The Darwin Information Typing Architecture (DITA) holds many advantages over information authored directly in HTML, including better reuse, easily changed presentation styles, and easy single sourcing. In Part 2 of this two-part series on how to quickly migrate HTML topics to DITA, the author explains the details of migration, and shows you how to override parts of this process for ideal results.

The first article in this series focused on how to use the HTML-to-DITA migration tool that is currently in use within IBM. It provides basic information on the cleanup required prior to migration, as well as tips on the types of markup the tool handles best.

DITA articles are generally divided by information type: The tool as written supports migrations to topic, concept, reference, and task information types. This article provides in-depth information on the migration path for each of these types, and indicates the entry points for those wishing to override portions of the conversion. Finally, I offer a small sample override for those who want to bypass specific elements, and suggest items to use for more extensive overrides.

Where to start
If you are still unfamiliar with DITA, the following two articles are a good place to start:

- "Introduction to DITA"
- "Why use DITA to produce HTML deliverables?"

If you are eager to get started, but have not read Part 1 of this series, I recommend that you read it first.

Converting to a basic topic
HTML and DITA have many elements in common, which makes it relatively easy to migrate from HTML into plain DITA topics. Although migrating to the topic information type is not ideal, it is
sometimes the appropriate solution; the basic migration into a topic is also the foundation upon which more specialized migrations are based.

This migration creates basic topics by default, so you can either set the `infotype` parameter to "topic," or leave it unspecified. For simple topics with only a single heading, the migration to a topic is very simple. The heading is placed in the topic's title, and any meta information is placed in the prolog. The content of the file is placed in the body. At the end of the topic, a collection of links will be generated; it includes all of the external links found in your file. You might want to update these manually; in many cases you will want to remove a generated cross reference within the file, while sometimes you'll find it more appropriate to keep the inline reference rather than the related link.

The `<title>` element in HTML is often a shorter version of the displayed title, and is also what many users will see if they locate your topic through a search. For this reason, the utility places the contents of the `<title>` element in the `<searchtitle>` of the DITA topic, and place the contents of your first heading into DITA's `<title>` element. Note that if the two values are the same, the `<searchtitle>` element is not created. If you need to update this for your migration, just search for the `gentitlealts` template in the provided XSLT.

Some HTML elements can pass directly into DITA, such as `<p>`, `<ol>`, and `<pre>`. Others must be modified through a simple conversion, such as the `<table>` and `<dl>` elements. One exception to both of these rules is the `<div>` element: Because this element can be nested indefinitely, and is often used with CSS for presentation, it doesn't lend itself well to a general migration rule. So, the utility simply processes the contents. If you have regular structures that use the `<div>` element, you will probably want to override this section of the migration.

The `<span>` element is generally easy to map through, though it tends to have the same problems as the `<div>` element. If you regularly use the class attribute on `<span>`, you will want to override it in order to preserve your source's semantics. However, if you tend to use phrase elements solely for presentation, the default migration may be enough; it assumes that you wish to use highlighting elements, and preserves style information such as bold and italics. This set of elements is known as the **highlighting domain** in DITA.

**Complex topics**

Topics with more than one heading, or with lower-level headings, are more difficult to process. As with simple topics, the first title becomes the topic title. Everything up to the second heading is processed the same way -- it falls through directly into the topic body.

The second heading is used to generate a `<section>` element. That heading becomes the title of the section, and everything up to the third heading is included in that section. Every additional heading also becomes a section in this manner. This is done using an XSLT mode, which processes elements sequentially until it encounters another heading. I describe the **modes used by this utility** in more detail later in this article.

One exception to the heading processing is that each heading must be at the same level as, or one level below, the original heading. That is, in a topic that begins with `<h1>`, any other `<h1>` or `<h2>` headings create sections, but `<h3>` is not allowed. In a topic that begins with `<h2>`, any other
<h2>or <h3>headings create sections, but <h4> is not allowed. If these invalid headings are found, a section is created, but all of the contents are placed in a <required-cleanup> element to be fixed after the migration.

Listing 1 is a sample migration, showing the input XHTML lined up with the DITA output. It demonstrates how a secondary heading creates a section, but lower-level headings must be fixed.

**Listing 1. Sample migration path to the basic topic information type**

```xml
<html>
<head>
  <title>Topic title</title>
</head>
<body>
  <h1>Topic title</h1>
  <p>Intro paragraph</p>
  <table>...</table>
  <ol><li>list item</li></ol>
  <h2>Sub-heading</h2>
  <p>Text in the sub-heading</p>
  <p>More text</p>
</body>
</html>
```

**Converting to a concept topic**

Creating a concept topic is actually the same as creating a basic topic, except that the <topic> and <body> tags have new names. The only difference in the content model is that once you place a section in the body, you can use only sections or examples. This restriction is already followed with the general topic migration, so no new processing is needed for concept topics. To produce concept output with the provided migration, just set the infotype parameter to "concept."

Listing 2 is a sample migration path for a simple concept, with multiple secondary headings:

**Listing 2. Sample migration path to the concept information type**

```xml
<html>
<head>
  <title>Concept title</title>
</head>
<conbody>
  <p>Intro paragraph</p>
  <table>...</table>
  <ol><li>list item</li></ol>
  <h2>Sub-heading</h2>
  <p>Text in the sub-heading</p>
  <p>More text</p>
</conbody>
```
Converting to a reference topic

Creating a reference topic is slightly more complicated than creating a basic topic, because reference topics have a more restricted content model.

To produce reference output from the sample XSLT, set the `infotype` parameter to "reference." The main difference between the reference model and the basic topic or concept model is that everything in a reference must go into a section or table. Consider these possibilities:

1. HTML topics with a single heading and only paragraph or list content are simple. The heading becomes the topic title, and all of the text content is placed in a section in the body.
2. Topics with a single heading and a mix of paragraph and table content are a little tricky. The tables themselves go directly into the body; sections must be created to handle all of the content outside of the tables.
3. Topics with multiple headings and no tables are processed like ordinary topics or concepts. The only difference is that everything up to the first or second heading must also go into a section.
4. Topics with a mix of headings, ordinary content, and tables are the most complex. In this case, each table becomes a child of the body. Likewise, each heading creates a section inside the body. Elements in between tables and headings must be placed inside another section, which does not have a title.

This describes how the output is created. If you look at the `gen-reference` template, it appears much more complicated than it is because it includes a long condition that determines whether content exists between the initial heading and the first table or subheading. If so, that content is processed prior to the rest of the body. When you ignore the verbose XSLT syntax and view this as a simple test, the coding is very similar to that of a concept or task.

Listing 3 is a sample of the reference migration. You can see how elements must be grouped into sections to be valid; tables can stand alone outside of the sections. A secondary heading starts a new section.
Listing 3. Sample migration path to the reference information type

```html
<html>
<head>
<title>Reference title</title>
</head>
<body>
<h1>Reference title</h1>
<p>Intro paragraph</p>
<table>...</table>
<p>Text after table</p>
<p>More text</p>
<h2>Sub-heading</h2>
<p>Text in the sub-heading</p>
<p>More text</p>
</body>
</html>
```

Converting to a task topic

Tasks are the most complex articles for migration: They have the most restrictive content model, and their contents are processed differently based on whether or not the task actually contains steps.

The task model is made up of several optional sections, which must appear in sequence. These sections are `prereq`, `context`, `steps`, `result`, `example`, and `postreq`. When migrating a task, the biggest problem is determining how to split up the contents among these sections. The basic migration is not very intelligent: Any content that appears before a set of steps (that is, before the first ordered list) goes into the context section. Anything that appears after the steps goes into the result section. As you might expect, the steps themselves go in between as steps.

To create task output, you just set the `infotype` parameter to "task."

Tasks without steps

Tasks without steps are generally overviews of other task groups. If a task does not contain any steps, everything is placed in the context section. If any additional headings are found, they end up in `<required-cleanup>` elements, because it is not possible to nest sections within the context.

If your overview tasks tend to provide information on prerequisites, rather than general context information, it is pretty straightforward to place all of this into a prereq section rather than a context. If you want to split up the information, analyze your content to figure out how to split it up.

Tasks with steps

Tasks with steps are more common. For this migration, steps are identified as the first ordered list inside the topic. DITA tasks also allow for unordered steps; this support does not yet exist in the migration utility.
As with the overview tasks, you cannot have a series of headings. Everything up to the first ordered list is placed in the context section; this includes paragraphs and text, as well as tables, headings, or other types of lists. If your tasks have a regular structure, you may want to come up with a way to distinguish prereq and context information.

The first ordered list in your file is converted to a `<steps>` element. Each step has a very restrictive model: It must start with a `<cmd>` element that contains only text or phrase-level elements; the command can be followed by a series of block elements, including substeps and info. The info tag can contain anything from text to large block-level elements.

Each `<step>` element is created by processing one list item. If that list item only contains text or phrases, everything is placed in the command. If the list item contains block-level elements, then everything up to the first block is placed in the command; after that, any non-list items are placed in an `<info>` wrapper. The only exception is for nested ordered lists, which are placed in substeps. The substeps are processed in the same way as steps, except that substeps cannot contain more substeps.

Anything that comes after the steps goes into another section-like element. The available tags are `result`, `example`, and `postreq`. This utility drops all the content that follows into the `<result>` element. As with the context element above, if you have well-structured task information you may be able to split this up among the result, example, and postreq sections.

Listing 4 shows a sample of the most common task migration. It uses a task with some introductory material, some steps, and some follow-up material.

### Listing 4. Sample migration path to the task information type

```xml
<html>  
<head>  
<title>Task title</title> 
</head>  
<body>  
<h1>Task title</h1>  
<context>  
<p>Intro paragraph</p>  
<table>...</table>  
</context>  
<ol>  
<li>step one</li>  
<li>step two</li>  
</ol>  
<p>Text after list</p>  
<p>This is summary info</p>  
<p>Here is what you do next</p>  
</result>  
</taskbody>  
</body>  
</html>  
</task>
```

### Converting to something else entirely

The provided transform is good for basic topics, but what happens if you have information that you want in specific tags?
Overriding the migration for specific patterns

One of the things that makes processing DITA with standard transforms so useful is that those transforms are easy to override. When you need to change the output for one of your specific elements, you can do so, but still use the default behavior for everything else. The same is true when migrating into DITA.

One example of when to override the migration is when you regularly use the class attribute to indicate semantic information. In this case, you might want to override an element by matching the class. You can match any span with a class value of `command`, and place the contents into an actual `<cmdname>` element, rather than using the default migration to phrase.

In a slightly more complicated example, assume that you have a standard template for your task articles. In that template, the first sentence of your step always contains the command, and the remaining sentences describe the result. However, the step contains no tags to distinguish the command from the result. In this case, because the step is entirely plain text, the migration utility will place everything in DITA's `<cmd>` element. Logically it makes more sense for you to place the first sentence in the `<cmd>`, while placing the rest of the information in the `<stepresult>` element. This is easy with an override of the migration.

For this override, first create a simple XSLT shell that pulls in the standard migration. You can pull it in with a simple `<xsl:import href="h2d.xsl"/>` command. Then, override the processing of the individual steps. If you look at the provided migration utility, you will see that steps and substeps are handled using the same template. Listing 5 is a copy of that template.

Listing 5. XSLT template for processing task steps

```xml
<xsl:template match="li" mode="steps">
  <xsl:param name="steptype">step</xsl:param>
  <xsl:element name="{$steptype}"
    xsl:choose>
    <xsl:when test="not(p|div|ol|ul|table|dl|pre)"
      mode="step-cmd">
      <cmd><xsl:apply-templates select="*|comment()|text()"/></cmd>
    </xsl:when>
    <xsl:otherwise>
      <cmd><xsl:apply-templates select="(./*.|./text())[1]"
        mode="step-child"/></cmd>
    </xsl:otherwise>
  </xsl:choose>
</xsl:element>
</xsl:template>
```

This is the template that you need to override. Start by using the same match template, with no content:

```xml
<xsl:template match="li" mode="steps"/>
</xsl:template>
```

Your XSLT shell imports the standard migration, but XSLT automatically gives your new rule priority over the original one -- so if you migrate at this point, all of your steps will disappear. Now, add your new code. It does not need to be as complicated as the original, because your task...
template only uses plain text; you do not need to worry about substeps or child elements. First, simply create the step. You know that it has at least one sentence, so create your command element and insert all of the text before the first period. Do not forget to add a period for the one that gets dropped. Next, check whether any content appears after the command. I've placed the remaining content in a variable `result`. If that variable has any content, it is placed in a `<stepresult>` element:

**Listing 6. Sample override for processing task steps**

```xml
<xsl:template match="li" mode="steps">
  <step>
    <xsl:variable name="result">
      <xsl:value-of select="substring-after(.,'.')"/>
    </xsl:variable>
    <cmd><xsl:value-of select="substring-before(.,'.')"/></cmd>
    <xsl:if test="string-length($result)>0">
      <stepresult><xsl:value-of select="$result"/></stepresult>
    </xsl:if>
  </step>
</xsl:template>
```

This is very similar to an override that's used by some groups within IBM. Many overrides are more complicated than this, while others are this easy or easier. In particular, phrase elements are especially easy to override.

### Migrating to another specialized DTD

If you can divide all of your topics into basic concepts, tasks, or references, then you are set; but what happens if you want to migrate into another specialized DTD? In that case you might still make use of some of the transform, but your override will be more complex than the sample in Listing 6. How complex depends entirely on the state of your input files, and on the complexity of your specialized DTD. Several points to remember are:

1. As mentioned in the first article of this series, for any migration, determine how to add a `DOCTYPE`. Whatever method you choose, update this to recognize your specialized DTD as well.
2. The template that matches "html" calls a named template based on the input parameter. For example, a basic topic is created in the `gen-topic` template, and a task topic is created in the `gen-task` template. If you are converting to a command reference topic, you will likely want to add a `gen-cmdref` template to set up your root element and create the major structures within the topic. If you only intend to use your override for this new DTD, you can also just override the HTML element and use that to set up the output structures; this eliminates the need to pass in an infotype parameter.
3. At this point, the complexity depends entirely on your specialized DTD. If you want to migrate to a command reference, and it has the same content model as the basic reference, then you can simply copy the reference model. If your command reference has a very strict content model, then this can make things easier or more difficult. A very strict content model can mean that your input files already follow a strict model, so you know what to expect, making it easy to process the contents. Or, if you have to be prepared for anything that comes your
way, you will end up with some complex conditional coding to fit your specialized DTD. As with the earlier treatment of the task structures, you might choose to drop most of the content into a couple of major structures and then sort it out later.

4. In looking through the code, you will notice that a lot of the processing is done with modes. You may be able to reuse these in your overrides, or modify them slightly to turn a section into a specialized section. A few commonly used modes are:

- **creating-content-before-section:** This mode is used when processing the contents of a body. It takes in a single element, text node, or comment. If that node appears before any section-level heading, it is placed in the output, and processing continues to the following node. Processing stops when it hits a heading.

- **add-content-to-section:** This mode is very similar to the first mode, and is used once in a section. Processing is handled the same way: Non-heading nodes are placed in the output, and the utility advances to the next node. Processing stops when it hits a heading.

- **create-section-with-following-content:** This mode is used with all headings after the main heading. It turns the heading into a section title, and then places everything up to the next heading into that section. Notice that when generating reference topics, a table does not go into the section; instead, it goes directly into the reference body. If you wish to preserve this behavior in a specialized reference, you need to update the add-content-to-section mode to make sure it is aware of your new information type.

**Conclusion**

If you've made it this far, then you probably see the value in moving your HTML to DITA. Just to remind you of the main points, you can more easily reuse topics or parts of topics, you can produce multiple output formats, and you can rapidly change the presentation for an entire deliverable just by switching your CSS file. If you are not convinced, take a look at the article "Why use DITA to produce HTML deliverables?"

If you already have well-structured topics, first try using the provided conversion tool. You may find that you don't even need to dirty your hands with XSLT. However, you could just as easily decide that a few minutes of coding can give you an error-free migration. In either case, you should find that migration provides an easy path to your new authoring environment.
## Downloads

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>XSLT migration script and samples</td>
<td>x-dita8bmigration.zip</td>
<td>21 KB</td>
</tr>
</tbody>
</table>
Resources

- Read the previous article in this series, "Migrating HTML to DITA, Part 1" (developerWorks, January 2005).
- Learn more about DITA in these developerWorks articles:
  - "Introduction to DITA" (October 2003)
  - "Specializing topic types in DITA" (October 2003)
  - The DITA FAQ (November 2004)
  - "Specializing domains in DITA" (October 2003)
  - "Why use DITA to produce HTML deliverables?" (October 2003)
  - "Design patterns for information architecture with DITA map domains" (September 2004)
- Find additional information on DITA at the OASIS Cover Pages.
- Here are some tools that can be useful to you when doing the migration:
  - HTML Tidy, available from SourceForge.net; a Java version is also available on SourceForge at the JTidy page
  - The Saxon XSLT engine, also from SourceForge.net
  - The Xalan XSLT engine, from Apache
  - "Hands-on XSL" by Don Day (developerWorks, March 2000)
  - Another helpful XSLT tutorial
  - The XSLT mailing list
  - As well as these additional XSLT links
About the author

Robert Anderson

Robert D. Anderson has worked on IBM's Internal Information Development Workbench since 1999, writing and maintaining transformation tools for both SGML (IBMIDDoc) and XML (DITA). Since 2001 he has worked primarily on conversion tools to and from DITA, as well as general DITA design (prior to its shift to OASIS). You can reach him at robander@us.ibm.com.

© Copyright IBM Corporation 2005
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)