DITA XML: A Reuse by Reference Architecture for Technical Documentation

Michael Priestley
IBM Canada
mpriestl@ca.ibm.com

ABSTRACT

The Darwin Information Typing Architecture is an XML architecture for producing and reusing technical information. DITA promises the following:

- Scalable reuse, so you can reuse content in any number of delivery contexts simultaneously without complicating the source
- Descriptive markup, so you can use markup that describes your information in terms your customers need
- Interchangeability, so you can treat specialized markup as if it were general, getting reuse of tools and processes defined at more general levels of descriptiveness
- Process inheritance, so you can reuse existing process logic in your specialized processes.

It accomplishes these goals by applying the principle of reuse by reference to the dimensions of content, design, and process within a technical communications workflow.

1. BACKGROUND

For the past two years, a workgroup inside IBM's User Technology community has been working on creating XML architecture for the next generation of technical documentation. It was released for public review and testing in March of 2001, and is continuing to evolve with the input of a growing community of writers and developers.

The Darwin Information Typing Architecture (DITA) is an XML-based architecture for authoring, producing, and delivering technical information. DITA is an end-to-end architecture. It consists of a set of design principles for creating information-typed topic modules and for using that content in various ways, such as online help and product-support portals on the Web. At its heart, DITA is an XML document type definition (DTD) that expresses many of these design principles. The architecture, however, is the defining part of this proposal for technical information; the DTD, or any schema based on it, is just an instantiation of the design principles of the architecture.

SIGDOC'01, October 21-24, 2001, Santa Fe, New Mexico, USA.

2. DITA PRINCIPLES

DITA simplifies the creation of audience-specific content, DTDs, and processes. It is based on principles of modularity and reuse that allow not only the fast deployment of customer solutions but also the painless evolution of those solutions as customer needs, and our understanding of them, evolves.

2.1 Four principles

DITA's basic principles are as follows:

2.1.1 Topic orientation

DITA focuses on the topic as the smallest independently maintainable unit of reuse. This allows authors to focus on writing topics that efficiently and completely cover a particular subject, or answer a particular question, without dwelling on the various places the topic might end up being read.

2.1.2 Information typing

DITA focuses on information types as a way to describe content independent of how that content is delivered. Instead of creating chapters and appendixes, authors can focus on writing concepts, tasks, and reference topics using structures and semantics that remain valid regardless of how the information reaches the reader.

2.1.3 Specialization

DITA allows authors to create more specialized information types, so that the structures and semantics of the information are as specific as they need to be for a particular audience

2.1.4 Inheritable processes

DITA-aware processes, such as publishing and translation, work automatically on more specialized types, and can also be specialized themselves.

2.2 Embodied in architectures

Those principles are embodied in two architectures:

2.2.1 Information architecture:

The information architecture describes what a topic is and what the three core information types are. This provides a basic level of consistency across all DITA content, which allows for reuse of infrastructure and interchange of content across the entire range of possible information types.

2.2.2 Specialization architecture:

The specialization architecture describes how a specialized type of topic is derived from a more general type of topic, and it describes how specialization-aware processes can access topics at whatever level of specialization they require. For example, a

generic print process may treat all types as just topics, and a task indexing system may treat all task types as just tasks.

2.3 Affecting content, design, and process

Both architectures are based around reuse by reference, in the dimensions of content, design, and process:

2.3.1 Content reuse

The information architecture, by dividing content into topics and freeing it of delivery-specific elements, allows content to be reused in multiple contexts without being rewritten. This means that many contexts can reuse the same topics without the reused topics being affected. By contrast, large documents that are conditionally processed to produce multiple outputs quickly run into problems of scalability: with each new output, new conditions must be added to the source, and ultimately the source becomes unmaintainable. Reuse by reference, on the other hand, makes sure that the source stays maintainable, by moving control of the reuse into the reusing context, keeping the source simple.

2.3.2 Design and process reuse

The specialization architecture, by creating a hierarchy of information types, allows reuse by reference both of design and of process.

When authors create a specialized type, they can reuse the design of a more general type by reference. This means that the new type can be created faster and requires less maintenance than it would if it were created from scratch. And since the new types can be processed just as if they were the more general type, content in the new types can be deployed immediately, giving customers immediate access to the more specific content categories and structures. For example, when a new API description topic type is created as a specialization of reftopic (reference topic type), the new markup can be displayed using existing reftopic processes, but searched using API-specific terms.

When programmers create a specialized process, they can reuse the code from a more general process by reference. This means that the new process can be created faster than if there were no model and requires less maintenance than if it were created from scratch. And since the new process reuses the more general process, it can even handle less specialized information, as well as more specialized information. For example, if a programmer wants to add keyword linking to API declaration sections, they can override the behavior for those sections with a new XSLT template, but continue to reuse the rest of the reftopic display logic for other sections and elements.

Reuse by reference lets DITA solutions be deployed quickly, maintained centrally, and improved cheaply.

2.4 Gives us the results our customers need

As a result, we can afford to provide customer-specific solutions (delivering the content our customers need, sorted and categorized according to their priorities) without compromising portability or flexibility: meeting the needs of users today, but not at the cost of meeting their needs tomorrow.

Architectures	Principles	Affecting
Information architecture	Topics	Content
	Information types	Design
Specialization architecture	Inheritable design	
	Inheritable process	Process

Summary: four principles embodied in two architectures affecting content, design, and process

3. CONTENT REUSE

When information (such as concepts, tasks, and reference topics) are assembled or aggregated into new contexts, there are three things that can make the process easier:

- Consistently chunked information (all the units you are assembling are the same size, or of a predictable size)
- Context-free information, that focuses on a single task, idea, or thing, with as few external references or dependencies as possible
- Automatic inclusion as part of a repeating process, so that there is only one copy of the source, and when it gets changed all the places that reuse the source pick up the change without manual intervention

The alternative (manually scanning through rambling documents, then cutting and pasting the applicable content into the new document you are creating) is time-intensive, error-prone, and non-scalable in terms of maintenance. Every time you copy information, you create a new place it must be maintained. In other words, reusing the information once doubles the cost of maintaining it; reusing it twice triples it, and so on.

In contrast, automatic inclusion does not increase the cost of maintenance. Every time you change the original, all contexts pick up the change. However, it puts considerable pressure on the writer to keep the reusable content as free from context as possible.

When you include content automatically, there are two standard ways to pick which content to include:

- As a property of the source that marks it as a candidate for certain kinds of inclusion
- As a reference from a context document or navigation map, that points specifically to the content it wants to include

The advantage of property-based selection is that you don't have to add or maintain a specific reference to the topic from anywhere else: the point of inclusion only has to identify the criteria for inclusion, it doesn't have to exhaustively list each of the information units that meets that criteria.

The disadvantage of property-based selection is that sometimes the properties themselves don't provide enough information, and you are then faced with the task of updating the properties of every unit you hope to reuse. For example, if you had set properties on 1000 units that identified them as applying to either novice, experienced, or expert users, and then realize that you need to subdivide those categories further - say into database administrator, Java programmer, and C++ programmer – then you are stuck with updating 1000 files to reflect the new properties for each. For example, where a task before applied to

expert users, you can now clarify that it is an advanced task for programmers but a simple novice task for database administrators.

The advantage of context or map-based references is that you can update the criteria for selection without affecting the units you are selecting. So in the previous example, instead of updating 1000 files to reflect the proliferation of properties, you simply expand or even split the one original map (which distinguished between three types of user) so that it makes the necessary distinctions (for example, one map for each user role). Adding new properties, as new needs arise, does not affect the units being reused, and therefore does not affect others reusing the same units.

The disadvantage of context or map-based references is that they can be mind-numbingly literal to maintain. For example, if you wanted to include the documentation for 1000 C++ classes in a reference manual, it seems pointless to maintain a list of those classes that must be updated each time a new class gets generated, when you could simply be including based on whether the class has public or private as its setting.

In practice, there are times when either approach is reasonable. For example, when you want to include based on a very stable property (such as one defined by a programming language standard), that's pretty safe. But if you want to include based on something more volatile (such as audience analysis), you may be better off maintaining an explicit list or map instead of properties on each unit.

Theoretically, properties and maps define the same kinds of information, and one can be transformed into the other as needed. For example, a map can be derived from properties on a set of units, and properties can be set based on listings in a set of maps. For many types of properties, it may be most appropriate to maintain the information in map form (which is the most maintainable, and keeps the topics as free from context as possible) and then process the map to set properties in each topic as they are published for a particular delivery context.

3.1 The continuum of reusability

DITA allows topics to be authored together, as a nested hierarchical structure, with relationships among different topics, metadata, and various other features that, strictly speaking, tie the topic to a particular context. However, the structure of a topic works to compartmentalize the contextual features, and preserve the reusable elements for easy access.

Topics have the following high-level structure:

```
<topic>
  <title>…</title>
  <prolog>…</prolog>
  <body>…</body>
  <topic>..</topic>
  <topic>…</topic>
<topic>
```

That is, a title, prolog, and body, followed by any number of nested topics.

The contents of the prolog (largely, metadata and relationships), and the content after the body (nested topics) are context: they embed the topic in a structure, they make assumptions about the existence of other topics, and potentially about the product, platform, and audience to which the topic applies. All of this matter is subject to change if the topic is reused in another context: some topics may no longer be available or applicable, the product and platform may have changed, and the definition of the audience may need refining.

A context-free topic, created for maximal reuse, may only allow this simple structure:

```
<topic>
  <title>…</title>
  <body>…</body>
</topic>
```

In other words, no prolog (with its relationships and metadata) and no nested topics.

This topic can then be combined with information stored for a particular context in the form of a separate document - a context or navigation map – which provides the necessary structures and data to populate the prolog and, if necessary, assemble topics into larger compound documents.

For any changes that need to be made within the body of a topic, you can set a variety of properties on any element within the body. These properties, including audience, platform, and product, allow more traditional filtering methods to be applied to the content of a topic, excluding elements when their properties flag them as inappropriate for a context.

The more you can depend on maps, and on reuse at the topic level, the more scalable your reuse is. It is often better to add things in (by applying maps) then to filter things out (based on properties). Given that both maps and properties are specific to particular contexts, you can add new contexts with maps simply by defining new maps; but adding new contexts with properties requires editing each of the affected topics: the separation of content from context is compromised.

The simple topic, with only title and body and without use of the various context attributes, represents the most reusable form of DITA content. However, it is at one end of a continuum: if you are authoring topics for a more constrained environment, in which the opportunities for reuse are well-understood in advance, it may very well be appropriate to sacrifice some of these principles on the altar of pragmatism, and create a more context-rich topic, adding related links, nesting structures, and metadata as part of the topic itself, rather than separated out into a separate context or navigation map.

The way that DITA compartmentalizes its content, with the body holding all the reusable content of a topic, allows you to easily revisit your reuse strategy at various points in your documentation lifecycle. For example, it may be appropriate to author your topics as a single document of nested topics when you first begin your project: getting information out fast, in a single context, may be enough of a goal for a first draft, or a beta, or even a version 1. But in later releases or drafts, as maintenance becomes more of an issue and your documentation potentially begins to spin off into different versions for variations of product, audience, platform, or other issue, you can choose to release your topics from their authoring contexts, and refactor them into individual topic documents that are related only by the maps that reference them.

4. DESIGN REUSE

DITA allows two kinds of design reuse: aggregation (in which the design for a compound document is assembled out of the type modules for each of the topic types allowed) and specialization (in which a new topic type is created with reference to an existing topic type).

4.1 Aggregating modules

The first kind of reuse is based on the use of DTD modules, somewhat after the pattern of XHTML and others. In order to create a customized DTD, all you need to do is include the modules you want, leaving out the ones you don't want.

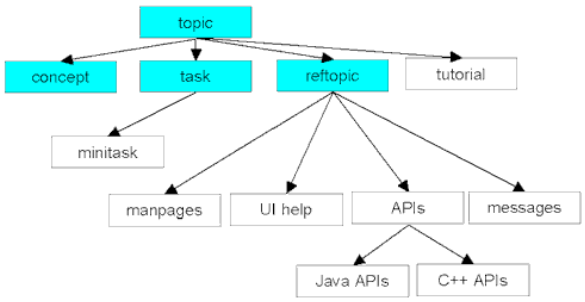
One of the advantages of modularized customization, as opposed to filter-based customization (in which you pick particular sections of the DTD to ignore), is that it allows you to define and include new modules as needed, without editing the base DTD file to add a new section. Indeed, modules may be incorporated from many different sources, to provide different sets of functional coverage. One of the inherent limitations of both approaches, however, is that you need to define your reusable units at a useful level. One of the reasons we did not directly incorporate an XHTML module into DITA, for example, was because every module it defined contained information we didn't want as well as information we did want: at some point the complexity of maintaining the reference outweighed the savings of reusing a module.

We chose to define modules at the topic level, and provide starter DTDs for each of the main topic types: the base topic, plus concept, task, and reference topic. Each of these DTDs can be used to author single-topic documents, or you can use the DITABase DTD to author mixed-type fully nested topic structures, in other words something less like a web and more like a book. To create your own aggregate DTD, you can simply include the modules you want for the topics you are concerned with, and use an entity redeclaration to define the nesting you want to allow.

4.2 Specializing topic types

The second kind of design reuse is unique to DITA. Specialization allows you to create a module for a new topic type that defines only the new elements required for that type: any reusable elements from ancestor types do not need to be redeclared.

DITA starts off with a type hierarchy of four topic types: a generic topic type, and three specialized topic types: concept, task, and reftopic (for reference topics). You can add new types based on any of these types, and also further specialize your own types to create whole branches of information typing:



Specialized topic types consist of a series of "deltas", the declarations for new specialized elements that describe and constrain your content more closely.

There are several rules to follow when you create a specialized element:

- Make sure the places you want to use it are specialized too. For example, if you want to define a new kind of section, you'll need to define a new kind of body too; and once you've defined a new kind of body, you'll need to define a new kind of topic element as well.
- Make sure the new element maps to an existing element in its type hierarchy. You can't just define new elements at random.
- Make sure the new element allows the same content, or a subset of the same content, as its more general equivalent. I other words, the new element can have a tighter or more restrictive content model, but not a looser or less constrained one.
- Make sure the new element has the same attributes as its general equivalent, or at least the required ones.
- Make sure you define a default value for the element's spec attribute that gives mappings for each of its ancestor types.

These rules are very important in making sure that your new type not only reuses design elements from general types, but also can be used to create specialized content that is compatible with general transforms, as explained in the next section.

Once you've defined the new module, you can create an authoring DTD by including the module, along with the modules for each of its ancestors.

Listing of an API description authoring DTD:

```
<!--Redefine the infotype entity to exclude other topic types-->
<!ENTITY % info-types "APIdesc">
<!--Embed topic dtd to get generic elements -->
<!ENTITY % topic_type SYSTEM "topic.dtd">
%topic_type;
<!--Embed reftopic module to get more specific elements -->
<!ENTITY % reftopic_type SYSTEM "reftopic.mod">
%reftopic_type;
<!--Embed APIdesc to get most specific elements -->
<!ENTITY % APIdesc_type SYSTEM "APIdesc.mod">
%APIdesc_type;
```

5. PROCESS REUSE

While it is beyond the scope of this paper to discuss in detail the processing model for DITA content, the main points in terms of reuse are as follows:

- specialized content (authored in specialized DTDs) can still be processed by transforms that have not been customized in any way: you can create a specialized DTD without losing any access to existing infrastructure

- specialized transforms (when the general transforms are insufficient) can and should reuse as much of the general transform logic as possible, overriding only the behavior that needs to be defined differently.

Some basic rules for defining specialized transforms apply, in much the same way that rules apply for defining specialized DTDs:

- Create one module per topic type, so you can cleanly assemble modules to create transforms for compound document types
- Create import cascades of modules to handle specialized topic types, so you can reuse general logic as appropriate, and also have a fallback for handling unknown topic types.

For more information on the spec attribute, and on creating specialized transforms, see the DITA web site on developerWorks.

6. INTERACTION OF DESIGN AND PROCESS

The net effect of the two type hierarchies: the topic type hierarchy, and its mirror, the transform import hierarchy, is to implement an object-oriented inheritance hierarchy, in which the data and behavior have been cleanly separated.

This allows any branch of the process hierarchy to handle content defined along any branch of the transform hierarchy: the import cascade provides transforms that will trigger at whatever the best match is, even if the content is unknown.

For example, if a specialized process defined for publishing C++ documentation is fed some Java documentation, it will fail to recognize the Java tags, but the reftopic logic it imports will recognize the spec attribute mappings to reftopic, and still manage to publish the document as a generic reference topic.

For transforms that are specific to a particular branch of the hierarchy (for example, a transform that emitted C++ header files from the documentation), obviously there can be no fallback, but it can in turn provide the base for further specialization and provide a fallback for those processes.

7. SUMMARY

DITA's reuse architecture accommodates a continuum of content reuse, from maximally reusable simplified topics through book-like richly annotated nested topic structures. On the design and process, it supports reuse of constraints and markup in a type hierarchy, and reuse of typed content in a type-aware transform hierarchy, adapting object-oriented inheritance to the data- and document-centric world of XML and technical documentation.

The result is an end-to-end architecture that lets you create specialized document types, with the markup and consistency appropriate to your users' domain, without losing access to wider realms of transforms, processes, and interchange.

This paper is not a complete description of the DITA architecture: it focuses on the reuse aspect, especially with regards to the reuse of content in topics and maps. For more information on DITA, please see the resources at the end of this paper.

8. WEB RESOURCES

Introduction to DITA:
http://www.ibm.com/developerworks/xml/library/x-dita1

Specialization in DITA:
http://www.ibm.com/developerworks/xml/library/x-dita2

Discussion forum:
http://www.ibm.com/developerworks/xml/library/x-dita4

DTDs and samples:
http://www.ibm.com/developerworks/xml/library/x-dita1/dita00.zip

9. RELATED PAPERS

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

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