domain type schema document defines a model group for the base element to a list of alternatives including the literal name of the element and the element extension entity from each domain that is providing specializations.

The schema document must define an a model group for each element extended by the domain. The contents of the model group must be the list of specialized

elements for the extended element. The name of the model group has a prefix of the abbreviation for the domain and an extension of the name of the extended element. In the following example, the user interface domain (abbreviated as `ui-d`) extends the `ph` element.

```
<xs:group name="ui-d-ph">
  <xs:choice>
     <xs:element ref="uicontrol" />
     <xs:element ref="menucascade" />
  </xs:choice>
</xs:group>
```

For each element extended by one or more domains, the document type shell redefines the model group for the element to a list of alternatives including the literal name of the element and the element extension entity from each domain that is providing specializations. To integrate a new domain in the document type shell use the schema redefine mechanism to manage the number of domains used by the document type shell. The model group requires a reference to itself to extend the base model group. To see an example, look at the topic.xsd schema document.

```
<xs:group name="pre">
  <xs:choice>
    <xs:group ref="pre" />
    <xs:group ref="pr-d-pre" />
    <xs:group ref="ui-d-pre" />
    <xs:group ref="sw-d-pre" />
  </xs:choice>
</xs:group>
```

To add domains to a new structural type you can copy the contents of the parent structural type domains schema document into the document type shell. Add or remove the model group from the new domain to the appropriate named group.

```
<xs:group name="pre">
 <xs:choice>
   <xs:group ref="pre"/>
   <xs:group ref="pr-d-pre" />
   <xs:group ref="domainName-d-element"/>
 </xs:choice>
</xs:group>
```

Specialization in processing

Specialized processing is not necessary for every specialized element, only for those elements which do not have appropriate default behavior based on their ancestors.

Whether creating a new transform or extending an existing one, there are several rules that should be followed to ensure the effectiveness of the transform for other specialized types, and also the maintainability and extensibility of the transform to accommodate new requirements.

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 change this to instead check the contents of the element's class attribute. Even if the element is unknown to the processor, 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 the rules for that class.

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 both 'task/step' and on 'notatask/stepaway').

Make sure that when you create a transform that targets more than one type that you give the more specific rules a higher precedence to avoid conflicts. For example, when you combine the existing processing rules for topics with more specific processing rules for tasks, use a shell file to import both sets of rules and use import precedence to ensure task-specific rules will not conflict with generic rules for topics.

Example: match statement for list items

```
<xsl:template match="li">
```

becomes

```
<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.

Example: match statement for steps

```
<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.

Modularization and integration of processing

Processing should be divided into modules based on the structural types or domains they support, and can be integrated together into transforms or stylesheets in the same way that structural type and domain modules can be integrated into document types.

Customization

When you just need a difference in output, you can use DITA customization to override the default output without affecting portability or interchange, and without involving specialization.

For example, if your readers are mostly experienced users, you could concentrate on creating many summary tables, and maximizing retrievability; or if you needed to create a brand presence, you could customize the transforms to apply appropriate fonts and indent style, and include some standard graphics and copyright links.

Use customization when you need new output, with no change to the underlying semantics (you aren't saying anything new or meaningful about the content, only its display).

Modularization in CSS

Stylesheet support in CSS for DITA specializations can be applied using the same principles as for the DTDs or Schemas, resulting in stylesheets that are easy to maintain and that will support any subsequent specialization with a minimum of effort.

Specification of module definition

A specialization-aware property for CSS has this form of selector:

```
*[class~="topic\/section"]   {
  margin-top: 12pt;
  display: block;
}
```

The CSS selector that associates the style to the element does not use a literal match to the element name. Instead, based on an element having the defaulted value `class="- topic/section reference/refsyn "` (for example) this rule will trigger on the “topic/section” value (or “word”) and perform the associated styling or transform, regardless of what the actual element name is.

Note that the attribute string must contain an escape character for the “/” character which is otherwise not valid in a CSS selector.

The selector pattern in this example effectively reads, in CSS terminology, ″Selects any element with a class attribute that contains the word topic\/section.″

Not all CSS systems can match based on values that are not physically present in the instance document. Since the class attribute values in DITA are typically provided by default declarations in the DTD or schema, not all CSS systems can match directly on DITA source.

When direct specialization-aware matches are not possible, alternatives include normalization (preprocessing the DITA source to push values from the DTD or schema directly into the instance) or the use of element-name-based rules.

Element-name-based rules will not be specialization-aware. Your calling-stylesheet will have to import each additional stylesheet required by the scope of specialized topics and vocabularies, each explicitly defined using element-name selectors. In this scheme, unsupported new elements will have no rendering properties associated, whereas in the specialization-aware systems such elements can fall back to a rule that triggers off a previously-supported value in the class attribute string.

Assembly rules for CSS

CSS supports specialization similarly to XSLT. This document describes a best practice for naming and populating CSS stylesheets that follow the specialization design pattern for DITA DTDs and Schemas. Although this practice is not required in order to implement CSS support for DITA, following the practice will make subsequent specializations off the pattern to be done with minimal work, and the files should be correspondingly easier to maintain.

To support a newly-specialized DITA DTD or Schema that has been specialization-enabled with unique class attribute values, create a module that will contain ONLY the rules required for the uniquely new elements in the specialization. This is similar to the mod files that declare the unique elements in the specialization. The name of this module should be the same as the root name

for the specialization module. In the case of DITA's reference DTD, the element declarations are in reference.mod and the corresponding delta rules for CSS are in reference.css.

Next, create an "override"CSS stylesheet that starts off with the @import instruction, naming the CSS file used by this specialization's parent DTD. This import picks up support for all elements that are common with the parent DTD. Then add another @import instruction in sequence, naming the CSS delta module that you created previously. Then copy in the CSS rules for any previously defined support that need to be associated to the new element names, and rename the selectors as needed to the new specialized values for each new element. These added CSS rules are deltas for the new stylesheet, much as specialized DTDs build on previous DTDs by adding delta element definitions. This technique approximates the "fall-through" support for what would normally happen if the class attribute actually could map to the root class.

Finally, if necessary, modify the behaviors of any of these new, delta CSS rules. Because this process reuses a great deal of previous behaviors, the time spent supporting the delta changes is minimal.

To use a specialization-enabled CSS stylesheet with a specialized DITA topic, simply associate it to the topic using either the W3C defined stylesheet link processing instruction or by following configuration rules for your editor or browser.

Modularization in XSLT

Stylesheet support in XSLT for DITA specializations can be applied using the same principles as for the DTDs or Schemas, resulting in stylesheets that are easy to maintain and that will support any subsequent specialization with a minimum of effort.

Specification of module definition

A specialization-aware template for XSLT has this form of match pattern:

```
<xsl:template match="*[contains(@class,' topic/section ')]">
  <div>
    <xsl:apply-templates/>
  </div>
</xsl:template>
```

The XSLT match statement that associates the style to the element does not use a literal match to the element name. Instead, based on an element having the defaulted value `class="- topic/section reference/refsyn "` (for example) this rule will trigger on the "topic/section " value (note the required space delimiters in the match string) and perform the associated template actions, regardless of what the actual element name is.

The XPath pattern in this example effectively reads, "Selects any element whose class attribute contains the space-delimited substring "topic/section"."

Assembly rules for XSLT

XSLT pattern matching is the basis for DITA's specialization-aware processing. As such, the base XSLT stylesheet for a DITA topic should minimally support any specialization, no matter how far removed in generations from the archetype topic.

To support a newly-specialized DITA DTD or Schema that has been specialization-enabled with unique class attribute values, create a module that will contain ONLY the templates required for the uniquely new elements in the specialization. This is similar to the mod files that declare the unique elements in the specialization. The name of this module should be the same as the root name for the specialization module. In the case of DITA's reference DTD, the element declarations are in reference.mod and the corresponding delta rules for XSLT are in reference.xsl.

Next, create an "override"XSLT stylesheet that starts off with the xsl:import instruction, naming the XSLT file used by this specialization's parent DTD. This import picks up support for all elements that are common with the parent DTD. Then add another xsl:import instruction in sequence, naming the XSLT delta module that you created previously. Additionally you can add imports for any domain-specific templates that need to be applied with this shell. Then copy in the XSLT templates for any previously defined support that needs to be associated *uniquely* to the new element names, and rename the match pattern strings as needed to the new specialized values for each new element. These added XSLT templates are deltas for the new stylesheet, much as specialized DTDs build on previous DTDs by adding delta element definitions. For XSLT support, you only need to define templates if you need new behavior or if you need to modify the behavior of an ancestor element's processing.

Because this process reuses a great deal of previous behaviors, the time spent supporting the delta changes is minimal.

To use a specialization-enabled XSLT stylesheet with a specialized DITA topic, simply associate it to the topic using either the W3C defined stylesheet link processing instruction or by following configuration rules for your processing tools (usually an XSLT processing utility such as saxon or xsltproc).