An XML Architecture for Technical Documentation: The Darwin Information Typing Architecture

Don Day, Erik Hennum, John Hunt, Michael Priestley, David Schell

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. We discuss several methods that can be used to extend DITA's basic topic types.

INTRODUCTION

Since 1999 IBM has been working on the move from SGML-based and HTML-based authoring systems to an XML-based authoring system for hypertext topic-oriented information. This move was motivated by XML initiatives from the W3C, internal work on topic-oriented architectures and information-typing, and a need for an authoring system with a low barrier to entry to begin using the system. We have described the details behind IBM's Darwin Information Typing Architecture (DITA) in previous publications ([1], [2], [3], [4], [5]). In this paper we continue to provide basics on the evolving roles and responsibilities of an authoring community related to DITA, provide more details on the extensible nature of DITA, and provide information about how to get started with DITA.

BACKGROUND

As with all current workhorse documentation solutions, truly the Darwin Information Typing Architecture has been built on the shoulders of giants.

Among the many kinds of document markup languages used in early text processing systems, IBM contributed the concept of generalized markup languages that was formalized in 1986 in the original ISO specification of the Standard Generalized Markup Language, SGML. Through the early 1990s, a number of major SGML markup languages and applications were created, primarily for various companies and governments. At this time, IBM created its own implementation, IBMIDDoc, which stands for "IBM Information Development Document type." For the past decade, this DTD (Document Type Definition) and its internal workbench (including editors, tools, and interfaces) have been the mainstay of IBM's delivery of product information in a multitude of national languages.

With the advent of the World Wide Web in the early 1990s and the initial specification of XML in 1998, IBM's ID community began an internal workgroup to evaluate XML and recommend its future use. The result of this activity was a topic-oriented design that consisted of an extensible core language; it was named the Darwin Information Typing Architecture to acknowledge 1) its dependence on principles of specialization and inheritance, 2) its incorporation of current content typing methodologies, and 3) its processing architecture, which is scalable to any number of delivery needs or aggregating principles.

Principles of DITA's design also reflect the legacy of major influences in the world of information systems. An early design principle was to borrow where possible on tag names that would be familiar to writers who had authored with HTML. Similarly, wherever there were tag names that were familiar to those who had authored in IBMIDDoc, these also were favored. The terminology associated with DITA's topic-orientation and information typing design comes from current best practices from research and academia. And DITA's processing model is based entirely on World Wide Web Consortium technologies, primarily XSLT, the standard transformation language, meaning that both the authoring and the production of deliverables can be based on standard function and easily available tools.

This architecture was initially published in the spring of 2000 on the IBM developerWorks™ site as a report and accompanying toolkit. Since then, IBM's teams have conducted usability sessions to improve the design, and used the DTDs in both prototypes and beta authoring systems for actual deliverables. Results of these activities, along with an updated specification on "domain specialization," were published in the spring of 2002. IBM's XML team continues to encourage the exploration and use of DITA through public forums, articles, and consultation.

THE PROMISE OF XML

Like many others, we were excited by the promise of XML:

- Single sourcing – you can use the same content to create books and online information
- Meaningful markup – gives you more useful information, with customizable views and intelligent search
- Interchangeability – based on standards, so can exchange content with others

As we looked closer, however, we discovered problems with the promises. When XML promises single sourcing, it's despite XML languages that enshrine media-specific constructs such as chapters in books or screens in presentations. When XML promises smart content that can generate customized views and intelligent search, it's in the face of standard XML languages that only know about paragraphs and lists. When XML promises interchangeability, it's limited to those who accept a common-denominator standard like XHTML.

Simply choosing XML doesn't in itself deliver any of the touted benefits. In fact, the second and third promises – smart content and interchangeability – appear to be in fundamental conflict. In sum, *the more useful your markup is to you, the more it will cost you, and the fewer people share that cost.*

In designing DITA as an architecture, we took aim at all three of the key benefits to see if we could assemble an XML solution that could bypass this traditional tradeoff. There are three separate areas that needed to be addressed:

- Fixing the content
- Fixing the design
- Fixing the processing

Fixing the content

While XML makes a big deal about separating form from content, media differences aren't just about fonts and page breaks: they're about how you structure content as well. If you turn a book into a set of Web pages, it's still going to read like a book, with all the standard limitations of a book, just transposed to a medium where they aren't necessary. As long as you author in media-specific structures, you'll be dragging that first medium's assumptions along with you to every new output form you attempt.

So what does media-neutral content look like? It focuses on tasks and concepts, not on chapters and appendixes. It follows the same basic information design principles that have informed good manual design and good online design for decades: task orientation, minimalism, and scenario-based development. If you author tasks and concepts, rather than sections and paragraphs, you have the makings of a topic collection that can be reordered for different needs, supporting different task flows for different users, and supporting different reading paths for different media. You don't have to add conditional processing directives to your source, and you can have truly descriptive markup that applies regardless of medium. You can read more about this in [3].

Summary:

- Topic-based authoring lets us reuse content without compromising the source
- Information typing lets us describe content within a topic

Fixing the design

Two of the key benefits – smart content and interchangeability – require a change in the way we design XML documents. Traditionally, people have either created their own DTD, used a standard one, or customized a standard one.

- When you create your own DTD, you need to create your own infrastructure as well: transforms to get to HTML and PDF, translation technologies, editor customizations, and so on. And if you need to exchange information with other people who don't use your standard, there's quite a bit of negotiating to do.
- When you use a standard DTD, you get a standard infrastructure, which is great as far as it goes. But it won't describe your content specifically, so you don't get smart search or customized views. And it won't describe your business rules, so you'll need to create special tools to check those, or invest more time in enforcing editorial rules, or just live with inconsistency and hope your customers don't mind.
- When you customize a standard DTD, you sacrifice compatibility. The more you customize, the less standard it is; the less you customize, the less useful it is. Customization can give you a continuum instead of a yes or no choice, but it doesn't get rid of the tradeoff, and it can land you in real trouble when the standard evolves and you're not part of it.

DITA avoids the tradeoff by using a technique called specialization, which applies the well-established principles of inheritance and polymorphism from object-oriented design and applies them to the world of information design. This lets you create new designs by specializing existing designs. At the end of it, you get a customized solution that is still compatible with the existing infrastructure. You can customize as much as you want without breaking the infrastructure, and without compromising interchangeability. In other words, you can make your markup specific to your

content and business rules, and still get the benefits of using a standard language.

The two main principles of specialization are modularization and inheritance: break your design into modules based on information type (such as concept and task) and domain (such as programming and user interfaces). Then map the modules into a hierarchy, so you can say, for example, that both "concepts" and "tasks" are kinds of topics.

Anecdotally, we've also found specialization to be a lot faster than traditional XML development. The example specialization in [3] took only one or two days to develop and test, as opposed to the months it would have taken using more traditional methods.

Summary:

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

Fixing the processing

Specialization lets us get output from newly designed content immediately, because the processes designed for higher-level information types apply by default to new, specialized information types. However, when the existing treatment isn't exactly what you want, you can modularize and override your processing just as you did for your design. This lets you get specific processing when you need it, again without compromising reusability or interchangeability.

Summary:

- 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.

Delivering on the promise

Following the principles of information typing and specialization in DITA, you can create task-oriented, audience-oriented information that is reusable, useful, and standardized, all at the same time. For a more in-depth discussion of these three areas, see [3]. The next sections of the paper discuss how you can implement these principles in a writing team, and discuss how you can use specialization with other extension techniques to customize DITA for your needs.

WORKING WITH DITA ON A TEAM

Now that we've discussed what uses DITA is designed to support, let's look at how its use affects a team, and the responsibilities and rewards for each of the roles on the team.

The roles and responsibilities related to DITA include the following:

- **Type architect.** Analyzes topic types needed to accommodate content being produced, and defines new topic types if needed.
- **Topic writer.** Writes and edits topics, according to the topic-type standards established for the project by the XML architect.
- **Information architect.** Analyzes the overall structure of the content, groups it into topic collections, and defines maps that describe the relationship of topics to each other.
- **Build developer**. Processes the DITA source topics into various formats, as needed for product deliverables.
- **Information designer.** Establishes the "look and feel" of the output presentation.

The following table describes the responsibilities and rewards associated with each of these roles.

Role	Responsibilities	Rewards
Type Architect	Analyze the information needs of the project. Identify DITA topic types that best meet those needs. Define specialized DITA types, as needed, to meet more specific information needs. Map elements in specialized types back to existing ancestor elements in DITA base types.	Possible to extend the DITA base model to new types and new purposes very quickly and with little new DTD code. Derive new DTD markup without changing existing, common markup. Reuse higher-level types by reference and automatically pick up changes and updates. Common content remains unaffected by specialized types, since those specializations occur at a lower level of DITA.
Writer	Select the appropriate topic type for the content. Use an XML editor that validates against the DTD (no extra effort from the writer).	Focus on writing and content, with DITA providing overall structure and "coaching." No need to re-work content every time it gets reused in a different context. DITA and XML ensure that if you create valid topics, the content will pass muster when used to generate project deliverables in various required output formats. Working with markup that is specific to your content means less extraneous markup to ignore.
Information Architect	Take a "big picture" view of the project content. Design topic maps, which describe the overall organization and relationship among topics. Develop specialized maps for particular contexts, such as different output devices, different audiences, or different platforms. Consider ways in which topic maps can get reused for different contexts and output formats. Identify information gaps and missing topics for writers to create.	Information about topic relationships and linking output cleanly and appropriately to each output format. For example, print output doesn't include links, and online output doesn't include print cross-references. No need to worry about overlapping information sets. Other projects can maintain topic maps that overlap with the same set of topics, but each map remains unique with its project. Easy maintenance and review, because you can view and maintain all of the relationships for a particular context in one place.
Build Developer	Define topic sets needed for generating specialized outputs, as specified by the information designer. Generate topics in specific output formats. Handle conversion of content to more general levels of DITA, as needed for reuse and exchange with content from other projects.	No need to establish a new process for each new topic type and DTD. Instead, you can rely on overall DITA process, with specializations for any specific types included in the current project. Any new process is defined as a delta against an existing DITA process. Changes and enhancements at the general DITA level automatically get picked up at more specialized levels. Strong input validation reduces problems of broken builds in output processing.
Information Designer	Identify the output and presentation requirements for a particular set of content, and work with the build person to get them implemented. Design search templates for specialized content.	Form of the input content is very consistent, due to the use of DITA topic types and validating DTDs. Easy to change and refine the output presentation, because it's applied as part of the build process and not stored as part of the input content.

EXTENDING DITA

A team that wants to extend DITA has several options. We've discussed how specialization lets you create new information types and domains, but it is only one of the ways to get the fit you need, as follows:

- Type architects **specialize** to create new information types or domains, and optionally new output behavior.
- Build developers **customize** to override only the output behavior, without changing the design of your content.
- Information architects **integrate** to select a subset of available information types, domains, and output behaviors for the needs of your team, and make them work together.

Compare these three approaches with designing a new DTD from scratch (entries in brackets are optional):

	Artifacts	**Costs/Benefits**
Specialization	Specialized DTD module Shell DTD (Specialized XSLT module) (Shell XSLT)	Small cost New design elements (New code) Migration/interchange supported by architecture's 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

By contrast, if you want specialized markup and behavior without using DITA, you must design from scratch:

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

GETTING STARTED

Before working with DITA, you need to set up a publishing environment. You'll need a minimum of an XML editor, an XSLT processor, and the DITA package.

This section lists some typical tools in each area. To check the full range of options, you might want to visit a Web site for XML resources such as XML Software:

http://www.xmlsoftware.com/

Installing an Editor

In most scenarios, you'll need a text editor to create your XML documents. If you have the gift of typing ever-valid XML, you can use your favorite text editor such as vi or Microsoft® Notepad. Otherwise, you might want to get a validating XML editor such as one of these:

Epic
http://www.arbortext.com/html/epic_editor_overview.html
XMetaL
http://www.softquad.com/top_frame.sq?page=products/xmetal/content_xmetal.html
XMLSpy
http://www.xmlspy.com/products_ide.html

Each of these editors has its own installation program.

Installing an XSLT Processor

The XSLT processor turns your XML into HTML and other formats.

Some popular XSLT processors are Java™ programs. If you use one of these XSLT processors, you'll need to install a Java runtime such as Java 1.4.1:

http://java.sun.com/j2se/1.4.1/

Java comes with an installation program.

Some popular Java-based XSLT processors are:

Saxon
http://saxon.sourceforge.net/
Xalan-Java
http://xml.apache.org/xalan-j/index.html

To install these programs, you unzip the packages and add one or more jar files to the CLASSPATH variable in the environment for your system. Look in the subdirectories of the unzipped packages to find the documentation that explains which jar files are required.

If you are using a Linux system, you might be interested in a native Linux XSLT processor:

libxslt
http://xmlsoft.org/XSLT/

Installing DITA

The final layer in this foundation is the release for the DITA architecture:

DITA
http://www-106.ibm.com/developerworks/xml/library/x-dita1/dita10.zip

Note: by the time you read this paper, there may be a more recent package. Check the main DITA developerWorks URL for details.You'll unzip the DITA package and reference the DTD and XSLT files in the subdirectories.

In your topic documents, use a SYSTEM document type (or an XML catalog) to point to one of the installed DITA DTDs. To transform your topic documents, use the appropriate command line for your XSLT processor to apply one of the DITA XSLT scripts to your document. That's all there is to it.

Enterprise Publishing Tools

What you've installed so far is enough to experiment with or even to use productively in a small group. If you're working with a large publishing organization, however, you'll want to install a build manager and a CMS (content management system).

The standard build manager for Java is called Ant:

http://jakarta.apache.org/ant/

For XML-based content management systems, you might check out Wyona or another CMS listed on the XML Software site at the start of this section.

CONCLUSION

DITA delivers on the promise of XML by focusing on fixing the content, design and processing problems in generic XML. Using DITA on a production team involves creating new roles and responsibilities on the technical writing team. In addition, a key attribute of DITA is its methods of extensibility, each with its ownspecific costs and benefits. A strength of DITA is its reliance on XML standard tooling to get started.

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
[5] Priestley, M., and Schell, D.A. (2002). Specialization in DITA: Technology, process, and policy. Proceedings of the 20th annual international conference on Computer documentation.. ACM: SIGDOC.
[6] Hennum, E., Priestley, M. and Schell, D.A. (2002). Specialization in DITA. Extreme Markup Languages 2002 conference proceedings.

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

David Schell is IBM's strategist and tools lead in support of its technical writing community. You can reach Dave at dschell@us.ibm.com

Don Day is DITA's lead architect. He has also represented IBM on the W3C XSL and CSS Working Groups. You can reach Don at dond@us.ibm.com

Michael Priestley is DITA's specialization architect. He writes product documentation, has published numerous papers, and is the vice-chair of ACM SIGDOC.
You can reach Michael at mpriestl@ca.ibm.com

Erik Hennum is DITA's domain architect. He is also the technical lead for User Assistance with the IBM Storage Group. You can reach Erik at ehennum@us.ibm.com

John Hunt is the User Assistance architect for Lotus. You can reach John at john_hunt@us.ibm.com

Trademarks