Specialization in DITA: Technology, Process, & Policy

Michael Priestley
IBM Canada
mpriestl@ca.ibm.com

David 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 coexist, 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.

For some groups, specialization requires a radical move away from centralized processes into a world of negotiated possibilities that introduces many new stakeholders to the information management infrastructure. For other groups, specialization introduces centralization, and, while it provides new opportunities for sharing and reusing logic and design, it also requires new policies and procedures to bring disparate design and development activities into a cohesive, coordinated framework.

Previous papers ([1],[2],[3],[4]) have described in some detail how the technology of specialization works, and how it can be implemented using off-the-shelf tools that are dependent only on base levels of W3C standards (XML 1.0, XSLT 1.0). This paper provides a brief summary of recent changes to DITA specialization, and describes their effects on processes, but concentrates primarily on policy considerations involved in the deployment of a specialization architecture.

Categories & Subject Descriptors: D.2.13 [Reusable Software]; I.7.2 [Document and Text Processing]; K.6.4 [System Management]: Centralization/decentralization

General Terms
Management, Documentation, Design, Standardization, Languages

Keywords
XML, XSLT, XML architectures, specialization, information typing, information architecture, information management, domains, ontologies, process, policy, Darwin, DITA

1 A DITA OVERVIEW

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 was designed by a 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 the IBM®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 choices.

2 THE TECHNOLOGY

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

2.1 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 the 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.

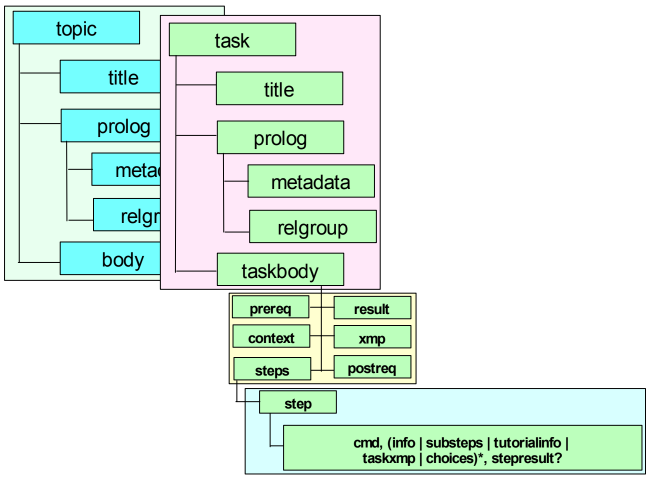
The two hierarchies are kept separate to make it easy to combine them as needed (for example, to give you a task that contains programming keywords). This means that, aside from their common root in topic, a domain will never specialize elements from an information type, and an information type will 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 need to make semantic distinctions in your content that are not available in the base DITA framework, or when 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.

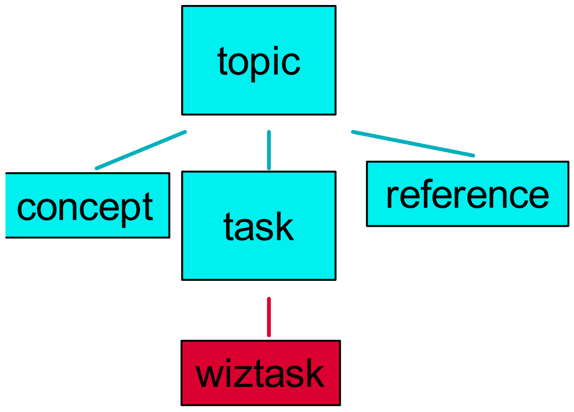
This gives you maximum flexibility in the way you create a specialized DTD:

- **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.

2.1.1 *Information types*

Information type specialization starts from the definition of a topic; all information types ultimately inherit from this topic definition.



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.

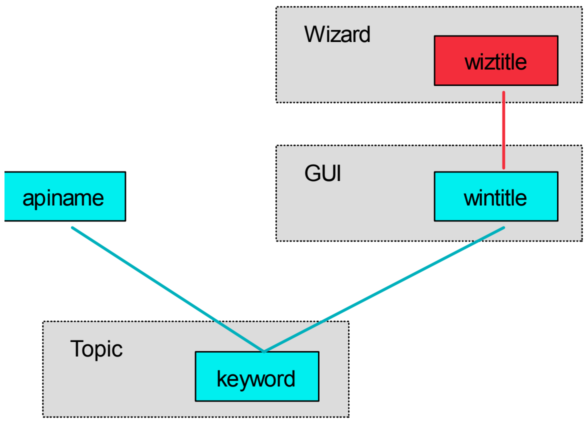
For example, a shell DTD that would support authoring wiztasks could:

- 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/

2.1.2 *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 the 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 (which 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 `figs` are allowed in the information type; specializations of `keyword` become allowable wherever `keywords` 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

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/

2.1.3 *Code*

You may find that the default processing for your new information types or domains is appropriate, and that 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); thus, `codeblock`, like `pre`, will get formatted with line breaks and in monospace font, without any extra code necessary.

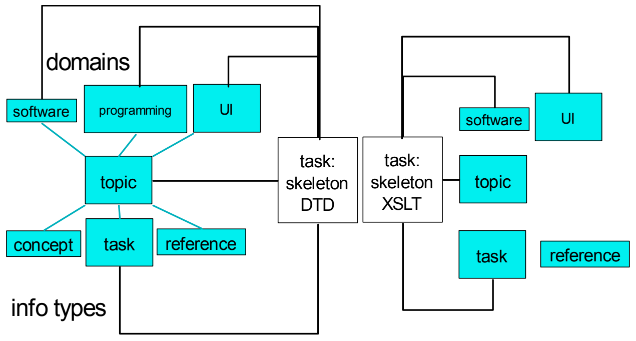
However, if you want different output, you can define the new template rules in code modules that are parallel to the information type and domain modules, so that they can be easily included by a specialized shell XSLT transform, which imports the existing base behavior plus the new overriding rules.

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)

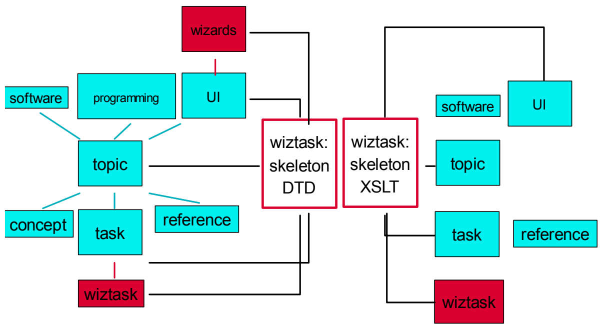
2.1.4 Example: base design for authoring tasks

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:



2.1.5 Example: extended design for authoring wiztasks

Because the base design is already modularized, extensions 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.



2.1.6 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 that 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.

2.1.7 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.

2.2 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 are used whenever 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)

2.2.1 Result

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

2.3 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 more compact 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:

Create 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)

Create 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

2.3.1 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).

2.4 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, that is, when you aren't saying anything new or meaningful about the content, only about how it is displayed.

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

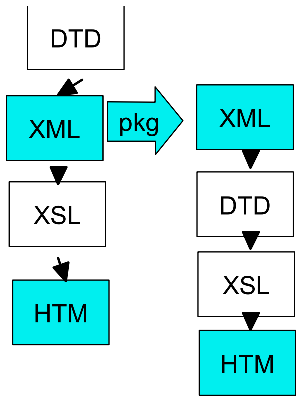
Summary: Technology		
	Artifacts	**Costs/Benefits**
Specialization	Specialized DTD module Shell DTD (Optional) Specialized XSLT module (Optional) Shell XSLT	Small cost New design elements (Optional) New code Migration/interchange supported by architecture (generalization transform) Reuse of most existing design and all or most code
Customization	Customized XSLT module Shell XSLT	Smaller cost No new design elements Some new code No migration/interchange issues Reuse of all existing design and most code
Integration	Shell DTD	Smallest 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 or 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 and may require cleanup before and after) No reuse of design or code

3 THE PROCESS

In this section, we explain how integration, customization, and specialization affect how documents get created, published, and translated.

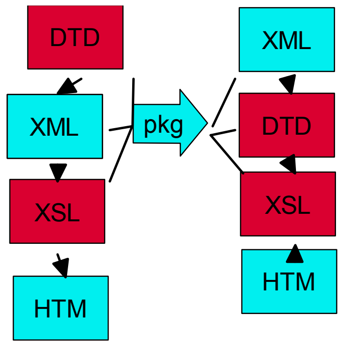
3.1 Standard

In a standard DITA process, authors create documents based on standard DTDs, and transform them using standard transforms to create English output; the documents are packaged for translation and sent to translators, who use translation tools to work with the content, validate in batch mode to make sure their editing has not broken the DTD rules, and then use the same or localized versions of the transforms to create output in various languages.



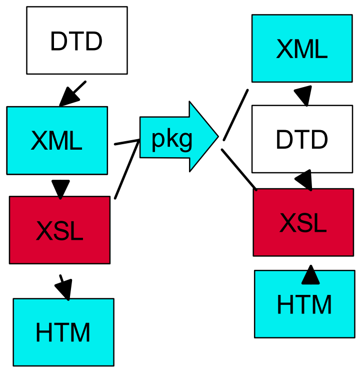
3.2 With specialization

In a specialized DITA process, the specialized DTD modules and shell DTD, and potentially the specialized XSLT modules and shell XSLT, must be packaged and used by the translators to validate the translated content and produce translated output. If the specialized module defines fixed or default text in translatable attributes, those attribute values must be defined in entities in an external file that can be localized; and if the transforms generate translatable text, those portions of the transform must also be isolated and localized.



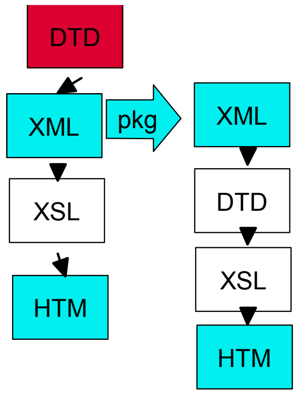
3.3 With customization

In a customized DITA process, the XSLT customization module and the shell transform that incorporates it must be packaged with the content and used by the translators to create the translated output. Various versions of the transform may need to be created; if, for example, the transform generates text into the output, that generated text must be localized for each target language.



3.4 With integration

In an integrated DITA process, there is no impact on translation: the modules selected by the integrated DTD or XSLT are a subset of those supported by the DTDs and XSLT already used by the translators.



4 THE POLICY

In the previous sections, we discussed the technology of DITA and its effects on processes: how to specialize, customize, and integrate DITA to suit your needs, how each process works, and how to identify their costs and benefits in terms of implementation and maintenance.

Having establishing how DITA's technology and processes work in principle, however, one should also address how these processes will work in a real company, with real requirements that go well beyond what's been described in our basic scenarios.

In the following sections, we review some of the issues involved in managing DITA-related projects, in terms of policies:

- Policy issues
- Policy for sharing
- Policy for risks

4.1 Policy issues

Some of the issues related to policy for managing DITA-related projects are:

- Legacy support and archiving
- Investment cost
- Organizational change

4.1.1 Legacy support and archiving

What happens when you create content that needs to be supported for the foreseeable future? When you work with a predictable DTD and transform set, you can easily archive the content and expect to be able to use it any time in the future. However, with integration, customization, and specialization, more than just content is affected: DTD elements and XSLT modules may be affected.

The first part of a legacy support strategy is to differentiate between versions of the DTDs and transforms: current files may safely point to the latest standard DTDs, and use the latest standard transforms, but once archived you will want to make sure content points to the version of the DTDs and transforms that were current at the time of archiving, rather than what will be current some time in the future. While future DTDs are very likely to be backwards-compatible, it makes good business sense to ensure that the content will be validated and processed in the future using the same rules they were originally authored for.

While each group could be responsible for archiving their own content, and potentially their own extensions to the DTDs and transforms, the base DTDs and transforms would be archived on an ongoing basis, using a versioning strategy: each update is reflected in a location that always contains the latest version as well as in a location that reflects the specific version number of the files. This is similar to the way the W3C handles versions of standards, for example.

In addition to archiving content with version-specific DTD references, groups that choose to extend the base process in some way can use additional strategies to manage their archival requirements:

- **Specialization:**
  If you are likely to create new content under the same rules in the future, you may want to archive the specialized modules and shell DTDs (making sure they reference version-specific standard files). Otherwise you can use the standard generalization mechanism to migrate all the content up to a standard DTD version, and archive the content at that level. If you choose to preserve the class attributes defined by the specialized DTDs even after generalization (as actual content rather than DTD-implied values), then you can continue to use XSLT specializations. Those XSLT specializations can be archived in the same way as for customized XSLT. Otherwise, if you are only interested in preserving content, you can completely throw away all your specialization investment, and archive only in terms of the standard (exactly where you'd be if you had stuck to the standard from the start).

- **Customization:**
  If you intend to continue creating the same branded output in the future (and do not just want to archive content for future reuse in other projects), you will want to archive the customized XSLT module, as well as the shell XSLT transform that incorporates it. Make sure the XSLT shell points to the version-specific standard modules.

- **Integration:**
  If you are likely to create new content under the same rules in the future, you may want to archive the integrating DTD as well as the content; otherwise, you can archive just the content, since the DTD remains the same.

To summarize, your archiving choices are as follows:

- **Archive just standard content.** You throw away any additional work you may have invested in integrating, customizing, or specializing.
- **Archive just standard content and output**. You preserve your ability to create specific kinds of output, but throw away the ability to create new content under the same rules as the archived content.
- **Archive the content, the output, and the input rules**. You preserve the content, the ability to create new content under those rules, and the ability to produce specific kinds of output for the content.

4.1.2 Investment costs

Extending DITA provides considerable benefits: eliminating unwanted markup options through integration, creating the output you want through customization, or creating both the markup and output you want through specialization. It can also lower the costs of adoption in many respects. It does this by reducing the tagset to be learned through integration, eliminating format considerations from content through customization, or getting design rules that coach new authors, and markup that describes real content distinctions through specialization. However, any investment beyond the simple authoring of content requires not only additional investment but also a different kind of investment:

- **Specialization:**
  Requires investment in creating the customized DTD modules and shell DTD, and potentially equivalent XSLT modules and shell XSLT. This requires access to skills with DTDs and DITA specialization, as well as to XSLT skills. These costs are best shared with others to avoid duplicating analysis work, and to avoid creating competing descriptions of the same information types or domains. However, sharing development costs does add collaboration costs, and ultimately could result in support costs, depending on who ends up owning the specialization and who ends up using it.
- **Customization:**
  Requires investment in creating the customized XSLT module and shell XSLT file. This requires access to XSLT coding skills. If you ship content to translation, or trade it with other groups, you may want to share the output customization as well as the content, which could put you in the position of supporting the customization when it used by others. However, the other group could reuse your content without using your output customizations, for example, if your customizations conflicted with their branding requirements.
- **Integration:**
  Requires some investment in creating the integrating DTD. This requires access to skills with DTDs. In addition, tools that are DTD-sensitive, such as editors, may need some work to recognize the new DTD as being a subset of the rules already expressed in a supported DTD. Also, if you ship content to translation, or trade it with other groups, you will need to either share the integrating DTD file, or adjust the DTD references in your content to point to the standard DTDs before you ship the content. Adjusting DTD references is easy to do and completely automatable, but still an extra step to incorporate.

4.1.3 Organizational change

DITA's modular design gives many different groups the opportunity to contribute to a common architecture, channeling energy that might otherwise be frustrated by the compromises required by completely centralized design, or marginalized by local design priorities. However, in order to harness this energy, the organization must systematically support new forms of collaboration.

With the definition of the topic, and a base set of DTDs and transforms, we can build in interoperability of content: that is, content created in one part of an organization can be reused elsewhere, with minimal, and automated, preprocessing. In other words, we generalize content to the standard, and adjust DTD references in content to point to the standard. However, sharing content is merely one part of a larger reuse picture: business rules, as captured in integrated and specialized DTDs, and specialized and customized transforms, are also reusable across the corporation, and the requirements for consistency and efficiency make coordination of these activities highly desirable.

For all groups involved, this new coordinating activity represents a redistribution of control: some parts of their design and output will now be controlled by others further up the hierarchy of specialization, and some parts of their design and output may in fact be used by others, reversing the dependency. It means that commitment to a specialization is not only tied to the needs of the maintaining group, but also to the needs of the whole organization: each specialization requires a long-term commitment, one that could potentially outlast the originating group itself.

In other words, by choosing to specialize, a group accepts a dependency on those supporting the higher-level design and output they reuse, and also accepts a duty to those who might choose to specialize from their base, thereby creating a dependency on them. By specializing, a group accepts dual status in both the role of user (of higher-level specializations) and the role of supporter (of their own specializations, which may be higher level, relative to some other group's specialization). This means duties beyond just DTD and XSLT creation, but also documentation, answering questions on internal or external forums, and potentially education and a formal defect and requirement process, depending on the level of formality required by the specialization in question and the organizations involved. The originators of the specialization need to make a long-term commitment to the specialization, including plans for handing off support to other groups, or archiving in a common repository, in cases where the group's interest in the specialization expires before the organization's interest does.

Beyond each group, there must also be a collaboration infrastructure that includes:

- Clearly defined ways to share design, code, and information
- Ways to find collaborators
- Ways to formalize support and archiving strategies

- Ways to track an evolving design, and help groups decide if and when to migrate to new information types or domains as they emerge

For each group, control may be distributed differently, depending on the level of responsibility they are able to assume, and the degree to which their work represents core interests of the organization. There is no longer a single law of governance for all groups, but a series of shifting priorities that draws lines as appropriate for each group, lines that may be redrawn in time as funding priorities and available skills move or evolve. Specializations will often be needed by the organization for the long term, even long after the originating group's interest has expired. As such, support for the specialization, while it may rest with individual groups, still needs to be managed in the interest of all groups, and of the larger organization. Any specialization policy must address such potential conflicts between the requirements and commitments of individual groups, and those of other groups, and of the organization as a whole.

4.2 Policy for sharing

Specialization provides a framework for both more formal and more fruitful collaboration: it allows organizations to reuse and share common elements, without the compromises and overhead associated with the negotiation of a monolithic unified standard. In addition to allowing more degrees of collaboration, it also allows different rewards and different responsibilities.

4.2.1 Sharing design, sharing code, sharing responsibility

When groups share their design, they do more than just reduce costs: they necessarily agree on a common understanding of a set of information domains, and adopt a common way to describe certain kinds of information. Such effort yields not only a more consistent approach, but also a more deeply understood subject. By placing common design efforts in the context of a larger hierarchy, groups participate in the collaborative authoring of an organization-wide ontology: the process of specialization becomes the process of agreeing on what the subjects involved are, and how they relate to each other.

When groups share their code, again they do more than just reduce costs and development time: they also necessarily collaborate on best practices. Each group contributes their best ideas, and over time a consensus on the best approaches to particular problems emerges, and is available to all.

When groups share code and design, they also share responsibility: with regard to the framework that they must work within, and also with regard to those who will build, in the future on the present contributions. The result should be:

- Reduced time to develop by starting from an existing design and code base
- Reduced risk by sharing development costs with others
- Reduced redundancy by eliminating parallel development streams

In some senses, DITA's specialization hierarchies could be viewed as inherently geared toward collaborative design and development: it is well suited to providing the framework for an open-source or shared-source development strategy, with clearly defined dependencies and responsibilities.

4.2.2 Policy implications for sharing

In order for specialization to work in the context of a large organization, we need clear policies on these issues:

- Why to share: why should I specialize?
- When to share: when is it appropriate to specialize?
- With whom to share: with whom can I collaborate?
- How to share: how can I coordinate with these others?
- What to share: what gets produced?
- Where to share: where does it go, to become available to others?

4.2.3 Process implications for sharing

We also need clear policies for how to develop specializations that transcend the technological issues. We need guidelines on collaboration, funding, distributing support costs, determining feasibility, and estimating value to the customer.

4.2.4 Technology implications for sharing

Finally, we need clearly structured repositories to enable people to make use of what's available, register their dependencies on what they use, and contribute to ongoing design and development through feedback or shared development effort.

4.3 Policy for risks in managing DITA-related projects

Specialization is a new way to manage shared information design and code. While it brings substantial rewards without many of the attendant risks of standard serial analysis and development, it also brings a set of unique risks.

4.3.1 Dead-end design

When a design turns out to be bad, or is rendered obsolete, you can save your content by generalizing it to an ancestor information type or domain. You can create a new integrated DTD that supports whatever your target is, or you can generalize all the way up to the base DTD if necessary. This means that, regardless of what you invest in new design or output transforms, your content should be safe.

Note that if you build structures into your design that are not appropriately processed in terms of its general equivalents, you will need to fall back on more traditional non-DITA-based recovery strategies: either you rewrite your content or you figure out a way to get your output through customization alone. For example, the DITA programming domain contains a fairly complex set of markup for describing syntax diagrams. The intended output from the language is a graphic or simple diagram. Without the override processing to create the graphic (for example, displayed using only the default topic2htm.xsl), the diagram appears simply as a list of keywords within a figure that would be readable but not particularly useful.

While we might have been able to get better default processing of syntax diagrams by designing its language differently, we wanted to keep its language as close as possible to the existing design in IBMIDDOC in order to leverage existing processes and skills. In the end, we decided that it was more important to be faithful to the original (pre-DITA) design than to adjust the design to find better mappings (and thereby better default processing). We traded some flexibility in content reusability for the sake of preserving an existing investment in design, education, and processes. However, we are also aware of the ongoing design risk.

To minimize the risk of dead-end design, you can:

- Make sure new specializations map accurately to general equivalents (so that generalization can provide automatic migration to a more general format).
- If some structures require specialized processing (such as `syntaxdiagram`), evaluate the benefit of using the specialization against the risk of future migration; make room in your fallback plan for a custom migration transform to handle special cases.
- Collaborate on the design, to reduce the chances of it being bad.
- Collaborate on the development of code, to reduce the chances of its being useless.
- Collaborate on the support, to reduce the chances of it becoming obsolete.
- Plan for the long term, including considerations for long-term support and eventual archiving.

4.3.2 Ongoing design

Since most specializations will be based on modeling a subject area, their designs will need to change if the subject areas change, or if your understanding of the subject areas evolves. When this happens, you have a few choices: i) you can change your design directly, thus rendering your content invalid until it has been edited into compliance with the new rules, ii) you can create a new information type and migrate to it, thus allowing you to continue working with the old rules and switch to the new rules only when needed, iii) you can loosen your design restrictions, staying with the current information type, but depending on authors and editors to gradually move content into compliance with the new guidelines.

Changing your design “on the fly” may be appropriate for small groups with no immediate deadlines and no dependent groups that would be affected by the change. The advantage is that your design stays clean, only one instance of the information type or domain stays in the hierarchy, your design is as up-to-date as possible, and your content is validated as it is authored.

Changing your design through migration is more appropriate when multiple groups depend on the information type or domain, and will be migrating to the new design at different times. However, this will take some coding. Generalization is free, but respecialization to a new information type will take some custom coding, and may require rewriting.

Simply loosening your design is certainly the easiest option to implement, but has the least satisfying results: your specialization will still support the old (wrong) way of marking up content, and enforcement of the guidelines is no longer automatic. In addition, those looking to reuse your specialization will now have much more to learn before they can successfully use your work (what parts are deprecated, what parts are recommended), and are much more likely to get it wrong, resulting in increased support costs.

4.3.3 External dependencies

You need to manage two kinds of dependency: your group’s dependency on higher-level information types and domains, and other groups’ dependencies on your information types and domains.

To manage your dependencies on others, make sure that you are pointing at version-specific DTD files unless you are confident that your specialized design and code are completely current, and are prepared to do the work to keep them current with future changes to the information types and domains that you depend on.

To manage others’ dependencies on you, make sure you provide version-specific locations for your design and code, as discussed previously. Also, avoid changes that are not backwards compatible unless there is a clear and overriding benefit both for you and for the others that use your work.

4.4 Summary of policy questions

Without policies, a specialization hierarchy could quickly grow out of control, with many different groups each describing the same information types and domains with subtly different, and fundamentally incompatible markup: while their common ancestors might provide a measure of compatibility, the true value of the hierarchy is in its unified descriptions of evolving information types and domains, providing a way to describe a consensus within the larger community about how to write and describe information.

In order to provide order and control the growth of the specialization hierarchy, and maximize its value for the organization, there are a number of questions that must be answered for each proposed specialization.

Roughly speaking, the process each specialization should follow is:

- Determine need
- See if the specialization already exists
- Justify costs with benefits
- Get stakeholders involved
- Design, implement, test, revise
- Get sign-off from management, specialization workgroup, and translation
- Publish, review, revise
- Continue to manage through lifecycle

In more detail, we have categorized the questions you need to answer into three stages:

- Preliminary investigation
- Collaboration
- Proof-of-concept validation

4.4.1 *Preliminary investigation*

In the preliminary investigation stage, the focus is on establishing the basic value of the specialization.

- Does the specialization give value to end-users? For example, does it increase consistency, usability, or searchability? Describe the value at this stage.
- Does the specialization add semantic meaning? That is, does it describe actual content, rather than intended use or output? If the answer is no, consider customization or integration.
- Is the specialization general enough to be useful? That is, is it so specific that no one else will use it?
- Is the specialization specific enough to be meaningful? That is, is it so general that it adds no value?
- Is it stable enough to describe? That is, is it still evolving so rapidly that any time spent defining it will be wasted?
- Is it new? That is, does an equivalent specialization already exists?
- Is it worth continuing? That is, given the amount of value relative to the amount of change required, and the expected scope of its effects, do the potential benefits justify the costs of proceeding?

4.4.2 *Collaboration*

In the collaboration stage, the focus is on sharing responsibilities and achieving a design consensus.

- Was a search conducted for existing standards? Incorporate them where appropriate, and document why you haven't if not appropriate.
- Have you invited other groups to participate, either through co-development or review of the design? Make sure you share costs, and eliminate divergent development streams where possible.
- Have you involved subject matter experts to ensure semantics are meaningful to users? Make sure you have someone thoroughly involved with the subject matter to estimate both the descriptiveness of the design and the usefulness to readers of the distinctions.
- Have you involved affected authors and editors? Make sure the structures you are defining are compatible with good information design principles, and that they support authoring (and customer) requirements and editorial guidelines.
- Is it worth continuing? That is, given the resources committed by all interested parties, can you deliver the specialization in time for the authoring groups that require it?

4.4.3 *Proof-of-concept validation*

In the proof-of-concept validation stage, the focus is on whether the specialization works, and on establishing plans for the future.

- Does the specialization (DTD and XSLT modules combined) work with existing end-to-end processes? That is, are you making any changes that could break existing investments in processes?
- Is the specialization consistent with requirements set by ancestor information types and domains?
- Are the DTDs and code prepared for translation concerns, for example by moving fixed attribute titles and code-generated text into separate files?
- Is the specialization valid, constructed according to module and DTD construction rules, and accurately and appropriately mapped to ancestor information types and domains?
- Is there a mechanism in place for collecting feedback from end-users and authors, to validate value and drive further evolution?
- Is the ongoing support and eventual archiving of the DTDs and transforms accounted for?
- Are fallback plans in place, in case the support agreements prove unsustainable or organizational priorities change?
- Is it worth continuing? That is, given the costs of development, support, and archiving, can you commit to the full deployment of the specialization within the organization, or even outside the organization?

4.5 Policy: costs, benefits, risks, rewards

With this view of policy issues, we can revisit the costs, benefits, risks and rewards of each approach.

4.5.1 *Specialization*

For the wider organization, there is a considerable investment required. This is likely to be an activity requiring collaboration and coordination across multiple groups, and creates a hierarchy of cross-group dependencies that will need to be tracked and managed, as well as a process for selection and validation of the new design and process modules. This should still be substantially cheaper than each group developing from scratch: the collaboration is meant to eliminate redundant effort, not overly complicate a development process.

For the specific group that accepts responsibility for owning a specialization, there is also a considerable investment required. In addition to the development or acquisition of the appropriate skill set (DTDs, XSLT, DITA, information analysis and design, domain familiarity), there are collaboration costs, coding, and testing costs.

The benefit of using specialization is an increasingly descriptive hierarchy of markup, which makes it easy for groups to develop integrated DTDs that meet their needs more specifically, creates a richer repository of information for reuse and development, and gives better value to customers. This means more consistency, lower learning curve for new writers (who have markup that matches the content they need to create), richer information, and happier customers (who have information tailored to their needs, rather than targeted at the lowest common denominator).

The risks of using specialization include specializations that become obsolete (with content that must then be generalized to a higher-level information type or domain); on the other hand, specialization also can require a long-term commitment, since the need to maintain the specialization may actually outlive the original content it was created for. The risks also include cross-group dependencies (both the group's dependencies on others for the maintenance of referenced design and code modules, and others dependencies on the group, for the maintenance of its design and code modules).

4.5.2 Customization

For the wider organization, this has some cost. While customization is likely to be group-specific, it still needs to be reviewed to ensure that organization-wide output guidelines are not broken, and that any generated text is translated.

For the specific group, this has some cost. They need the appropriate skill set (XSLT) and will need to work with translation groups, but there is not much need for cross-group collaboration (except where groups share a brand identity, for example).

The benefit of using customization is customized output without compromise of content: the content remains portable and reusable across the organization, while individual groups still get their specific design they need for the audience or marketplace.

The risk of using customization is that the customization will be rendered obsolete, at which point it can be thrown away with no impact to the content, or won't be rendered obsolete, in which case it will need to be archived indefinitely.

4.5.3 Integration

For the wider organization, this has no cost. The wider organization can concentrate on supporting the base DTDs and processes, which the integrated DTDs select subsets from.

For the specific group, this has some cost. They need the appropriate skill set (DTDs) and may need to customize editors.

The benefit of using integration is immediate enforcement of group-specific guidelines, using a smaller set of markup that is specific to a group's needs. This means a lower learning curve for new authors, and increased consistency in markup and content as a whole.

Summary: Technology and Policy		
	Artifacts	**Costs/Benefits**
Specialization	**Technology**: Specialized DTD module Shell DTD (Optional) Specialized XSLT module (Optional) Shell XSLT **Policy:** Cross-group dependencies	**Technology**: Small cost New design elements (Optional) New code Migration/interchange supported by architecture (generalization transform) Reuse of most existing design and all or most code **Policy:** Costs: Skills, collaborative development, support, archiving Benefits: Reusable hierarchies, lower learning curves, customer-centered content and output
Customization	**Technology**: Customized XSLT module Shell XSLT	**Technology**: Less cost No new design elements Some new code No migration/interchange issues Reuse of all existing design and most code **Policy**: Costs: Skills, support, development, archiving Benefits: Reusable content, customer-centered output
Integration	**Technology**: Shell DTD	**Technology**: Close-to-zero cost No new design elements No new code No migration/interchange issues Reuse of all existing design and code **Policy**: Costs: Skills, support, archiving Benefits: Smaller learning curve, tighter fit to customer requirements

Summary: Technology and Policy (Continued)		
	Artifacts	**Costs/Benefits**
New design from scratch	**Technology**: Complete DTD Complete XSLT Any migration/interchange transforms when required	**Technology**: 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 **Policy**: Costs: Skills, development, support, archiving; must support complete set of costs from start to finish; low likelihood of later reuse Benefits: No dependencies on others, customer-centered content and output

5 SUMMARY

DITA provides three main ways to extend design or output capabilities: specialization (of information types, domains, and code); customization (of output); and integration (of design subsets). Each of these ways must be supported by strategies for technology, process, and policy, which manage the risks to content and to infrastructure posed by uncontrolled mutation of the specialization hierarchies.

Over time, these strategies can be used to manage the evolution of the specialization hierarchies: new information types and domains can be developed to inhabit specific niches in the information ecosystem, evolving as the ecosystem evolves, or preserving their skeletons in the fossil record.

Standard and monolithic DTDs are often unable to move quickly enough to respond to changing conditions, and as a result force their users through mass migrations from one standard to another, often at great expense, and often stranding some content beyond retrieval.

In contrast, DITA provides a diversity that can cushion against change, as well as a set of processing standards that support evolution without forced migration. While some specializations will always end up extinct, generalization can save the content, and the decoupling of design and process hierarchies can save the infrastructure from the fate of the design. In effect, instead of a series of mass extinction events, DITA provides a more rational way to evolve: pursuing some branches and pruning others, as the needs of your information environment dictate.

6 REFERENCES

[1] Priestley, Michael. Specializing topic types in DITA. http://www.ibm.com/developerworks/xml/library/x-dita2/

[2] Hennum, Erik. Specializing domains in DITA. http://www.ibm.com/developerworks/xml/library/x-dita5/

[3] Priestley, M., Hargis, G., and Carpenter, S. (2001) DITA: An XML-based Technical Documentation Authoring and Publishing Architecture. Technical Communication, Technical Communication, Volume 48, No.3, p.352--367.

[4] Schell, D.A., Priestley, M., Day, D.R., Hunt, J. Status and directions of XML in technical documentation in IBM: DITA. Conference proceedings, Make IT Easy 2001 http://www.ibm.com/ibm/easy/eou_ext.nsf/Publish/1819

7 RESOURCES

Main developerWorks site:
http://www.ibm.com/developerworks/xml/library/x-dita1/

DITA DTDs and transforms:
http://www.ibm.com/developerworks/xml/library/x-dita1/dita10.zip

DITA FAQ:
http://www.ibm.com/developerworks/xml/library/x-dita3/

DITA forum:
news://news.software.ibm.com/ibm.software.developerworks.xml.dita

8 TRADEMARKS