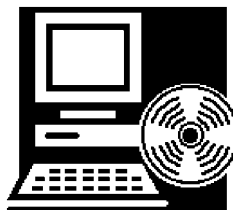


Deborah S. Ray
and Eric J. Ray
Editors



DITA: An XML-based Technical Documentation Authoring and Publishing Architecture

Michael Priestley, Gretchen Hargis, and Susan Carpenter

This column examines emerging technologies of interest to technical communicators to help them identify those that are worthy of further investigation. It is intended neither as an endorsement of any technology or product, nor as a recommendation to purchase. The opinions expressed by the column editors are their own and do not represent the views of the Society for Technical Communication. All URLs and site contents were verified at the time of writing.

The Darwin Information Typing Architecture (DITA) is a technical documentation authoring and publishing architecture that is based on principles of modular reuse and extensibility. This article discusses how DITA affects how we write, how we design, and how we process technical documentation, and what benefits the DITA approach can deliver that traditional documentation strategies cannot.

Over the past few years, XML (Extensible Markup Language) has gained popularity in the technical writing profession by offering us a

logical and fairly straightforward framework for developing structured information. For technical communicators, XML promises capabilities to separate form from content; to use specific, customized markup to describe content; and to use a standard solution without depending on proprietary tools or formats. The promised result of XML is documentation that is reusable in any medium, useful for specialized tools and for our customers, and interchangeable without depending on a particular authoring environment.

XML in and of itself, however, has not, to date, been a panacea in our quest to achieve these goals; instead, we often still struggle to develop processes that realize the potential of XML. In this article, we introduce the Darwin Information Typing Architecture (DITA), which provides technical communicators with an XML-based architecture for authoring, producing, and delivering technical information.

As you'll see, DITA goes further than other currently available solutions by allowing us to easily create highly specialized structure and content, yet still retain interchangeability

and reuse of the content and process. As a result, DITA helps solve current problems in information development, including those of information reuse and information delivery in multiple media (single-sourcing), and helps us maximize the potential of XML for technical communicators.

In the following sections, we provide a brief overview of terms and concepts; describe the promise of XML and its shortcomings; and describe how DITA addresses content-, design-, and process-related problems.

A BRIEF INTRODUCTION TO MARKUP LANGUAGES AND XML

If you are already familiar with XML DTDs and XSLT as used for documentation, you can skip to the next section. Otherwise, read on for a brief introduction to the principles of markup languages in general, and XML and related standards in particular.

A *markup language* is a set of start and end tags you can use to "mark up" text with additional information about your content—for example, `<xmp>` the xmp tag set tells processes that this text is part of an example `</xmp>`. This information can be used for

- ◆ Displaying the text, to apply different fonts and styles to different types of information
- ◆ Processing the text, to extract particular subsets of the information for particular uses
- ◆ Searching the text for particular kinds of information

XML is a standard for defining markup languages. XHTML (Extensible HTML) is an example of an XML-compliant markup language, as are WML (Wireless Markup Language) and DocBook. XML is a streamlined version of SGML (Standard Generalized Markup Language), an older and broader standard for defining markup languages.

While this article will concentrate on XML, most of the issues noted here, as well as most of the solution, could be applied using SGML.

An XML *document type definition* (DTD) is a file that defines the allowable markup and markup rules for a particular markup language. For example, XHTML's DTD defines `<head>` and `<body>` elements, and says that you can't put `<head>` after `<body>`. The core of the XML standard is really just a set of rules and guidelines for creating DTDs.

An XML *document*, for the purposes of this article, is any marked up document that points to an XML DTD to describe what markup rules it is following.

A *stylesheet* (Cascading Stylesheet [CSS] or Extensible Style Language [XSL]) is a mapping of XML elements (tag sets) to display properties. For example, a stylesheet could say that `<xmp>` content should be displayed using the Courier font.

An *XSLT transform* is a mapping of one XML structure to another XML structure or to plain text. This allows you to transform your XML source into an HTML or Acrobat PDF file, or create summaries and indexes and links automatically.

Figure 1 shows how the different parts of an XML solution fit together.

In this article, we describe how information architecture can follow this same basic scheme, but rework the elements into a system designed to support more reuse, faster deployment, and easier maintenance.

THE PROMISE OF XML

XML has been touted as the philosopher's stone of technical writing—able to turn documentation into gold with the application of three simple principles:

- ◆ Separate form from content, so you can reuse the same content with different presentation media. For example, the `<example>` tag describes content,

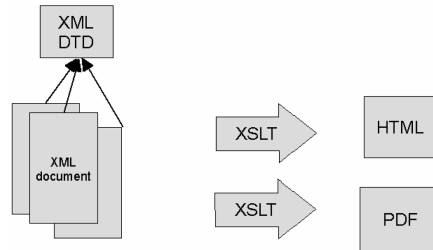


Figure 1. XML documents created following the rules in a DTD can be processed with XSLT transforms to create different outputs.

while `` describes form (presentation).

- ◆ Use specific markup to describe your content, so you can apply presentation styles intelligently, mine information for later reuse, and allow intelligent online search. For example, if your markup distinguishes a task `<step>`, then users can search for the text string *address*, wherever it occurs in a task step, to retrieve a short list of tasks they might have to repeat when their address changes. While the same results could be retrieved with simple text search, the good hits could be buried in thousands of irrelevant matches.
- ◆ Use a standard solution, so your information can be easily exchanged with others, without dependencies on particular authoring tools or proprietary formats. For example, WML for wireless applications and XHTML for Web information provide these exchange abilities.

The promised result is documentation that is reusable in any media, useful for specialized tools and for your customers, and interchangeable without dependencies on a particular authoring environment or proprietary format.

THE REALITY GAP

The problem with the three simple principles is that they're not that simple. XML documentation solutions, like SGML solutions before them, can be expensive to implement and maintain; and promises are not always what they seem.

The problem of form versus content

Although this is a much-hyped and fundamentally accurate separation, the fact is that "form" isn't just about fonts and page breaks; it is also about structure. You can take the font information out of a book, but if you leave the content in a foreword-chapter-appendix structure, it's still going to look like a book when it goes online. Many of the most popular or commonly used documentation DTDs today are actually *book* DTDs or *helpset* DTDs. They mix presentation structures with content-specific markup, until the best you can hope for is a book that looks nice online, or a helpset that prints nicely from a browser.

The problem with specific markup

A heavy-duty documentation DTD can take years to develop (and usually does) and is heavily dependent on your environment, your needs, and your special cases. Once you have your specific markup, you still have to develop those intelligent applications and data mining tools that take advantage of your markup. And what happens when your needs change, or your understanding of the information evolves and your markup doesn't support your new requirements?

In the world of software documentation in particular, where whole categories of information appear and disappear within the span of a year, a heavy upfront investment in analysis and tools can be untenable, especially if those categories will evolve

and change and require rewriting of the tools and processes you spent so much time developing.

The existing model of DTD development seems geared for heavy upfront investment and long-term payoffs; but these assumptions ring false in the world of technical communication, where technologies can't wait to be documented, and analyses have a useful half-life of a year or two at most, not decades. So despite the promise, many people using XML or SGML for documentation simply pick a general DTD, use existing tools, and forget about the promise of specific markup.

The problem with standard solutions

Although XML is a standard, most of the markup languages created with it are not. If you use one of the standards (for example, XHTML or WML), you usually get something that is not content specific (`` rather than `<step>`). In other words, when you create a new markup language (using XML to define its markup and rules), you shut yourself off from interchange with the rest of the world; when you adopt a standard markup language, you lose the benefits promised by content-specific markup.

If you want useful markup (that is, markup specific to your content), you need to spend a lot of time on upfront analysis and need to create custom tools. Existing solutions often have formatting structures embedded in them, and those formatting structures create a customization cost even when they are specific enough in their markup. And the more widely supported a markup language is, the less likely it is to be useful for your specific content.

That's the tradeoff. The more useful your markup is to you, the more it will cost you, and the fewer people there will be to share those costs.

THE SOLUTION: DARWIN INFORMATION TYPING ARCHITECTURE

It is possible to get the benefits without the tradeoffs, but this reward requires some serious thinking about what your content is, how you categorize it, and what you do with it.

We propose the XML-based Darwin Information Typing Architecture (DITA) as an end-to-end architecture for creating and delivering modular technical information. This architecture consists of a set of design principles for creating information-typed topic modules and for using that content in various ways, such as on-line help and product support portals on the Web. At its heart, DITA is an XML 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 implementation of the design principles of the architecture.

As you'll see in the following sections, this solution has three parts:

- ◆ Fix the content—Authors need to rethink how to write content to thoroughly separate form from content.
- ◆ Fix the design—Architects need to rethink how to classify and design information, to reduce the cost of upfront analysis and ongoing maintenance for content-specific markup.
- ◆ Fix the process—Programmers need to rethink how to create transforms and processes, to allow content-specific information to be exchanged, and to make it easier to create and maintain specialized processes.

You can implement these concepts by using the DITA DTDs, XSLT transforms and other supporting information available on the Web, and any XML and XSLT tools at your disposal, including a variety of freely

available or open-source implementations. In the next sections, we'll discuss strategies for each of these three areas—fixing content, design, and process—and explain how DITA supports these strategies.

FIX THE CONTENT

How can technical writers write so that the content is reusable and not oriented toward a particular presentation, such as a book? One aspect of the solution to the reuse problem is to fix the content. Writers must see beyond a linear or sequential presentation when they write the content.

Ditch the book as the basic structure

Many technical writers start a writing project by forming an outline. They use containers such as parts and chapters, and drill down on each major content area, subdividing some subjects into at least two other subjects.

Arranging information in a linear structure usually proves both challenging and awkward. For example, the first chapter in a book might set the stage for the whole book and not just for the next chapter. The first subject in a chapter might set the stage for the whole chapter and not just for the next subject. When writing in a linear structure, writers must force subjects into order, even though some subjects could fit in different places. Writers often pick the order that flows best for most of the material and that requires the least repetition and cross-referencing. Writers might add advance organizers, overviews, summaries, or some combination of these features to help clarify the flow and make sense of the order for users.

Information written for a linear structure tends to explicitly receive the strand of meaning from the preceding subject and pass the strand to the next one. This type of information also often refers to more distant subjects within the same linear structure. These

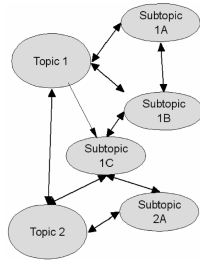


Figure 2. A network of subjects shows connections among them.

transitions and cross-references mean that a subject is not independent. Writers have a hard time rearranging the material for different presentations, and users who come into this structure somewhere other than at the beginning can be confused.

A network of subjects, as shown in Figure 2, is less artificial and rigid than a linear structure. Such a network consists of nodes (for the subjects) and connections, which show relationships between the nodes. In a network there is obviously more than one path from start to finish, and more than one place to start or to finish. Although less tidy, a network more closely resembles, according to educational theory, the way that people actually store knowledge. This representation is related to concept maps, semantic networks, and knowledge maps, which might add labels (such as *creates* or *causes*) to the connections to clarify the relationships.

The network also more closely resembles the way that users actually retrieve the information. The place at which a user enters the document can easily be different for different users with different needs.

Write chunks of information

An alternative to forcing a linear structure on inherently nonlinear information is to write chunks (or topics) of information. Each chunk treats a specific subject, signified by its title.

In this approach, each chunk is a

discrete entity. It can stand alone because it does not explicitly indicate that users arrived there by a particular route or that users will read a particular chunk of information next. The chunk is untethered—that is, it has no sentence or paragraph at the start to tie it backward and no sentence or paragraph at the end to tie it forward. It has no lead-in such as “As you saw in the last chapter” or references to parts of other topics such as “in Figure 10 of Chapter 4.”

The subject of a chunk should be small enough that it can be treated within a few paragraphs, which is an amount that a user can reasonably read online without having to scroll more than a single screen. A subject that would be appropriate for a chapter in a book is too long for a topic. The subject should also be large enough that users are likely to find the information that they need. One sentence or even one paragraph is probably too short. A subject that is at the third or fourth level of a traditional outline is probably appropriate in length.

So a chunk (or topic) has these characteristics:

- ◆ One subject, signified by the title
- ◆ Wording that is independent of any other topic
- ◆ Appropriate length to treat the topic adequately yet not require lots of scrolling

Topics, then, can be freely combined as appropriate for various needs. For example, topics A, B, and C could be used in a task-oriented presentation or as B, C, and A in a different task-oriented presentation. With the addition of overviews as topics, topics can then be collected again in even a book format, as shown in Figure 3.

Although a topic is not locked into a particular sequence, it does have relationships to other topics. These relationships are based on organizations that a writer might want to im-

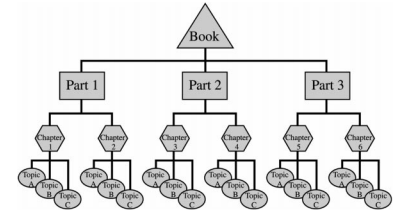


Figure 3. A book can be a collection of topics in a hierarchy.

pose, such chronology, frequency, comparison, or priority. Some, if not most, relationships can be stored in the delivery context and thus be accessible to readers. For example, if a writer is developing online help, the relationships can exist as a page with links—in the appropriate order—that refer to the topics, thus providing any alternative structure that the writer wants to deliver.

Categorize topics by type of information

When writing software manuals, technical writers have historically distinguished task-based instructions (guidance information) from reference information. The content and organization of a user's guide, for example, is different from the content and organization of a reference manual. By writing topics, however, you can make finer-grained distinctions about the type of information that a user can expect than just at the level of the whole document.

A topic can contain task information, concept information, or reference information. Each of these (task, concept, reference) is a specialized type of information and shares the characteristics of a topic, but each type has its own distinguishing features (Figure 4).

Task (or procedure) information contains steps describing how to do something. The instructions typically take the form of a numbered list with an imperative sentence for each list item. Additional information for a

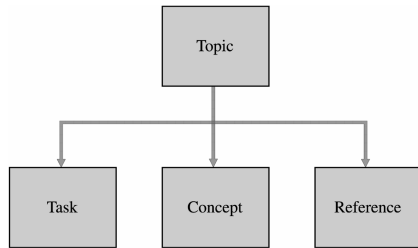


Figure 4. A task, concept, or reference is a specific type of information, each of which also shares the characteristics of a topic.

task might include:

- ◆ Rationale for the task: why or when a user would want to perform this task
- ◆ Prerequisites for the task: what a user should do before performing this task
- ◆ Responses to the actions: what the user should see as a result of doing the action
- ◆ Examples: examples of what information to enter or what to do
- ◆ Postrequisites for this task: what to do next after this task is completed

A task is usually the right size for a topic. If a task runs into tens of instructions, it should be broken down into meaningful parts that a user can reasonably accomplish without feeling overwhelmed.

The second main kind of information is a *concept*. A concept is an extended definition of a major abstraction such as a process or function. It might also include an example and a graphic, but otherwise there are not many parts to a concept.

The last major type of information is *reference*, which is factual in nature. It includes properties and syntax. The breakdown of reference information is often prescribed by conventions, such as those used in documenting a programming language.

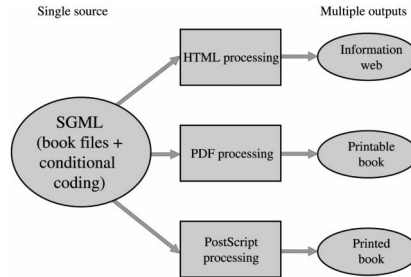


Figure 5. Processing book files as the single source for multiple outputs.

These types of information are media-neutral. That is, an information type can be presented in any form. By contrast, a chapter, part, or appendix makes sense only as part of a book. Topics made up of these information types can be assembled into various forms, including helpsets, information webs, or books.

Provide multiple outputs to multiple contexts

Many technical writers have tried to use their book files as a single source for multiple outputs such as a user's guide (in PostScript or PDF) and an information web (in HTML), as shown in Figure 5. Although it might be relatively easy to specify a file format to export to, a different file format is usually not all that's needed. The optimal organization of an information web, for example, is different from that of a book. Similarly, only certain files or parts of files might be needed for certain outputs, reflecting the specialized types of audience or product.

The means for achieving multiple outputs from a single source has historically been tortuous for writers:

- ◆ Conditional coding to control what's included and what isn't for the type of output and for the type of audience or product
- ◆ Conversion tools (or transforms) to produce the right output in the right format (such as HTML in frames for a web and PDF for

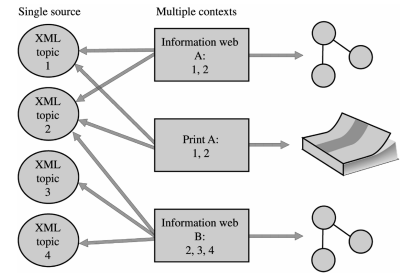


Figure 6. Processing XML topics as the single source for multiple contexts.

a printable book)

The conditional coding (usually supported by homegrown tools) is stored with the content and generates more and more programming requirements to untangle the right pieces for the right output. For example, for a product that is available on more than one operating system, a particular function might have some variations by platform; documenting the right variation for the output by platform can require conditional coding even of sentences and parts of sentences. With each new output, the source becomes more complex and more difficult to maintain.

Topics (written according to the guidelines in this article) offer the possibility of a single source that writers can combine in many different delivery contexts. By listing the appropriate files, you can collect the necessary topics for a particular deliverable. So instead of having a single large book that you process twice to produce two outputs, you have a single collection of topics with two different sets of selection criteria. The same task topic, for example, can be part of several different information webs and several different printable or printed documents, as shown in Figure 6.

Topics free you from having to concentrate on the output or the medium as you write. Instead, you can concentrate on writing good chunked

information on specific topics based on appropriate information types. The reuser (such as an information architect, information designer, or even a program) is responsible for assembling the topics to fit the need.

A business procedure for preparing an expense report shows several of the parts of a task topic, as in Figure 7.

What do chunking and information typing gain you

In summary, chunking and information typing have several benefits for writers and for users. As a writer, you benefit in these ways:

- ◆ You can write stand-alone chunks of typed information once rather than the same information several times in different formats.
- ◆ You can provide for new reuse contexts without having to change the source.
- ◆ You can update the single source, and have the updates picked up automatically wherever the source is used. This technique is called reuse by reference.

Similarly, users benefit in these ways:

- ◆ Users get information that is clearly organized as tasks, concepts, or reference, and the information is delivered in chunks that they can easily use.
- ◆ Users can take advantage of information that is appropriate to them and their situation (assuming that presentation methods can mediate more effectively between the XML and the user than in the past).
- ◆ Users can get up-to-date information sooner because less time is needed to produce it.

Information-typed topics support reuse by default. You would have to break the information type architecture to thwart reuse. Furthermore, adding a reuse context does not increase the overall complexity of the information. The information stays simple, at least from each perspective.

Using this information-typed topic as a basis for writing content can fix the content problem and separate form from content; however, to realize the full potential of XML and implement the DITA proposal, the design and processes must also be fixed. The following section describes how to address the information design.

FIX THE DESIGN

How can we make new types of information easier to define? Architects don't have years to spend on DTD design, especially not a design that will require multiple revisions over the following years, with each revision requiring more investment in tools and processes. We need a way to define new types of information more quickly and more effectively, so we can get the benefits of content-specific markup for our authors

and our users. But we need to do so in a way that still allows interchange with other groups, and reuse of common tools and processes.

Easy typing

DITA provides the basic structure for documents corresponding to the three information types discussed earlier in this article (tasks, reference, and concepts). However, you will still probably need more specific or more specialized markup to address characteristics of your tasks, reference, or concepts that are unique to your needs or context. To do this, you will still need to create a "new" information type, but DITA gives you a head start.

DITA doesn't eliminate all the work required to create a new document type. You still need to create a DTD, and before you do that, you

Preparing an expense report

Before you start, have the following handy:

- Itinerary from the travel agent
- Airplane ticket stubs or e-ticket confirmation number
- Meal receipts totaling more than \$25 total
- Transportation receipts (car rental contract or taxi receipt)

- | | |
|---|---------|
| 1. If you traveled internationally, find the exchange rates for the days that you traveled at http://sample.only.com/travel . This information will be requested later. | Step |
| 2. Using the GETPAID accounting tool, enter information about your trip. GETPAID is available from Jane Doe in IT | Step |
| a. On the itinerary, find the trip reference number. Type it in the Trip number field and press Enter. | Substep |
| This action transfers itinerary information into the form. | Result |
| b. On subsequent pages, enter additional information as requested. | Substep |
| c. On the last page, click Finish. | Substep |
| The tool displays a summary report and saves it to the hard disk as an HTML file. | Result |
| 3. Using a Web browser, submit the report. Your browser must support Java 1.2.2. | Step |
| a. Open the newly created HTML file and verify the information | Substep |
| b. If all is correct, click Submit. Otherwise, return to the GETPAID step. | Substep |
| This action submits the report to the accounting system and requests a paper copy from your default printer. | Result |
| 4. Send the paper copy with receipts to John Doe in Accounting | Step |

If you do not receive a confirmation within two weeks, contact John Doe at (800) 555-1212. Postrequisite

Figure 7. A business procedure shows several characteristic parts of a task topic.

still need to analyze your information and figure out what markup the DTD should allow. But DITA does make it easier to define DTDs.

- ◆ First, you can concentrate on topic types rather than entire document architectures. The rules within a single topic are a lot easier to analyze than the rules that might apply across an entire book or web.
- ◆ Second, using a technique DITA calls *specialization*, you can create new topic types relative to an existing topic type: so instead of defining all the markup you'll need and all the rules that apply, you need only to define the *delta*: the new elements and rules you require.

A complete documentation DTD can have thousands of elements. In DITA, we make do with about a hundred, most of them already familiar from HTML: simple elements like `<p>` for a paragraph and `` for a list item. Then we *specialize* these to create more specific elements for tasks, concepts, or reference topics. For example, in a task, `` becomes `<step>` or `<substep>` or `<choice>` (all variants of a list item), and rules state where each of those elements is allowed inside a task.

Taking the next step and defining a new type of task (*specializing* task) might require five new elements. Once you know which elements you need, a new DTD file can be written in about ten lines (two for each element). That's less analysis than would be required for a complete document type analysis because you're looking only at single topics, and it's less analysis than would be required to create a topic type from scratch because you're reusing a hundred-odd elements from the existing definitions of what a topic is and what a task is. Figure 8 shows how a specialized task relates to the task information type discussed above.

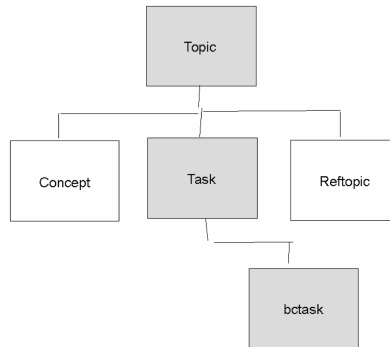


Figure 8. Specialization of the information type *Task* creates a new kind of task, called *bctask*.

Low-cost design

You now have a way to define new types of information more quickly and easily by restricting your analysis to the topic level, and by reusing existing design work that works at more general levels. This strategy reduces the cost of entry for content-specific markup, which makes it easier for you to deliver advanced functions like intelligent search to your users. But it can also reduce the cost of maintenance: because the new task type reuses the more general designs *by reference*, it automatically picks up enhancements made to the more general types that it specializes. Specialization also lets you manage groups of related types by making changes to their common ancestor, instead of maintaining each instance separately.

Specialization can also reduce the cost of changing a type: as we'll discuss later, processes you create to work with a specific type will automatically work with its child types. So if you need to enhance a type to meet new requirements, you can simply create a new specialized type based on the existing one, and start using it immediately: all your tools and processes will continue working without any extra maintenance.

If this scheme sounds like the inheritance feature in object-oriented systems, that's because it's very simi-

lar. In both cases, you're reusing a general design to create more specific designs, using a hierarchy with general types at the base and specific types at the ends of the branches. So when an ancestor type gets edited, its descendant types pick up the changes automatically. The key difference is that while inheritance lets you add new kinds of information in child classes, specialization only lets you more closely define the kinds of information you already have.

For example, in inheritance terms, there would be nothing wrong with adding an extra level of steps to a new task type (steps, substeps, subsubsteps, and so on). But in specialization terms, since such structures aren't allowed in the parent type, they can't be allowed in the child type. The content that you allow in a child type must be the same as or a subset of the content that you allow in the parent type.

This restriction is important because it vastly widens your reuse community. You can share your content not only with your immediate community (those who need the same specific markup that you do) but also with a larger community of people (or processes) who care only about your content's more general features. This larger community can use your content because they have a guarantee that what you're providing is a subset of what they already know how to handle.

For example, here is one branch of the hierarchy showing the specialization levels for a customized business process task:

- ◆ Topic—anyone can share content and all processes
- ◆ Tasks—people who care about how-do-I information and task-specific processes
- ◆ Business-control tasks—people who care about business tasks and their processes

- ◆ My business tasks—people within my particular company or group, and our specific processes

The further down the hierarchy you are, the more specific your design and the more useful your content can be to you. But being far down the hierarchy doesn't lock anyone out: people with more general concerns can access your information as the ancestor type that makes sense to them. They don't lose access to your information, and you don't lose access to their tools and processes.

The way specialization works

We'll discuss later how to make tools and processes that can be reused on different types. For now, let's look at how DITA supports specialization and how you can create new types based on DITA using specialization.

DITA distinguishes between DTD files and DTD modules. DTD files provide all the information you need to create a topic of a particular type. DTD modules store only snippets of DTDs that contain the unique markup for a particular type. Each type gets its own module, which defines the markup that is unique to that type. That's all you need when you're defining the type (which keeps the file short), but it's not enough for your authors. They need a real authoring DTD that pulls in both the specialized elements for the new type and the more general elements it reuses from ancestor types.

An authoring DTD can pull in other DTD and module files dynamically (that is, by reference). When an XML tool (like an editor or a parser) encounters an external entity reference in a DTD, it pulls in the referenced file and treats it as part of the DTD. This is important functionality for DITA, because it lets you split your DTD into chunks based on topic type and maintain each chunk in a separate module. This approach

lets you get the type maintenance you need: you can manipulate general characteristics of your types by editing the elements in the ancestor modules, and you can tweak specific characteristics of your types by editing the elements in the child modules. This lets you choose what markup is shared across some or all your types, and what markup is specific to a single type. Even though each module defines only part of a full DTD, assembled together they define a complete topic type.

For example, to define the DTD for a new task type (`bctask`), you need modules for the topic (`topic.dtd`), for the task (`task.mod`), and for the new specific type (`bctask.mod`). You can make the process a little simpler by just pulling in `task.dtd` (which will pull in `topic.dtd` for you) and `bctask.mod`. This also helps insulate you from any changes further up the hierarchy: for example, if `topic.dtd` were changed, you wouldn't be directly affected, since you need to reference only `task.dtd`, which in turn references `topic.dtd`.

The three information types discussed (task, concept, and reference or reftopic) form the base of a hierarchy of types that you can add to. When you add a new type to the hierarchy, you need to

1. Identify the elements that you need.
2. Identify the mapping to elements of a more general type.
3. Verify that the content models of specialized elements are more restrictive than their general equivalents.
4. Create a module file that holds your specialized element and attribute declarations.
5. Create an authoring DTD file that imports the new type modules, and the modules for its ancestor types. The ancestor type modules can be referenced either directly, by referenc-

ing each of their module files, or indirectly, by referencing the authoring DTD for the parent type only.

For example, for a new task type (`bctask`), that specialized task would need the following files:

- ◆ `bctask.mod`—declares `bctask`-specific elements
- ◆ `bctask.dtd`—embeds `bctask.mod` and `task.dtd`

The modular format lets you easily assemble an authoring DTD that includes the topic types that you need. The DITA package includes authoring DTDs for each individual type, plus an authoring DTD for creating compound documents.

By following these conventions when you create a new topic type, you should

- ◆ Reduce the cost of entry by making it easier to define a new type
- ◆ Reduce the cost of maintenance by making it easier to maintain types, and type hierarchies
- ◆ Reduce the cost of exiting the type by allowing easy specialization from your existing type to even more specialized types

A topic with a specialized type is still, conceptually, a topic of a more general type as well. That is, a specialized task is still a task and still a topic; and a Java API description is still an API description, still a reference topic, and still a topic. The more specific types are subsets of the more general types.

This type of relationship exists not only at the topic type level, but also at the individual element level. A specialized kind of task step is still a kind of step, which is still a kind of list item. These equivalencies are very important for allowing interchange at more general levels (you may care that an element is a step, but someone else may need to know only that it's a list item), and also for allowing reuse of processes (many processes will care only about list items, and may not know about steps at all).

Preparing an expense report

Before you start, have the following handy:


- Itinerary from the travel agent
 - Airplane ticket stubs or e-ticket confirmation number
 - Meal receipts totaling more than \$25.00
 - Transportation receipts: car rental contract or taxi receipt
1. If you traveled internationally, find the exchange rates for the days you traveled at <http://sample.only.com/travel>. This information will be requested later.
 2.  Using the GETPAID accounting tool, enter information
 - a. On the itinerary, find the trip reference number. Type it in the Trip number field and press Enter. This action transfers itinerary information into the form.
 - b. On subsequent pages, enter additional information as requested.
 - c. On the last page, click Finish. The tool displays a summary report and saves it to the hard disk as an HTML file.

Figure 9. Specialized content allows specialized display.

While it's pretty easy for humans to figure out the type hierarchy (figuring out that `bctask` is a task is a topic isn't very hard work), XML tools and processes aren't as smart. Class or type hierarchies aren't things that XML currently does very well. To make the equivalency between a type and a subtype available to processes that need to work on one level or the other, the mapping between general and specific elements needs to be stored somewhere.

The answer is to store the information as default attribute values for each element type. The default values will be treated as if they're there in every XML document that references the type DTD. For example, here are three increasingly specialized versions of a list item (showing the spec attributes, though an author wouldn't necessarily see them):

1. ``This element is part of a topic.``

2. `<step spec=" topic.li task.step ">`This element is part of a task, but can be treated as part of a topic.`</step>`

3. `<appstep spec=" topic.li task.step`

`bctask.appstep ">`This element is part of a specialized task, but can be treated as part of a general task, or as part of a topic.`</appstep>`

And here's the actual DTD declaration for the spec attribute for `<appstep>`:

```
<!ATTLIST appstep spec
CDATA " topic.li task.step
bctask.appstep " >
```

This spec attribute is the key to two of the biggest promises of DITA: the ability to interchange information in a wider community, and the ability to define a new topic type and still use processes that were created for a more general type. No matter how specialized your type is, it always maps all the way up to simple topic structures, and thus it's interchangeable with anyone else using DITA. And no matter how specialized your information is, you can still use any of the general processes and transforms that apply, such as publishing to HTML or preparing a PDF file.

Right now, DITA is still in its infancy, and there aren't actually a lot of processes around to be reused. But the architecture is in place, and processes will follow. The good

news is that when they come, even though they weren't written with your specialized types in mind, they'll still work on your content because of the spec attributes.

Extended example:

`bctask.dtd`

Earlier, we discussed a sample task: how to complete an expense report. The basic structure of that task is as follows:

1. What you need before you start (prerequisite items or tasks)

2. Steps for this task

3. Follow-up after this task is completed (post-requisites)

For this example, we've chosen to distinguish steps that involve software from other kinds of steps, presumably because employees need to know where to get these tools to use them. For "application steps," a small computer graphic indicates that the step requires software. Alternative text for the graphic indicates the name of the tool and where to get it. Figure 9 shows how the final product looks.

To implement this design in a DTD module, we extended the task info-type to create a specialization called `bctask` (for "business-control task"). We added one element for the new step type, `appstep` (an extension of the `step` element) and one for `appdesc`, which includes an attribute for the alternative text. Its label attribute is used in the XSL script to select the correct graphic. Figure 10 shows a snippet from the DTD module.

We also had to implement the container elements up to and including the `info-type` container, as shown in Figure 11.

Then we included specialization attributes for each of these elements so that existing XSL scripts for the task info-type could be used, as shown in Figure 12.

The result is an XML DTD module that defines five elements and their attributes but can be com-

```

<!-- new step type and description -->

<!ELEMENT appstep (cmd, appdesc,
(info|substeps|taskxmp|choices)*,
stepresult?)>
<!ELEMENT appdesc #PCDATA)* >
<!-- ATTLIST appdesc label (SAP|Web|Notes|Custom) "Web"
%univ-atts;>

```

Figure 10. An excerpt from a DTD module defining specialized elements.

bined with the more general types to create an XML document that looks like Figure 13. The specialized element names are in bold; all the other elements are reused from task or topic.

Only a few of the elements are actually defined by *bctestp*; most of them are just reused from task, or even from topic. The spec attributes aren't shown because they don't need to be there: as long as they are defined in the DTD, XSLT (XSL Transformation) processors will treat them as if they're present.

The advantages of specialization

In summary, specialization offers several benefits:

- ◆ You have to do less up-front analysis.
- ◆ Because of general categories, you can share information with wider groups.
- ◆ Because of specific categories, you can concentrate on meeting your audience's needs.
- ◆ Because of reuse by reference, your information can remain compatible, even when the general categories evolve.

FIX THE PROCESSES

How can we make it easier to create the processes you need for your specialized markup? We've discussed how processes can make use of the spec attribute to allow specialized content to reuse general processes, but that's not enough. We also need to make it easier for you to create new processes so that you can get the benefit of your new markup with as little investment as possible.

In this section, we'll cover in more detail how general processes can work on specialized content, and

```

<!-- specialize the task element to include bctaskbody -->
<!ELEMENT bctask (title, prolog?, bctaskbody, (%info-types;)* )>
<!-- ATTLIST bctask appid CDATA #IMPLIED
id CDATA #REQUIRED
%univ-atts;>
<!-- specialize the taskbody element to include bcsteps -->
<!ELEMENT bctaskbody (prereq?, context?, bcsteps?, result?, xmp?,
postreq?) >
<!-- ATTLIST bctaskbody %univ-atts;>
<!-- specialize the steps element to include appstep -->
<!ELEMENT bcsteps ((step|appstep)* ) >
<!-- ATTLIST bcsteps %univ-atts;>

```

Figure 11. Implementing additional custom elements in a DTD.

```

<!-- specialization attributes -->
<!-- ATTLIST bctask spec CDATA " topic.topic task.task
bctask.bctask " >
<!-- ATTLIST bctaskbody spec CDATA " topic.body task.taskbody
bctask.bctaskbody " >
<!-- ATTLIST bcsteps spec CDATA " topic.ol task.steps
bctask.bcsteps " >
<!-- ATTLIST appstep spec CDATA " topic.li task.step
bctask.appstep " >
<!-- ATTLIST appdesc spec CDATA " topic.ph task.info
bctask.appdesc " >

```

Figure 12. Adding specialization attributes for new elements to allow generic transforms to handle custom elements.

```

<?xml version="1.0"?>
<!DOCTYPE bctask SYSTEM "bctask.dtd">
<bctask id="sample">
  <title>Preparing an expense report</title>
  <bctaskbody>
    <prereq>
      <p>Before you start, have the following handy:</p>
      <ul>
        <li>Itinerary from the travel agent</li>
        <li>Airplane ticket stubs or e-ticket confirmation number</li>
        <li>Meal receipts totaling more than $25.00</li>
        <li>Transportation receipts: car rental contract or taxi receipt</li>
      </ul>
    </prereq>
    <bcsteps>
      <step>
        <cmd>If you traveled internationally,
        find the exchange rates for the days you traveled at
        http://sample.only.com/travel.
        This information will be requested later.</cmd>
      </step>
      <appstep>
        <cmd>Using the GETPAID accounting tool, enter information about your
        trip.</cmd>
        <appdesc label="Custom">GETPAID is available from Jane Doe in
        IT.</appdesc>
        <substeps>
          <substep>
            <cmd>On the itinerary, find the trip reference number.
            Type it in the Trip number field and press Enter.</cmd>
            <info>
              This action transfers itinerary information into the form.
            </info>
          </substep>
          <substep><cmd>On subsequent pages, enter additional information as
          requested.</cmd></substep>
          <substep>
            <cmd>On the last page, click Finish.</cmd>
            <info>The tool displays a summary report and saves it to the hard disk
            as an HTML file.
            </info>
          </substep>
        </substeps>
      </appstep>
      <appstep>
        <cmd>Using a Web browser, submit the report.</cmd>
        <appdesc>Your browser must support Java 1.2.2.</appdesc>
        <substeps>
          <substep>
            <cmd>Open the newly created HTML file and verify the information.</cmd>
          </substep>
          <substep>
            <cmd>If all is correct, click Submit.
            Otherwise, return to the GETPAID step.
          </cmd>
        </substep>
        </substeps>
      <stepresult>This submits the report to the accounting system and
      sends a paper copy to your workstation's default printer.
    </stepresult>
    </appstep>
    <step>
      <cmd>Send the paper copy with receipts to John Doe in Accounting.
    </cmd>
    </step>
  </bcsteps>
  <postreq>
    <p>If you do not receive a confirmation within two weeks,
    contact John Doe at (800) 555-1212.</p>
  </postreq>
</bctaskbody>
</bctask>

```

Figure 13. The specialized element names are in bold; all the other elements are reused from task or topic.

we'll also discuss how to create new processes that reuse the general processes by reference. The first capability (upwards compatibility) ensures that you don't lose access to any-

thing you need when you specialize: your content-specific markup doesn't cut you off from the world of general processes. The second capability (reuse by reference) ensures that you

can create and maintain new processes as easily as you create and maintain new topic types.

To get the reuse and compatibility your specialized types require,

```
basictransform.xsl
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
  <!--match the root - add the html and body tags-->
  <html>
    <body>
      <xsl:apply-templates/>
    </body>
  </html>
</xsl:template>
<xsl:template match="title">
  <h1>
    <xsl:apply-templates/>
  </h1>
</xsl:template>
<xsl:template match="steps">
  <ol>
    <xsl:apply-templates/>
  </ol>
</xsl:template>
<xsl:template match="step">
  <li><xsl:apply-templates/></li>
</xsl:template>
</xsl:stylesheet>
<!--any markup that doesn't have a rule associated with it (like
taskbody) is stripped out by default-->
```

Figure 14. An XSLT transform to convert to XHTML.

you need to

- ◆ Create processes around topic types instead of around document types.
- ◆ Create processes that depend on the information in the `spec` attribute, so they can tell not just what the current element is, but what its equivalents are in higher-level types.
- ◆ Create new processes by extending existing ones, reusing general logic by reference, and defining your specialized logic only as a delta, relative to the general process.

As with content chunking and DTD design, reuse by reference is

key. For processes, it means that if a fix or enhancement is applied to the original process, your new processes automatically pick up the change.

This approach guarantees that you have the specific behavior your specific content needs without ballooning maintenance costs.

Some XSLT background

The examples in this section use processes implemented as XSLT transforms. XSLT is a programming language specifically created to work with XML. Generally, an XSLT transform consists of a number of *template rules* that say what to do when the transform encounters a particular kind

of element. Typically the template rule instructs the process to continue working on the current element's children, applying any template rules that match. Figure 14 shows a simple XSLT transform that turns the content into an HTML document.

Given the following input:

```
<task>
  <title>My title</title>
  <taskbody>
    <steps>
      <step>My step</step>
    </steps>
  </taskbody>
</task>
```

the result is an XHTML document with the following structure:

```
overridingtransform.xsl
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:import href="task.xsl"/>
<xsl:template match="step">
  <li><xsl:apply-templates/><br/></li>
  <!--br/ is the XHTML way of saying br-->
</xsl:template>
</xsl:stylesheet>
```

Figure 15. A specialized XSLT transform that extends and customizes the formatting from a standard transform.

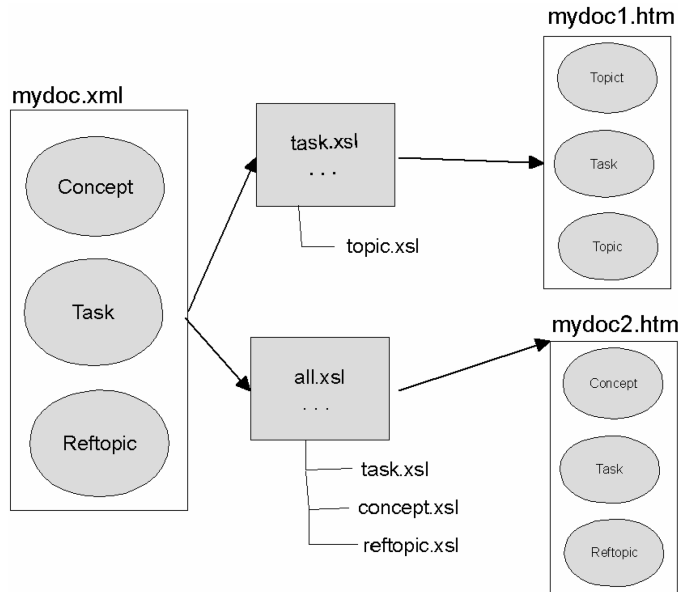


Figure 16. A compound document gets processed by a specific transform that treats unknown content generically, and then processed again by a compound transform.

```
<html>
  <body>
    <h1>My title</h1>
    <ol>
      <li>My step</li>
    </ol>
  </body>
</html>
```

XSLT takes a while to get the hang of, but provides a very flexible way to work with documents.

XSLT also lets you override behavior in an existing transform by defining new template rules in one file and then importing the existing transform to provide default behavior for everything else. For example, if you wanted to force a new line after each step (to create more white space in a printed version), you could create a new XSLT transform that imports the old one and then overrides the step template, as in Figure 15.

The `<xsl:import/>` statement always gives higher precedence to template rules in the *importing* stylesheet, so even though two tem-

plates match `<step>` (one in `basictransform.xsl` and one in `overridingtransform.xsl`), it's the expanded list template (in `overridingtransform.xsl`) that is applied, because it's in the importing (top-level) stylesheet. So you can

use `basictransform.xsl` when you want compact lists, and `overridingtransform.xsl` when you want expanded lists, with complete reuse of all the transform logic except the template for `<step>`. This overriding behavior is very important for DITA, because it lets you create a complete stylesheet out of multiple smaller modules without worrying about potential conflicts: the importing stylesheet always takes precedence.

Design processes around topic types

Earlier in this article, we covered chunking your content into topics and chunking your design into modules. Now you need to do the same kind of chunking with your processes for many of the same reasons: smaller chunks let you easily reuse logic among transforms, let you assemble document-specific transforms easily, and let you maintain common rules and behavior in a hierarchy.

So instead of a single large XSLT process, you get a bunch of smaller modules. You can then create the process you need by including the modules for each of the types you care about. For example, if all you

Preparing an expense report

Before you start, have the following handy:

- Itinerary from the travel agent
 - Airplane ticket stubs or e-ticket confirmation number
 - Meal receipts totaling more than \$25.00
 - Transportation receipts: car rental contract or taxi receipt
1. If you traveled internationally, find the exchange rates for the days you traveled at <http://sample.only.com/travel>. This information will be requested later.
 2. Using the GETPAID accounting tool, enter information about your trip.

GETPAID is available from Jane Doe in IT.

 - a. On the itinerary, find the trip reference number. Type it in the Trip number field and press Enter. This action transfers itinerary information into the form.
 - b. On subsequent pages, enter additional information as requested.
 - c. On the last page, click Finish. The tool displays a summary report and saves it to the hard disk as an HTML file.

Figure 17. Task transform applied to `bctask` content.

```

<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <!--get default behavior by importing the more general module-->
  <xsl:import href="task.xsl"/>
  <!--specify that you're outputting html-->
  <xsl:output method="html" encoding="iso-8859-1"
    indent="no" />
  <!-- Assign graphic variable for each application type.
    For now, they're all the same -->
  <xsl:variable name="webicon">appstep.gif</xsl:variable>
  <xsl:variable name="notesicon">appstep.gif</xsl:variable>
  <xsl:variable name="sapicon">appstep.gif</xsl:variable>
  <xsl:variable name="customicon">appstep.gif</xsl:variable>
  <!-- dummy template rule; this is actually processed with appstep -->
  <xsl:template match="appdesc"*[contains(@spec,' bctask.appdesc ')]" />
  <!-- The following template rule reflects default label value of "Web" -
    -->
  <xsl:template match="appstep"*[contains(@spec,' bctask.appstep ')]">
    <li>
      <xsl:variable name="alt-text">
        <xsl:value-of select="appdesc" />
      </xsl:variable>
      <xsl:choose>
        <xsl:when test="@label='Custom' ">
          
        </xsl:when>
        <xsl:when test="@label='Notes' ">
          
        </xsl:when>
        <xsl:when test="@label='SAP' ">
          
        </xsl:when>
        <xsl:otherwise>
          
        </xsl:otherwise>
      </xsl:choose>
      <xsl:text> </xsl:text>
      <xsl:apply-templates />
    </li>
  </xsl:template>
  <!-- Templates for other specialized elements should not be necessary
    because no specialized processing is done for them in this sample.
    The more generic template rules should apply.
    -->
</xsl:stylesheet>

```

Figure 18. The XSLT stylesheet for custom formatting.

care about are topics, all your process needs is a topic module. Every kind of DITA content you throw at this process will get treated the same way, as a topic. But if you want to distinguish between tasks and reference topics, you'll need to add a task module and a reftopic module, where you can define what you want to do differently for tasks and reference topics.

Each XSLT module needs to address only the special behavior that you need for that type. If there is

existing general behavior that is appropriate, you can incorporate it by reference. Generally speaking, each XSLT module will import the module for the parent type as well.

This modular approach lets you assemble the behavior that you need for a particular document, regardless of the topic types in it. And the inclusion of the general rules means that you have a fallback for content that you hadn't anticipated. For example, if your task process gets applied to a mix

of concepts, tasks, and reftopics, then the tasks get treated as tasks but the others get treated as topics (as with `mydoc1.htm` in Figure 16). If you want to extend your process, add the modules for the other topic types (as with `mydoc2.htm`, in Figure 16).

Extend processes with reuse by reference

When you want to extend a process to create special handling for a new information type or to change the han-

dling for an existing type, you don't need to recreate all the logic of the generic process. You need to create only the logic that's different from the generic. Then you can import the generic process to provide the rest of the behavior.

To create an XSLT transform that provides specialized behavior, you need to

1. Import the transform that you want to extend (probably the transform for one of the existing ancestor types), using the `<xsl:import/>` statement.

2. Identify the elements that you need to treat specially.

3. Add template rules that match those elements, both by element name and by `spec` attribute content.

The new template rules will take precedence over the existing ones, thanks to the way import works (imported templates have lower precedence). New template rules will be used in place of their equivalents in the original transform. And the rest of the rules (those that you didn't override) handle the rest of your content.

One of the advantages of reuse by reference is that if you apply a fix or enhancement to the original process, your new transform picks up the change automatically.

Your transform can be specialization-aware: it will work not only with your content but also with any content written in more specialized types. If you choose to extend your type hierarchy at some later date, your process will handle the new subtypes automatically.

Extended example:

`bctask.xsl`

Once you have a specialized topic type, you can decide whether any of your specialized elements need special treatment. In our business control task, `appstep` and `appdesc` are the only

elements that require special treatment: all the other elements (including specialized ones) can be handled by the more generic `task.xsl`.

Figure 17 shows what our sample content would look like, with just the default task transform applied.

By comparing Figure 17 with the earlier version in Figure 9, you can see that the computer graphic is gone, and the sidebar information is included inline. Everything else is as it was in the first case. In fact, almost all processing in the first case was handled through the task-based script. Figure 18 shows the complete XSLT transform that produces Figure 9.

Because we've included specialization attributes for the new elements, we can rely on existing scripts to handle any new element that does not require special processing. Processing for both `appstep` and `appdesc` is handled under a single template rule for `appstep`; only an empty rule is included for `appdesc`.

And because this transform also checks the `spec` attribute (that's what the "contains" part of the match statement is doing), it too can be reused by further specializations: no matter how specialized

your task hierarchy gets, any topic types that are descendants of `bc-task` can be validly processed by this transform.

The benefits of specialization-aware transforms

In summary, specialization-aware transforms offer several benefits:

- ◆ You don't absolutely require custom tools and scripts to support your custom DTD.
- ◆ You can more easily create custom tools, because you can reuse general logic.
- ◆ You get the benefit of improvements or fixes to the core logic automatically because of reuse by reference.

CONCLUSION

Each of these three components of the DITA solution focuses on a different aspect of reuse: content, design, and processes. In each case, we need to make some changes in the way that we do things before we can get the reuse that we need. In each case, we get the benefits of reuse by reference: control of reuse is in the hands of the reuser, reused content is insulated from the contexts that reuse it, and reusers

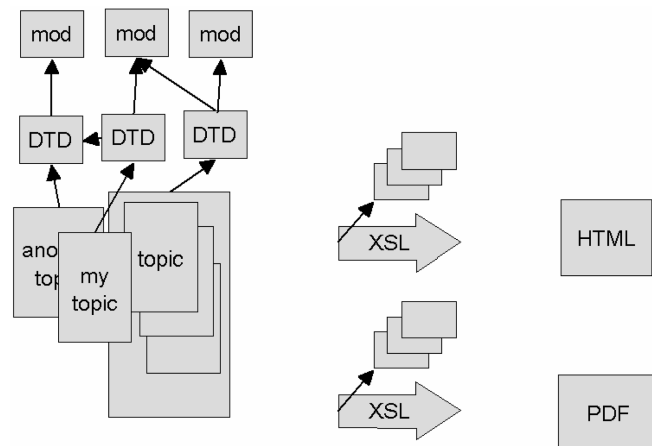


Figure 19. Documents reuse topics, DTDs reuse modules, and transforms reuse templates.

dynamically pick up changes to content.

To summarize:

Content reuse

- ◆ Chunking lets us reuse content without compromising the source.
- ◆ Typing lets us describe content within a chunk.

Design reuse

- ◆ Specialization lets us describe content in both general terms and specific terms.
- ◆ Standard information types make it easier to create the specific information types your audience needs.
- ◆ Specialized information types automatically pick up fixes or enhancements to higher-level types.

Process reuse

- ◆ General processes work automatically on specialized information types.
- ◆ Specialized processes can reuse the general processes and need define only the differences.
- ◆ Specialized processes automatically pick up fixes or enhancements to higher-level types.

This solution gives us an XML publishing scenario in which the three major components—design, content, and process—are factored out into smaller, reusable chunks, and assembled as required to build the solutions your content requires. The concept shown in Figure 19 isn't as simple as the one shown in Figure 1 at the beginning of this article, where each component was a single file. But the reality is that it never was that simple. If you want to reconcile reuse and usefulness, you need an architecture that breaks things down to the level of reuse you need, so you can build them up into the useful solutions

you require.

The promises of XML were reuse, descriptive markup, and interchangeability. By using the principles outlined in this article, it is possible to deliver on these promises. Information typing and topic chunking get us content reuse, but DITA goes further by easing the creation of specialized topic types for descriptive markup and by defining a mechanism for accessing a type hierarchy that allows interchangeability both of content and of processes.

The fundamental tradeoff of XML has always been that the more useful the markup is to you, the less interchangeable it is with other people. The fundamental promise of DITA is that you can have both useful markup and interchangeability. **TC**

RESOURCES

XML

<http://www.w3.org/TR/REC-xml>
<http://www.oasis-open.org/cover/index.html>
<http://www.oasis-open.org/cover/xmlIntro.html>

XSLT

<http://www.w3.org/TR/xslt>
<http://www.ibiblio.org/xml/books/Bible/updates/14.html>

Introductions to XML and XSLT

[http://www-106.ibm.com/developerworks/xml/\[Select "Education."](http://www-106.ibm.com/developerworks/xml/[Select)

Information architecture

<http://argus-acia.com/>
<http://www.builder.com/Authoring/AllAboutIA/>

DITA on developerWorks

<http://www-106.ibm.com/developerworks/xml/library/x-dita1/index.html> [Registration required.]

The DITA forum

[news://news.software.ibm.com/ibm.software.developerworks.xml.dita](http://news.software.ibm.com/ibm.software.developerworks.xml.dita)

The DITA package (DTDs and samples)

<http://www-106.ibm.com/developerworks/xml/library/x-dita1/dita00.zip>

MICHAEL PRIESTLEY is an information developer for the IBM Toronto Software Development Laboratory. He has worked as a writer, team leader, and information architect for various application development products. He has written numerous papers on subjects such as single-sourcing, hypertext design, and documentation processes. He is currently the specialization architect for the Darwin Information Typing Architecture. Contact information: mpriestl@ca.ibm.com

GRETCHEN HARGIS is a senior software engineer at the Silicon Valley Laboratory of IBM. She has worked as a writer, team leader, and editor on many projects, including knowledge-based systems and application development tools. She is an author of *Developing quality technical information* (Prentice Hall). Contact information: ghargis@us.ibm.com

SUSAN CARPENTER earned an MS in journalism (science communication) from Boston University in 1982. Shortly thereafter, she joined what was IBM's Federal Systems Division (FSD) laboratory at Manassas, VA. For the last 7 years, she has written product information or sample code to support C++, Java, and Smalltalk tooling on various workstation operating systems at IBM's Research Triangle Park, NC, software laboratory. She recently joined the WebSphere Application Server team, where she writes about Java application programming. She has been an avid fan of markup languages since her FSD days. Contact information: carpnter@us.ibm.com