Specialization and modularization in DITA

Erik Hennum
IBM
ehennum@us.ibm.com

Michael Priestley
IBM Canada
mpriestl@ca.ibm.com

Dave A. Schell
IBM
dschell@us.ibm.com

Abstract

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 information types and describing new information domains, allowing groups to create very specific, targeted document type definitions using a process called specialization while at the same time reusing common output transforms and design rules.

Specialization provides a way to reconcile the needs for centralized control of major architecture and design with the needs for localized control of group-specific and content-specific guidelines and controls. Specialization allows multiple definitions of content and output to co-exist, related through a hierarchy of information 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. You get the benefit of specific solutions, but you also get the benefit of common standards and shared resources.

Background

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

This architecture and DTD were designed by a cross-company workgroup representing user assistance teams from across IBM. After an initial investigation in late 1999, the workgroup developed the architecture collaboratively during 2000 through postings to a database and weekly teleconferences. The architecture has been placed on IBM's developerWorks Web site as an alternative XML-based documentation system, designed to exploit XML as its encoding format. With the delivery of significant updates in 2002, which contain enhancements for consistency and flexibility, we consider the DITA design to be past its prototype stage.

For more information on DITA, including the base DTDs and sample transforms, see http://www.ibm.com/developerworks/xml/library/x-dita1/ .

At the heart of DITA, representing the generic building block of a topic-oriented information architecture, is an XML document type definition (DTD) called "the topic DTD." The extensible architecture, however, is the defining part of this design for technical information; the topic DTD, or any schema based on it, is just an instantiation of the design principles of the architecture. The consistent use of DTD and XSLT examples in the rest of this paper are meant to show how the principles of DITA are or can be implemented, using DTDs and XSLT, and do not mean that DITA is limited to that implementation choice.

There are three basic ways to extend DITA. Each has a unique role, and associated costs and benefits:

- Specialization
    - Information types
    - Domains
    - Code
- Customization of code
- Integration of design

Specialization

When you require a difference in output that reflects a real difference in input, or you want to make changes to your design for the sake of increased consistency or descriptiveness (regardless of output), you can use DITA specialization to define new information types or new domains.

Specialization allows you to define new kinds of information (new topics, 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.

There are two specialization hierarchies: one for information types (with topic at the root) and one for domains (with elements in topic at their root). Information types define topic 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 information 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.

The two hierarchies are kept separate to make it easy to combine them as needed (for example, to give you a task that contains programming keywords). This means that, aside from their common root in topic, a domain will never specialize elements from an information type, and an information type will never specialize elements from a domain.

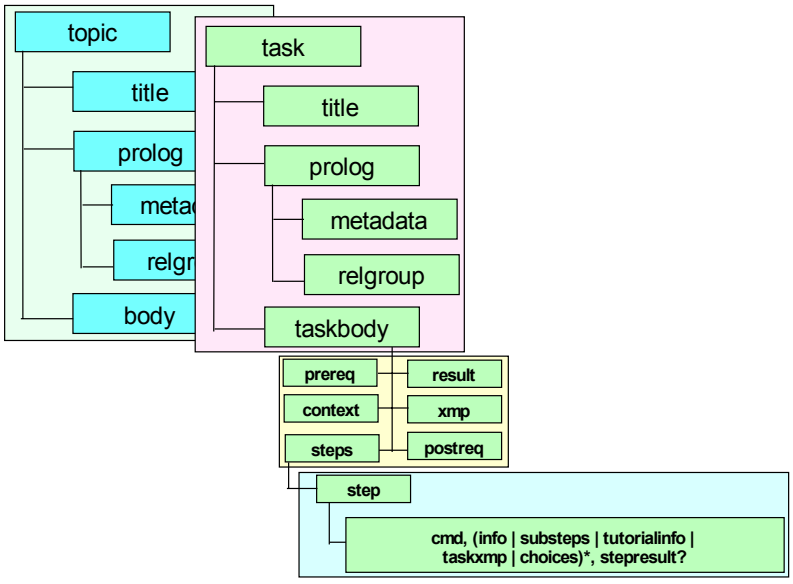
The two hierarchies are implemented as a set of module files that declare the markup and entities required by each specialization. A DTD for authoring specialized content then

embeds the modules for the appropriate specializations, plus the modules for their ancestors. Each of the modules, aside from the base topic.mod, is insufficient for independent authoring, but can be combined with others.

This 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 DTDs). This makes it easy to assemble design elements from different sources into a single integrated DTD.

Specialization involves creating new design modules, and new shell DTD files to embed them. It may also involve creating matching code modules, with new shell XSLT transforms to import them.

When you have semantic distinctions you need to make in your content that are not available in the base DITA framework, or you need to prune the structure of an existing information type to suit more restrictive guidelines, you can create specialized information type or domain modules to incorporate into your design. If appropriate, you can also create matching specialized code modules, to add distinctive output behavior for your new semantic elements.



A specialization can reuse elements from higher-level designs (as task reuses title and prolog), but each specialization module only declares the elements that are unique to it (as task declares taskbody, prereq, context, and so on).

While specialization lets you define new elements, you must map them to pre-existing elements in an existing information type or domain module (as taskbody maps to body, and so on). The mapping must be valid, which means the new element is as restrictive or more restrictive than the parent in its allowed content and attributes, and does not break requirements set by the parent such as required attributes or content. It is encoded in a

special “class” attribute, defined in the DTD as an attribute with a default value, but not actually coded in the content. This lets content in newly specialized information types or using newly specialized domains be processed by pre-existing code, so you can continue to refine your design while preserving your investment in existing infrastructure.

There are two separate ways to specialize:

- New **information types**, which define new kinds of topics, with specific structures as well as specific elements
- New **domains**, which define new kinds of elements (for example new kinds of paragraph, new kinds of phrase, new kinds of keywords) that can be made available in any existing information type, as variants of the ancestor element

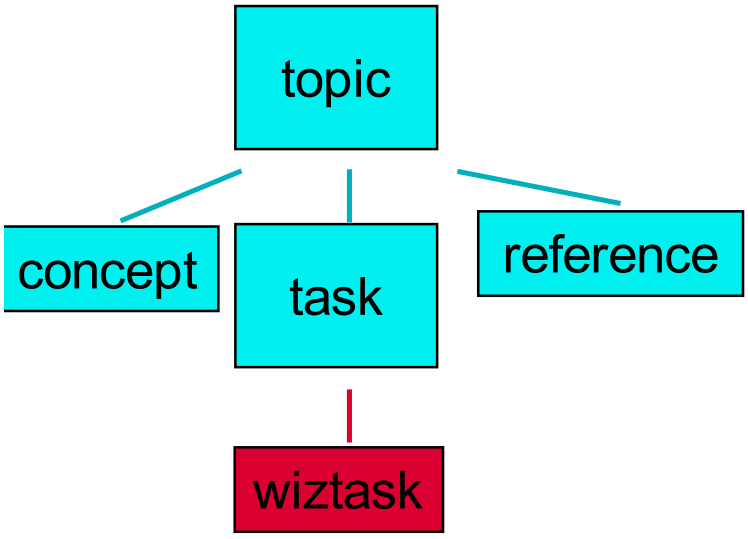
This gives you maximum flexibility in the way you create a specialized DTD:

- **Information type** specialization starts from the top (the definition of the topic) and works down through the structure to whatever level is required (to the section level, or even down to the phrase level, as in the contents of a task’s steps)
- **Domain** specialization starts from the bottom (the definition of an element) and lets you include new variants of that element wherever the original was available

Because you can reuse existing design and code, you don’t need to define an entire DTD from scratch, only the differences between your more descriptive semantics and the already defined semantics in the parent information type or domain modules.

Information types

Information type specialization starts from the definition of a topic, and all other information types ultimately inherit from that.



Each information type’s module contains the declarations for the markup it defines. A shell DTD can then embed the specialized module with its ancestor modules to support authoring topics of the specialized information type.

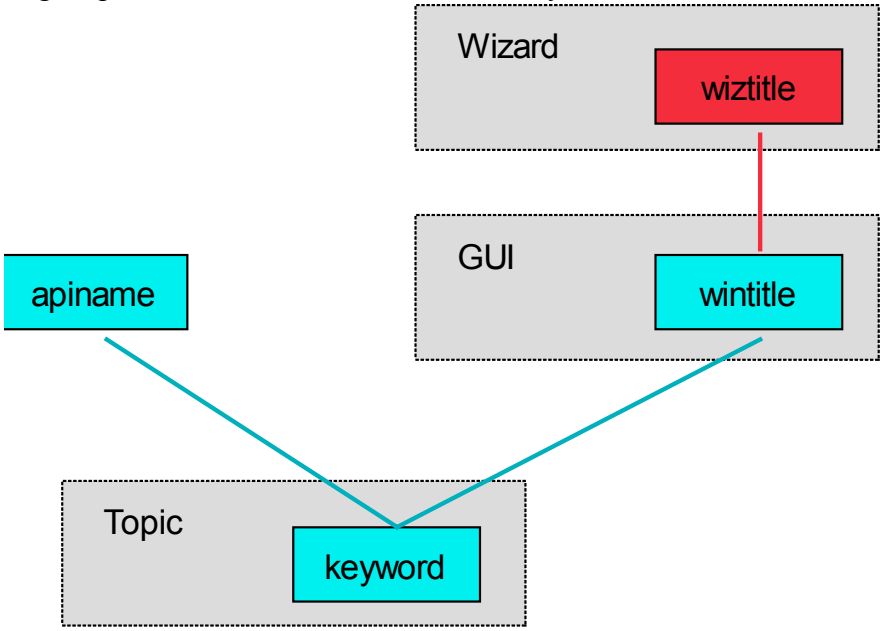
For example, a shell DTD that would support authoring wiztasks could:

- Embed topic.mod (to get default elements from topic, such as p for paragraph)
- Embed task.mod (to get default elements from task, such as cmd for a command in a step)
- Embed wiztask.mod (to get the specialized topic structure for the new topic type, and any new elements declared as part of that structure)

For more information on information type specialization, see:
http://www.ibm.com/developerworks/xml/library/x-dita2/

Domains

Domain specialization also starts from the definition of a topic, although unlike information types, domains can start specializing at any level, that is based on any element in topic, without regard for the elements that contain it. For example, a domain might specialize fifteen new variants of keyword, and touch no other elements.



As with information types, domains must provide mappings from their new elements to ancestral equivalents. For example, a `wiztitle` element in `wizards` could specialize `wintitle` in `UI`, which in turn specializes `keyword` in `topic`: giving `wiztitle` mappings to both `wintitle` and `keyword`.

Each module defines a set of domain-specific elements, such as `syntaxdiagram` (and its component elements) in the programming domain, or `wintitle` (a window title) in the user interface domain. The elements can be quite complex, as in `syntaxdiagram`, which is a specialization of `fig` (a figure in topic) containing eight other specialized elements; or they can be quite simple, as with `wintitle`, which contains only text and is a specialization of `keyword` in topic.

A shell DTD can then embed an entity file (that declares what each element is a variant of) and a module file (to get the specialized markup), and the new domain markup becomes available in whatever information types you are including, wherever the original markup was allowed. In other words, once properly assembled into a DTD, the new markup becomes available wherever its ancestors are allowed. Specializations of `fig` become allowable wherever `fig`s are allowed in the information type; specializations of `keyword`, wherever `keyword`s are allowed in the information type.

To integrate a domain with an information type, create a shell DTD that embeds the domain entities, redeclares content models for the affected elements (for example `fig` and `keyword`), redeclares domain attributes that list the domains in use, and embeds the requisite information type and domain modules, along with those of their ancestors.

For example, if you wanted to include the wizard domain in the concept information type, you would create a shell DTD that:

- Includes the declarations for the domain entities, which define the specialized variants of each ancestor element
- Overrides the definitions for the ancestor element entities, to allow the domain variants into existing content models
- Overrides the content of the domains attribute, so it lists the domains in use by the information type
- Includes the modules for the information types, starting with the least specific (topic), and ending with the most specific (in this case, concept)
- Includes the modules for the domains, again starting with the general and proceeding to the specific

Aside from the modules in which you define the new elements in your domain, all the other modules in the shell DTD already exist. Most of the work for creating a robust document definition has already been done for you.

While the shell DTD is doing considerably more work than it does for information types on their own, note that there is still no markup actually declared in the shell file: all the markup declarations are in the information type and domain modules (.mod files), where they can be reused without conflict by any number of other shell DTDs.

For more information on domain specialization, see:
http://www.ibm.com/developerworks/xml/library/x-dita5/

Code

You may find that the default processing for your new information types or domains is appropriate, and you don't need any new code. For example, the programming domain's `codeblock` element specializes `pre` (equivalent to the HTML `pre` element, meaning preformatted); so `codeblock`, like `pre`, will get formatted with linebreaks and in monospace font, without any extra code necessary.

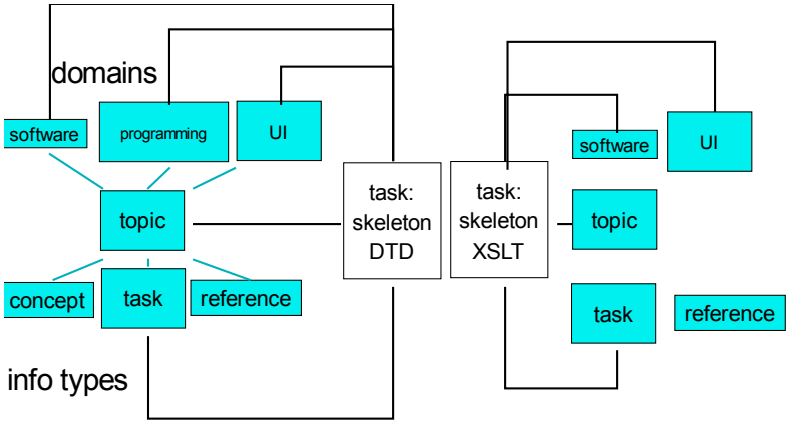
However, if you want different output, you can define the new template rules in code modules that are parallel to the information type and domain modules, so that they can be easily included by a specialized shell XSLT transform, which imports the existing base behavior plus the new overriding rules. Using this approach, you can realize the benefits of the separation of content and presentation by modifying the presentation at will through your own extensions to the base code.

For example, if you wanted to add a special "fastpath" icon to each occurrence of a `wiztitle` in the output, you could create an XSLT module for the `wizards` domain (say, wiz2htm.xsl) that contained a template that matches on `wiztitle` and outputs an icon before the contained text. To incorporate the new rule for the `wizard` domain into an HTML output transform for concepts, you could create a shell XSLT transform that:

- Imports topic2htm.xsl (default behavior, for example pre)
- Imports concept2htm.xsl (concept-specific behavior if any)
- Imports ui2htm.xsl (base behavior for UI-specific elements)
- Imports wiz2htm.xsl (the new domain rule, adding a fastpath icon to wizard titles)

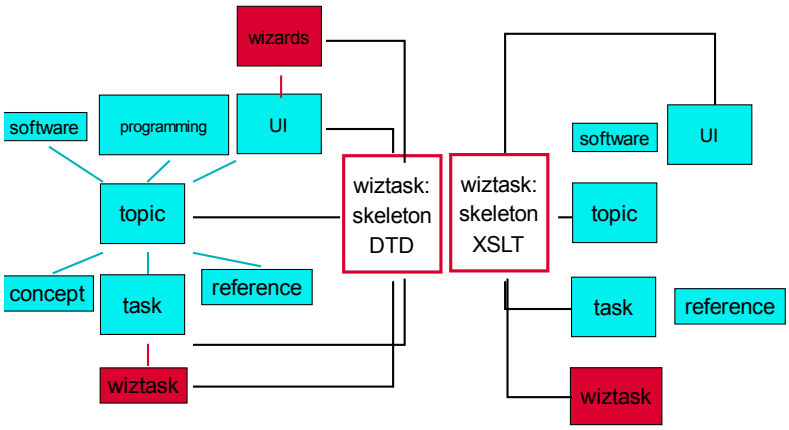
Example: base design for authoring tasks

By designing in modules, and tracking the modules as parts of a specialization hierarchy, we get maximum reuse of markup and code, and maximum maintainability within a formal structure:



Example: extended design for authoring wiztasks

Because the base design is already modularized, extension to the design can easily build on the existing structure, adding modules to the hierarchy and then creating shell DTDs that select the necessary existing modules along with the new ones. Because the new design is also modularized, it in turn is reusable by future extensions.



Specialization and generalization

When content is created with specialized DTDs, it uses new design elements, which could create issues when sharing your content with other groups that don't share the new design elements. Specialization and generalization provide ways to avoid these issues, which would otherwise create substantial barriers to interchange and reuse.

If the reusing group only needs output, they can just run their existing transforms against your content, and get output based on whatever the lowest common denominator is between your specialization hierarchies. For example, if you send them a wiztask, they may process it as a task. This means that other groups can use your content without committing to your output rules or infrastructure: design and output are decoupled, and can be considered, and adopted, separately.

If the reusing group needs to take over the content, however, but is unwilling to adopt the specialization, you can back your content out of the specialization and into an ancestor design, using a process called generalization. This lets you migrate any specialized content into a more general design, taking advantage of the design's built-in mapping, using a standard transform (no need for complex mappings, no need for cleanup). This means that other groups can adopt your content without committing to your design: content and design are decoupled, and can be considered, and adopted, separately.

Result

The result is specialized design both in terms of information type (structure) and domain (subject), with optionally matching specialized output: the markup you need to describe your content for search and enforce consistency of structure, and any output differences you want for your more closely described content.

This gives you the same benefits as a new DTD developed from scratch, but without compromising reuse or interchangeability of content, and with substantially less design and code to create and maintain.

Note that all of these principles and strategies, while demonstrated here with DTDs, can also be implemented with XML schemas, which in fact have some built-in support for validating inheritance relationships that specialization can leverage.

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.

Customization lets you get different output effects without touching your design or content. Your content is insulated from locally-driven design initiatives, such as branding or market-specific requirements, so that if the content gets used by a different brand, or published for new markets, you only need to change customization modules: your base processing model, and all your content, are reusable without editing. This also lets different groups, with different branding requirements, share content without conflict, since their branding requirements are factored out of the content into processes, and even the processes are entirely shared except for brand-specific modules.

Customization involves creating new XSLT modules (that provide the new behavior rules) and new shell XSLT files that import both the existing modules (to provide default behavior) and the new modules (to provide overriding behavior.).

For example, to add a default image and link to every output HTML page, you would need:

- A new customization module that defines the override templates. These may be overrides of existing named templates in the base transforms, or just match-based templates that will fire when they have higher priority than the base ones.
- A new shell XSLT file that imports the existing transform modules, and then imports the custom module (so that it has higher priority than the base modules)

Result

The result is customized output, without affecting the reusability or interchangeability of the content, and with a minimum of new code to maintain.

Integration

Because of DITA's specialization hierarchies, which provide a set of design modules for information types and domains, you can quickly create a DTD that integrates the subset of information types and domains you require, using a shell DTD that embeds the appropriate design modules and leaves the others out.

Integration allows you to select a subset of existing design. You can then use existing default transforms that support all information types, or create a more selective transform that applies only to the design you are using. The result is information that can be processed with existing transforms, and authored with existing DTDs.

DITA is lightweight by design, and specialization is intended to allow you to meet specific needs without increasing the size of the core standard. DITA integration allows you to create an even lighter weight solution, ignoring any branches of the hierarchy that you don't need, even within the base, but also allowing you to selectively integrate additions to the hierarchy, rather than accepting an all-or-nothing proposition. This gives you a "light" version of the DTD on your terms: you get to define what "light" means, what markup you need and what markup you don't, without ever touching the files that hold the markup declarations.

For example, if another group added three information types and three domains to the hierarchies, you could choose to integrate one of the information types and two of the domains, and ignore the rest. This allows you to include the extensions that make sense for your group, without being affected by the extensions that don't apply to you.

When you need a different configuration of existing DITA elements, DITA integration provides a formal, disciplined way to recombine existing information types and domains, without compromising portability or maintainability: since any documents created are subsets of the full supported list of information types and domains, there is no new markup or code to support, and all content created is within supported boundaries.

For example, if you wanted to create documents that consisted of a task topic with child reference topics and support for software and user interface domains (but with no other information types or domains supported, and no other nesting allowed), you could create shells as follows:

A shell DTD that:

- Includes the declarations for the domain entities (software-domain.ent, ui-domain.ent)
- Overrides the definitions for the ancestor element entities, to allow the domain variants into existing content models (pre, keyword, and ph)
- Overrides the nesting entities that define what each information type can nest (task-info-type allows reference, reference-info-type allows no-topic-nesting)
- Overrides the content of the domains attribute, so it lists the domains in use by the information type (sw-d and ui-d)
- Includes the modules for the information types (topic.mod, task.mod, reference.mod)
- Includes the modules for the domains (software-domain.mod, ui-domain.mod)

Note that all of the modules listed in your new shell DTD already exist. That is, you don't have to writing any new modules for integration. You merely plug in the existing modules.

A shell XSLT transform (optional) that:

- Imports topic2htm.xsl (the common root module for topic to HTML transforms)
- Imports task2htm.xsl and ref2htm.xsl
- Imports ui-d2htm.xsl and sw-d2htm.xsl

Result

The result is an integrated design and equivalent output, without any new DTD declarations or transform templates (only shell DTDs and shell transforms that reuse the available existing modules).

Specialization vs. customization vs. integration

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

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).

Use integration when you need to change topic nesting relationships, or restrict the available information types or domains.

Summary

	Artifacts	**Costs/Benefits**
Specialization	Specialized DTD module Shell DTD (Specialized XSLT module) (Shell XSLT)	Small cost New design elements (New code) Migration/interchange supported by architecture (generalization transform) Reuse of most existing design and all or most code
Customization	Customized XSLT module Shell XSLT	Less cost No new design elements Some new code No migration/interchange issues Reuse of all existing design and most code
Integration	Shell DTD	Close-to-zero cost No new design elements No new code

		No migration/interchange issues Reuse of all existing design and code
New design from scratch	Complete DTD Complete XSLT Any migration/interchange transforms when required	High cost New design elements New code Migration/interchange supported by single-purpose transforms; no built-in mappings (transform may be complex, may require cleanup before and after) No reuse of design or code