

Hacking the DITA Open Toolkit

WHITE PAPER



Simon Bate
Senior Technical
Consultant

The Darwin Information Typing Architecture (DITA) defines a set of XML elements for creating and organizing content. However, the DITA specification is silent on transforming DITA into user-readable documentation. The DITA Open Toolkit (DITA OT) fills that gap, providing a mechanism for transforming DITA content into multiple output formats, including HTML and PDF. The DITA OT formatting for both of these formats is basic, at best. This paper focuses on the changes you can make to the DITA OT HTML output to create attractive output. These modifications include changes to cascading stylesheets (CSS), headers and footers, and more advanced customizations. The paper also illustrates how you can create content-specific elements through DITA specialization.

NOTE: The information in this white paper covers increasingly complex changes to the DITA OT. Depending on the types of changes you decide to make, your skills should encompass these areas: XHTML (including CSS), XSLT, Ant, and XML DTDs.

What's in this paper?

This white paper describes changes you can make to the DITA OT in increasing level of technical difficulty.

- ❖ [Hacking vs. elegance](#)—Contrasts different approaches to modifying the DITA OT and provides suggestions for file management.
- ❖ [DITA OT Overview](#)—Provides background on the DITA OT, including how to install and run the DITA OT.
- ❖ [CSS changes](#)—Describes various strategies for changing CSS files in the DITA OT.
- ❖ [HTML headers and footers](#)—Describes how to use HTML files to create static headers and footers.
- ❖ [Modifying XSL transforms](#)—Describes different ways to change the XSL transforms in the DITA OT.
- ❖ [Packaging up your overrides](#)—Illustrates how to create a plugin for the DITA OT.
- ❖ [Hacking and specialization](#)—Describes how to modify DITA OT dtd files to specialize individual elements.



Hacking vs. elegance

When modifying or adapting existing code to your purposes, there are two extremes in how you implement the changes:

- ❖ You can change what needs to be changed to get the work done (hacking).
- ❖ You can make changes using a provided framework for integrating changes with existing code (elegance).

The DITA OT offers many opportunities for hacking, but it also provides a number of frameworks within which you can formalize your changes (separate CSS files, XSL override templates, and so on). It's up to you and your circumstances to decide between speed and maintainability. In general, you can start changes as hacks (just to get things working or as a proof of concept). Once you have the behavior you need, you can go back and reimplement the changes using the appropriate framework.

Comments and backups

No matter which route you take, it's vitally important that you add comments explaining your changes. The DITA OT consists of many thousands of lines of code distributed over dozens of files. You need comments to keep track of your changes.

Attaching the contributor's name and the date to comments can save many hours of searching and re-work. For example:

```
<!-- SFB 27-Jan-2009: Changed external to peer -->
```

Before you start to make changes, make a backup of the original file. Nothing is more frustrating than changing something and not being able to undo your changes because of a minor typo that might take hours to find. It's better to have a backup copy, which you can either restore or check with a file-comparison utility (`diff`), and proceed from there.

Finally, if you have source code management software available to you, use it! Common source code systems include Subversion, CVS, Perforce, IBM Rational ClearCase, and Microsoft Visual SourceSafe, but there are many more.

Output formats

The standard DITA OT distribution can create a number of output formats, including:

- ❖ XHTML—essentially HTML that uses XML syntax rules.
- ❖ Microsoft HTML Help (.chm files)—built from XHTML output.
- ❖ Eclipse help—a Java integrated development environment (IDE), the help is built from XHTML output.
- ❖ PDF—Adobe Acrobat format documents, generated using XSL-FO.

This paper focuses on XHTML-based modifications, mostly because the concepts of HTML and CSS should be familiar to most readers. However, many of the same principles used to generate XHTML output can be applied to generating XSL-FO output for PDF files.

DITA OT Overview

The files in the DITA OT fall into five fundamental categories:

- ❖ Ant project files that drive the DITA OT processing. These are XML files found in the root folder (ditaot) and in the ditaot/ant folder.
- ❖ DTD and related files that describe the DITA architecture, including content models and attributes. These files are stored in the ditaot/dtd folder. If you use DITA plugins, additional DTDs might exist in the ditaot/demo/<my-plugin>/dtd or ditaot/plugins/<my-plugin>/dtd subfolders. The name <my-plugin> is a placeholder for a plugin name. (This paper describes how to create your own DITA plugin under “Packaging up your overrides” on page 12.)
- ❖ XSL stylesheets that transform the source XML files into deliverables. These XSL files are typically found in the ditaot/xsl folder. If you use DITA plugins, additional stylesheets might exist in the ditaot/demo/<my-plugin>/xsl or ditaot/plugins/<my-plugin>/xsl subfolders.
- ❖ Documentation and examples for the DITA OT. These include sample Ant project files in the ditaot/ant folder, documentation (in DITA and output formats) in the ditaot/doc folder, and DITA example files in the ditaot/samples folder.
- ❖ Miscellaneous support files that provide processing assistance (ditaot/css, ditaot/lib and ditaot/tools folders) or support various output formats (ditaot/resources).

Running the DITA OT

The DITA OT requires that you have the Java SDK (JDK) version 1.5 or 1.6 installed. Download the “full easy install” version of the DITA OT and uncompress the file to a folder. The path to the folder should not contain spaces, so use something like c:\ditaot rather than c:\dita ot. This paper refers to the folder ditaot, which represents the directory in which the DITA OT is installed. You might choose a different file name. This is just a reference point.

You begin running the DITA OT by running a script or batch file that creates some environment variables. In Windows, the file is ditaot/startcmd.bat; in UNIX or Linux, the file is startcmd.sh.

In addition to creating the environment variables, these scripts open a command window. From the command prompt, you enter Ant commands that run the DITA OT. The top level of the ditaot folder contains a number of Ant project files that you can use to test the DITA OT installation. However, the Ant project files that you need to modify are in the ditaot/ant folder.

The examples in this white paper use the sample files from the ditaot/samples folder. The sample Ant file ditaot/ant/sample_xhtml.xml is already set up to process ditaot/samples/hierarchy.ditamap into XHTML output, so it’s a good file to use as the basis for your modifications. Make a copy of the file ditaot/ant/sample_xhtml.xml and name it something like my_xhtml.xml to make a custom test file.

When you first open my_xhtml.xml for editing, there are several locations to note in the file.



```

<?xml version="1.0" encoding="UTF-8" ?>
<!-- This file is part of the DITA Open Toolkit project hosted on
      Sourceforge.net. See the accompanying license.txt file for
      applicable licenses.-->
<!-- (c) Copyright IBM Corp. 2004, 2006 All Rights Reserved. -->
...
<project name="sample_xhtml" default="sample2xhtml" basedir=". ">
    <!-- dita.dir should point to the toolkit's root directory -->
    <property name="dita.dir" value="${basedir}${file.separator}.." />
    <!-- if file is a relative file name, the file name will be resolved
          relative to the importing file -->
    <import file="${dita.dir}${file.separator}integrator.xml" />
    <target name="sample2xhtml" depends="integrate">
        <ant antfile="${dita.dir}${file.separator}build.xml" target="init">
            <!-- please refer to the toolkit's document for supported parameters, and
                  specify them base on your needs -->
            <property name="args.input"
                      value="${dita.dir}${file.separator}samples${file.separator}hierarchy.ditamap" />
            <property name="output.dir"
                      value="out${file.separator}samples${file.separator}xhtml" />
            <property name="transtype" value="xhtml" />
        </ant>
    </target>
</project>

```

The callouts in the sample indicate places where you make changes to this file, as explained here:

- ❶ Change the project name `sample_xhtml` to `my_xhtml`, so that it matches the file name.
- ❷ and ❹ It's not essential to change the name of the default target (`sample2xhtml`), but if you do, make sure you change it both in the project declaration as well as in the target declaration.
- ❸ Later sections in this paper describe how to add additional `<property>` directives. This is the best place to add them.

Save your changes and run the Ant project from the command line that was opened by the `startcmd` script. The Ant command to run this modified project file is:

```
ant -f ant/my_xhtml.xml
```

You'll get lots and lots of output. If the project created its output correctly, the next to last line should say:

```
BUILD SUCCESSFUL
```

To view the output, use a web browser to open the file `ditaot/ant/out/samples/xhtml/index.html`.

Some who start working with the DITA OT are surprised or disappointed to realize they're back to the "old DOS prompt." If you are in this group, note that the command line is just a starting point; you can build scripts or batch files that set up environment variables and run Ant projects. Additionally, several commercial tools are available that hide the command-line interface from users, such as XMetaL Author and oXygen.

CSS changes

Changing the DITA OT CSS files used by the output XHTML takes very little time and can produce some of the most visible results.

The fastest way to make changes (hacking) is to make direct modifications to the CSS files that are a part of the DITA OT. There are two files, both in `ditaot/resource`. The file `commonltr.css` describes formatting for left-to-right languages (most Western languages, including English and European languages). The file `commonrtl.css` provides the same information for right-to-left languages. The rest of this discussion assumes you're using a Western language and modifying `commonltr.css`.

Any change you make to `commonltr.css` will affect all HTML deliverables generated by that modified DITA OT. That might be good (you want the same effect to appear in all docs processed by this copy of the DITA OT) or bad (not all docs that use this copy of the DITA OT requires this behavior). For example, if you changed

```
.dangertitle { font-weight: bold }
```

to

```
.dangertitle { font-weight: bold; color: #FF0000;}
```

all occurrences of the word “Danger” in a danger admonition, for all documents processed by that OT, will be colored red.

Adding your own CSS files

The more elegant way to modify the CSS is to create your own CSS file that overrides the CSS in the common CSS files. To specify your own CSS file, modify the Ant file that builds your deliverable (`my_xhtml.xml`). Add three new properties at the location indicated by the ❸ in the example on page 4.

```
<property name="args.css" value="${dita.dir}${file.separator}css${file.separator}garage.css"/>
<property name="args.csspath" value="CSS"/>
<property name="args.copycss" value="yes"/>
```

The `args.css` property specifies the source location of the new CSS file. In this case, it is in the folder named `ditaot/css`. The `{file.separator}` is a default property defined by Ant. It represents an OS-dependent file separator character, which is a forward slash for UNIX (/) and a backslash for Windows (\).

The `args.csspath` property indicates a relative path in the output location to the folder that will contain the CSS file. In this example, the CSS files will reside in a folder named `CSS`, which is a child of the folder containing the output HTML files. In our example, the CSS file will be copied to `ditaot/ant/out/samples/xhtml/CSS`.

The `args.copycss` property tells the DITA OT to copy the CSS file to the path specified in the `args.csspath` argument (there may be times when you don't want the file to be copied). This property is required if you want the CSS file to be copied.

Note that these Ant properties limit you to a single additional CSS file. If you need to reference multiple CSS files, use these properties to copy a “master” CSS file. Then in the master CSS file, you can use one or more `CSS @import` directives to include other CSS files:

```
@import url("group-styles.css");
@import url("company-styles.css");
```



The outputclass hack

Under normal circumstances, the DITA OT XHTML transformation outputs HTML tags, adding class attributes as the transformation dictates. If you know that your files will be used to generate HTML output, there's a further hack you can use along with modifying or specifying the CSS file.

DITA defines an attribute named `outputclass` that can be used with any DITA element. When the DITA OT processes a DITA element with the `outputclass` attribute, it automatically adds a class attribute to the corresponding output HTML tag, using the value of the `outputclass` attribute. Therefore, if you wrote:

```
<ul outputclass="square_b">...
```

The DITA output would contain:

```
<ul class="square_b">...
```

To complete this hack, modify a CSS file to support the new class. For example, you might add the following line to a CSS file so that any `` element with `output-class="square_b"` is styled with square bullets:

```
.square_b { list-style-type: square; }
```

This is particularly useful (as a hack) because it enables you to control the appearance of your output directly from a DITA attribute. A `` element without this `outputclass` attribute is rendered normally.

Again, the `outputclass` attribute only works with HTML. Note also that there is no value checking: if you or another content creator types the name of the attribute incorrectly, no error messages are generated in the DITA OT. A more elegant alternative, which would allow the DITA OT to check your input, would be to create a simple specialization that implemented a new square-bulleted `` element, perhaps `<ul-square>`. You could also create a specialization that adds a new bullet-type attribute to the `` element.

HTML headers and footers

The DITA OT allows you to specify files that contain boilerplate headers, footers, and HTML `<head>` information.

You specify the locations of these files with the Ant properties listed in Table 1.

Table 1: Ant properties for headers and footers

Property Name	Specifies
<code>args.hdr</code>	The running header for the output file.
<code>args.ftr</code>	The running footer for the output file.
<code>args.hdf</code>	Additional information for the HTML <code><head></code> tag. This is useful for specifying metadata or additional link information.

The files referenced by these properties must contain well-formed XML. If you're using HTML, your HTML must meet all strict XHTML rules, including balanced open and close tags (and), empty tags for stand-alone elements (<hr/> and
), quoted attribute values (attribute="value"), and so on.

The example shown here illustrates how to add a footer. The steps for adding a header or <head> tag information is similar, only you use a different property name and specify a different file.

To add a copyright footer to your HTML pages:

1. Create a footer file in the resource folder, named `my_footer.html`, that contains:

```
<p class="copyright_footer">
Copyright &#169; 2009 Scriptorium Publishing Services. All rights reserved.
</p>
```

2. Modify your Ant project file (`my_xhtml.xml`), adding this line near the location indicated by the ❸ in the example on page 4:

```
<property name="args.ftr"
value=" ${dita.dir}${file.separator}resource${file.separator}my_footer.html"/>
```

When the DITA OT processes a source document, this footer is inserted automatically near the end of the output:

```
<p class="copyright_footer">
Copyright © 2009 Scriptorium Publishing Services. All rights reserved.
</p>
```

Modifying XSL transforms

The next level of change you can make involves changes to the actual DITA OT XSL transforms. The changes you can make range from minor output changes to massive changes in the XSL processing.

The changes you make in CSS usually take little time, but have a big visual effect. Most of the changes you need to make to get the output looking the way you want will be in CSS. By contrast, the changes you make in XSL are often very necessary, will have much lower visual impact, but will consume a large amount of your time and effort.

The payoff is that modifications to XSL can accomplish things that cannot be done any other way. The changes you can make vary widely in scope and complexity:

- ❖ The DITA OT provides a number of empty XSL templates that are intended to handle headers, footers, and the like (these are called “stub” templates). Unlike the boilerplate headers and footers mentioned earlier in this paper, these templates allow you to incorporate information from the DITA source you're processing. For example, a template can find the most recent <title> element used in a <topic> or <section> and include it in a running header.
- ❖ Output different HTML tags to produce different results. For instance, replacing a <p> tag in HTML with a <div> tag (perhaps including a class attribute).



- ❖ Use information in the DITA content in a different order or different way from which it was originally intended. For instance, adding a summary of the subheads at the beginning of a topic, or filtering out elements based on an attribute value (conditional text).
- ❖ Add capabilities that are not in the current stylesheets. For example, adding the ability to sort items in a list (see “Modifying transforms: sorting lists” on page 10).
- ❖ At the high end of modifications you can make, you can create a specialization to support content not handled in standard DITA topics. Specializations themselves can range from adding new elements into existing topics to implementation of a new topic type.

Apart from using the stub XSL templates and *true* specialization, most of these approaches represent varying degrees of hacks.

Stub templates provided by the DITA OT

The DITA OT predefines a number of named templates that provide common functions, such as headers, footers, and side-TOCs.

These templates are in `ditaot/xsl/xslhtml/dita2htmlimpl.xsl`; each template is named `gen-user-*`. The templates are listed in Table 2. The idea is that you should copy the template to your own XSL stylesheet file and make your modifications there. However, in the spirit of hacking, until you get around to creating your own XSL file, you can make the modification directly in `dita2htmlimpl.xsl`.

Table 2: DITA OT Stub Templates

Stub template	Purpose
<code>gen-user-head</code>	Create additional information for the HTML <head> tag. You can use this template to add dynamic metadata to the output HTML.
<code>gen-user-header</code>	Generate header information.
<code>gen-user-footer</code>	Generate footer information.
<code>gen-user-sidetoc</code>	Generate a list of subordinate topics or sections.
<code>gen-user-scripts</code>	Generate JavaScripts for the document.
<code>gen-user-styles</code>	Generate style (CSS) information for the document.
<code>gen-user-external-link</code>	Handle links to external information. Note that this is the only template for which there is no <code><xsl:call-template></code> directive in the <code>dita2htmlimpl.xsl</code> stylesheet. To implement this, you'll need to modify the template that handles <code>topic/xref</code> (for HTML, usually <code>ditaot/xsl/xslhtml/rel-links.xsl</code>).
<code>gen-user-panel-title-pfx</code>	Generate a prefix for the topic title. You might use this if you need to number topics.

There is some overlap with the HTML header and footer files described earlier in this paper. The difference is that the HTML files are static; each file is included in the output stream with no further processing. The XSL stub templates allow you to generate content dynamically, based on other information, such as the content of the DITA source topic, Ant parameters (for example, a draft stamp), or system and environment information (for example, the date).

The following example illustrates modifications to the gen-user-header template (shown in **bold**). The XSL does two things: it emits the message “Using override header.” to the DITA OT log file and adds a paragraph indicating the name of the root element of the topic.

```
<!-- Override the template in dita2htmlImpl.xsl -->
<xsl:template match="/[node()]*" mode="gen-user-header">
  <xsl:message>Using override header.</xsl:message>
  <p class="mytitle">
    This is a <xsl:value-of select="name()"/>.
  </p>
</xsl:template>
```

In the command window, you should see lines such as these (the DITA OT renders the `<xsl:message>` directive as a Warning):

```
[xslt] Processing C:\DITA-OT1.4.2.1\ant\temp\tasks\washingthecar.xml to C:\DITA-
OT1.4.2.1\ant\out\samples\html\tasks\washingthecar.html
[xslt] file:C:/DITA-OT1.4.2.1/xsl/xslhtml/dita2htmlImpl.xsl:4467:14: Warning! Using override header.
```

Each output file will contain something like:



The line “This is a concept.” comes from the new header.

“Creating an override template” on page 13 describes how to formalize this change.

Combining this concept with the CSS concepts described earlier in this paper, you could add `mytitle` to a CSS file so that the header stands out more clearly.

Element matching in the DITA OT

Before looking further at XSL stylesheets in the DITA OT, it’s important to understand how the DITA OT matches XML elements.

If you are familiar with XSL stylesheets, you know that a `<xsl:template>` declaration includes an XPath expression in its `match` attribute. Similarly, an `<xsl:apply-templates>` directive often uses a `select` attribute that also contains an XPath expression. The XPath expressions used in the `match` and `select` attributes usually match the name of a specific element.



Therefore, a template that handles the element named `javaClass` might begin with:

```
<xsl:template match="javaClass">
```

However, the DITA OT requires a slightly different strategy. Because each layer of specialization inherits behaviors from the more general layers above it, each element usually contains information listing each of those previous layers. To implement this behavior, a preprocessing stage in the DITA OT creates a class attribute for every element in a DITA source file. The new class attribute contains a space-separated list of topic-name/element-name pairs. For example, the `JavaAPIRef` specialization is a further specialization of the `APIRef` and `Reference` specializations, which both are specializations of `Topic`. As a result, the class attribute for a `javaClass` element is:

```
class=" - topic/topic reference/reference apiRef/apiRef apiClassifier/apiClassifier javaClass/javaClass "
```

To match elements, the DITA OT uses the `contains()` function to determine if an element's class attribute contains the appropriate topic-name/element-name pair. Therefore, a template that matches a `javaClass` element begins:

```
<xsl:template match="*[contains(@class,' javaClass/javaClass ')]">
```

```
...
```

That is, match all elements where the class attribute contains “ `javaClass/javaClass` ”. The class attribute is useful because this same element can also be matched by:

```
<xsl:template match="*[contains(@class,' topic/topic ')]">
```

And anything in between.

NOTE: The spaces between the single quote and the beginning or end of the topic-name/element-name pair are important (for example, ‘ `javaClass/javaClass` ’ and ‘ `topic/topic` ’), because they prevent mis-matches of the topic-name and element-name. Without the spaces, the function “ `contains(@class,'javaClass/javaClass')` ” could also match `javaClass/javaClassDetail` in another class attribute, which is incorrect.

Modifying transforms: sorting `` lists

The following example enables the DITA OT to sort the contents of an unordered list. (Perhaps the content includes a list of terms that must be presented alphabetically.) The file `ditaot/samples/concepts/tools.xml` contains an ideal list for this example: an unordered list that itemizes a set of tools. By applying this change, you can sort the list.

This modification takes advantage of another hook for hacking the DITA OT: the `otherprops` attribute is defined for all DITA elements. It's useful because its definition is left entirely open. The only caution about using it is to make sure that no one else modifying the same toolkit is using the `otherprops` attribute for their own purposes.

To start, edit `tools.xml` and add an `otherprops` attribute to the `` element (shown in **bold**):

```
...
Useful tools include the following items:</p>
<ul otherprops="sort">
  <li>Hammer</li>
  <li>Screw driver set</li>
...
```

The next problem is determining where is handled. The *DITA Language Reference*¹ indicates that ul is defined by the generic topic, so the template will match on “ topic/ul ”. A quick search of the ditaot/xsl folder shows that the template to modify is in ditaot/xsl/xslhtml/dita2htmlImpl.xsl.²

Before doing anything else, make a backup copy of this file.

After finding the template to modify, add an <xsl:choose> directive (shown in **bold**) to detect the value “sort” in the otherprops attribute and implement the new behavior:

```
<xsl:template match="*[contains(@class,' topic/ul ')]" mode="ul-fmt">
...
<xsl:apply-templates select="@compact"/>
<xsl:call-template name="setid"/>
<xsl:choose>
  <xsl:when test="contains(@otherprops,'sort')">
    <xsl:message>SORTING!!!!</xsl:message>
    <xsl:apply-templates>
      <xsl:sort />
    </xsl:apply-templates>
  </xsl:when>
  <xsl:otherwise>
    <xsl:apply-templates/>
  </xsl:otherwise>
</xsl:choose>
</ul>
...
```

Run the DITA OT:

```
ant -f ant/my_xhtml.xml
```

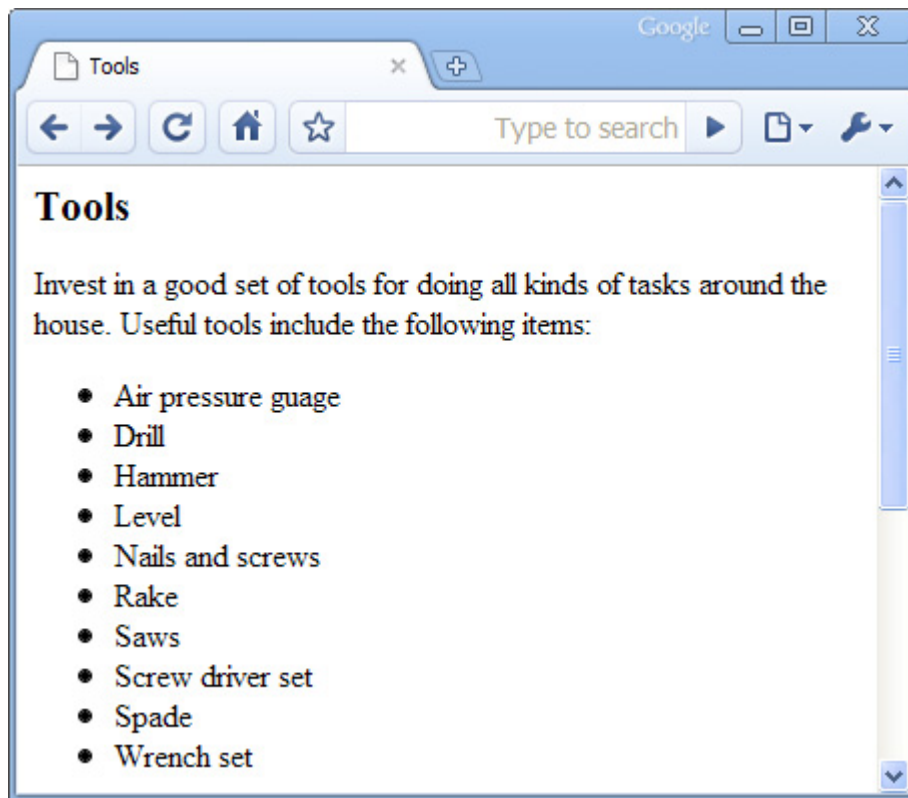
When the DITA OT processes tools.xml, it displays the message indicating that the sort is occurring:

```
...
[xslt] Processing C:\DITA-OT1.4.2.1\ant\temp\concepts\tools.xml to C:\DITA-
OT1.4.2.1\ant\out\samples\xhtml\concepts\tools.html
[xslt] file:/C:/DITA-OT1.4.2.1/xsl/xslhtml/dita2htmlImpl.xsl:1155:23: Warning! SORTING!!!!
...
```

Open the file ditaot/ant/out/samples/xhtml/concepts/tools.html in a browser to see the sorted list.

-
1. The *DITA Language Reference* is included in the DITA OT files in ditaot/doc/ditaref-book.chm or ditaref-book.pdf. The DITA sources are in ditaot/doc/langref.
 2. Working with the DITA OT would be almost impossible without a good search tool. I use AgentRansack, a free version of FileLocator Pro, to search for strings in Windows files and directories. For more information, see <http://www.mythicsoft.com/agentransack/>.



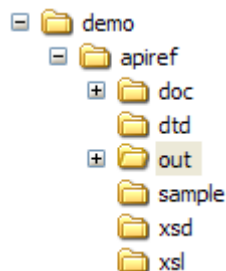


Packaging up your overrides

At some point you may want to be able to redistribute your customizations. When this time comes, you create a DITA OT plugin.

The DITA OT plugin architecture allows you to modify the files used in the DITA OT transformation including the DTDs, the XSL stylesheets, and the Ant project files that control the transformation. Behind the plugin architecture are a number of helper transforms that actually modify the key DITA OT files.

A good way to learn how to create a plugin is to look at an existing plugin. The APIRef plug-in is an excellent place to start (you can download this plugin from [SourceForge](#)). If you download and install the plugin, you'll see that it adds a new apiref folder to the demo folder (for historical reasons, most plugins are added to the demo folder, rather than the plugin folder). Note how the folders in the apiref folder mirror the folders in the ditaot folder.



Creating an override template

Before building the plugin, you must separate your hacked changes from the DITA OT XSL stylesheets and create your own stylesheet file.

To create an override template for the list sorting example shown earlier:

1. Create a new XSL stylesheet file to contain the template. In this example, the new file is called `sort-ul.xml`.
2. Edit the new stylesheet file and add the standard XML required for an XSL stylesheet:

```
<?xml version="1.0" encoding="UTF-8" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

```
</xsl:stylesheet>
```

3. Open the file `ditaot/xsl/xslhtml/dita2htmlimpl.xml`, and find the template that contains the new sorting code. The template begins with:

```
<xsl:template match="*[contains(@class,' topic/ul ')]" mode="ul-fmt">
```

4. Copy the entire template into the new stylesheet. Add a comment at the beginning of the copied template, indicating that the template overrides the behavior of `dita2htmlimpl.xml`. Then save and close the file.

The plugin mechanism will invoke the new stylesheet, so your modified template overrides the behavior of the DITA OT version of the template.

5. Restore the old DITA OT template to its previous state. (You did make a backup copy, right?)

Creating a plugin

To create a plugin:

1. Create your XSL, DTD, and Ant files, as appropriate for your changes. In this example, there is only one XSL file containing the template that handles list sorting.
2. Create a new folder for your plugin in the `ditaot/demo` folder.
3. Create folders inside the `ditaot/demo` folder that mirror the `ditaot/demo/apiref` plugin folder. (Actually, you only need folders for the components you're distributing. Therefore, if you're only updating the XSL, you only need to create an `xsl` folder.)
4. Copy the new stylesheet file (`sort-ul.xml`) into the `xsl` folder.
5. Copy the file `plugin.xml` from the `apiref` folder. (The existing file is a good place to start because you can take away what you're not going to use, then make minor tweaks to point to your own files.)
6. Open the `plugin.xml` file in a text editor. It should look like this:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!--
| (C) Copyright IBM Corporation 2005 - 2006. All Rights Reserved.
*-->
<plugin id="org.dita.specialization.apiref">
<feature extension="dita.specialization.catalog"
value="catalog-dita.xml" type="file"/>
```



```
<feature extension="dita.xsl.xhtml"
value="xsl/apiref2xhtml.xsl" type="file"/>
<feature extension="dita.conductor.target"
value="conductor.xml" type="file"/>
</plugin>
```

7. Delete the first and last `<feature>` elements, leaving only the `<feature>` element with `extension="dita.xsl.xhtml"`.
8. Change the value in the remaining `<feature>` element to point to your new XSL file.

```
<feature extension="dita.xsl.xhtml" value="xsl/sort-ul.xsl" type="file"/>
```

To specify multiple XSL stylesheet files, add them to the value attribute of the `<feature>` element, separated by commas:

```
<feature extension="dita.xsl.xhtml" value="xsl/sort-ul.xsl,xsl/my_heads.xsl" type="file"/>
```

Installing and testing the plugin

Since DITA OT 1.2, you do not have to run a separate installation step. When you run the DITA OT, it looks for plugins and integrates them, if necessary. To install and test your plugin:

1. Install a new copy of the DITA OT. Rename the folder to something like `ditaot-test`.
2. Copy your plugin folder to the demo folder of the new `ditaot-test` folder.
3. Copy the modified version of `ditaot/samples/concepts/tools.xml` (the one that includes the `otherprops="sorted"` attribute) to the folder `ditaot-test/samples/concepts`.
4. Copy the file `ditaot/ant/my_xhtml.xml` to the folder `ditaot-test/ant`.
5. Use the `startcmd.bat` or `startcmd.sh` file in the `ditaot-test` folder to start a command line.
6. Run the DITA OT with the command:

```
ant -f ant/my_xhtml.xml
```

7. Open the file `ditaot/ant/out/samples/xhtml/concepts/tools.html` in a browser to see the sorted list.

Hacking and specialization

The most complex type of modification you can make to the DITA OT is specialization. A strict interpretation of the DITA OT documentation would hold that “specialization” means creating an entirely new topic type based on an existing topic type. However, in the spirit of hacking the DITA OT, it is possible to change, add, or remove any of the sub-elements within a given topic type.

Specialization provides support for content that isn’t handled in standard DITA topic specializations (concept, task, and reference). Specializations themselves can range from adding new elements to existing topics all the way up to the complete implementation of a new topic type.

The reasons for modifying or specializing the DITA OT include:

- ❖ Refining a set of elements. You might want to enforce a particular order for a set of elements.
- ❖ Defining new elements that clarify or better identify certain pieces of content. The example shown below involves adding an ingredient list to the `<prereq>` element in a task topic.

- ❖ Greater semantic accuracy. Specialization allows you to attach specific names to elements. These semantic elements will aid your content creators in making decisions about what goes where.
- ❖ Providing formatting information. Sometimes the organization of content doesn't follow neat rules that can be codified in the XSL transforms. Sometimes a human needs to indicate where something happens, which cannot be deduced automatically.

It is possible to create most aspects of a specialization by modifying the existing files in the DITA OT. As your work progresses, you can move your changes to a separate set of files and create a DITA plugin. When you get to the level of creating new topic types, you usually need to create dtd files that reflect your topic type, and unless you're interested in diving down into the inner workings of DITA OT, it's usually best to add these with a plugin.

Building a specialized structure

Because specialization affects the structure of your topics, it means that you will need to modify your DTDs. As described before in this paper, there are a number of levels at which you can change your DTDs. The hacker's solution is to modify existing dtd files (mod files, actually³). There are more formal strategies in which you create a new set of dtd and mod files and add them to the DITA catalog. Unfortunately there isn't room to discuss catalog modifications in this paper.

The following example adds a new <ilist> (ingredient list) element to the task topic <prereq> element. The *DITA Language Reference* indicates that <prereq> contains text data, a number of standard in-line elements, and several block elements, such as <p>, , and . We will specialize the element to create our <ilist> element. The <ilist> can contain one or more <ingr> (ingredient) element; <ingr> is an element specialization of the element.

NOTE: When you specialize an element, the content model for the new element can only contain elements, or specializations of the elements, that are in the content model of the original element. That is, you cannot add new element types that do not exist in the content model of the original element.

To find out where to modify the structure, search the ditaot/dtd folder for the file in which <prereq> is defined. The file is task.mod.

There are three areas in task.mod that require changes: adding entity declarations for the new elements, modifying and adding new content models, and adding inheritance information about the new elements.

Near the top of the file is an area labeled "ELEMENT NAME ENTITIES." Find this section and scroll to the last of the entity declarations. After the last entity declaration, add a comment indicating the purpose of the modifications, and then add entity declarations for the two new elements:

```
<!-- Added new elements for ingredient lists -->
<!ENTITY % ilist    "ilist"          >
<!ENTITY % ingr    "ingr"          >
```

NOTE: The percent sign (%) in the entity declaration indicates this is a parameter entity, which is used only in the DTD.

3. The dtd files actually define the domain integration and then point to the mod files, where the structure is formally defined.



It's a good idea to add your changes at the end of each section in the dtd and mod files. That way you can locate your changes quickly.

A number of lines below the entity declarations are the element declarations. Scroll down until you find the declaration for the <prereq> element.

```
<!ELEMENT prereq    (%section.notitle.cnt;)*    >
```

Copy and paste the declaration line and comment one of the copies (that way you can remember what you changed). Modify the other declaration, adding the changes shown here in **bold** (the line that is commented out is also shown).

```
<!-- <!ELEMENT prereq    (%section.notitle.cnt;)*    > -->
<!ELEMENT prereq    ( %section.notitle.cnt; | %ilist; )*    >
```

Note that a more sophisticated change would entail finding the definition of section.notitle.cnt and adding the <ilist> element to that definition.

At the end of the other element declarations, add the content model declarations for the <ilist> and <ingr> elements:

```
<!--          LONG NAME: Ingredient List          -->
<!ELEMENT ilist    (%ingr;)+                    >
<!ATTLIST ilist
    %univ-atts;
    outputclass    CDATA          #IMPLIED    >
```

```
<!--          LONG NAME: Ingredient Item          -->
<!ELEMENT ingr    (%listitem.cnt;)*            >
<!ATTLIST ingr
    %univ-atts;
    outputclass    CDATA          #IMPLIED    >
```

These declarations were modeled on declarations of and found in commonElements.mod. Because <ingr> and both use the entity listitem.cnt for their content, the <ingr> element can contain the same content elements as the element.

At the bottom of the file, under "SPECIALIZATION ATTRIBUTE DECLARATIONS," add inheritance information for the new elements. Again, make your changes at the end of these declarations, so that you can find them more easily.

```
<!-- Inheritance for ingredient list -->
<!ATTLIST ilist    %global-atts; class CDATA "- topic/ul task/ilist "    >
<!ATTLIST ingr    %global-atts; class CDATA "- topic/li task/ingr "    >
```

Because <ilist> inherits from and <ingr> inherits from , it's not necessary to create an XSL template that handles <ilist> and <ingr>. The DITA OT will process this list as if it were a standard element.

Testing the modification

To test the modification, you need a file that employs the <ilist> element.

Use the following XML to create a new file in ditaot/samples/tasks named bubbleformula.xml:

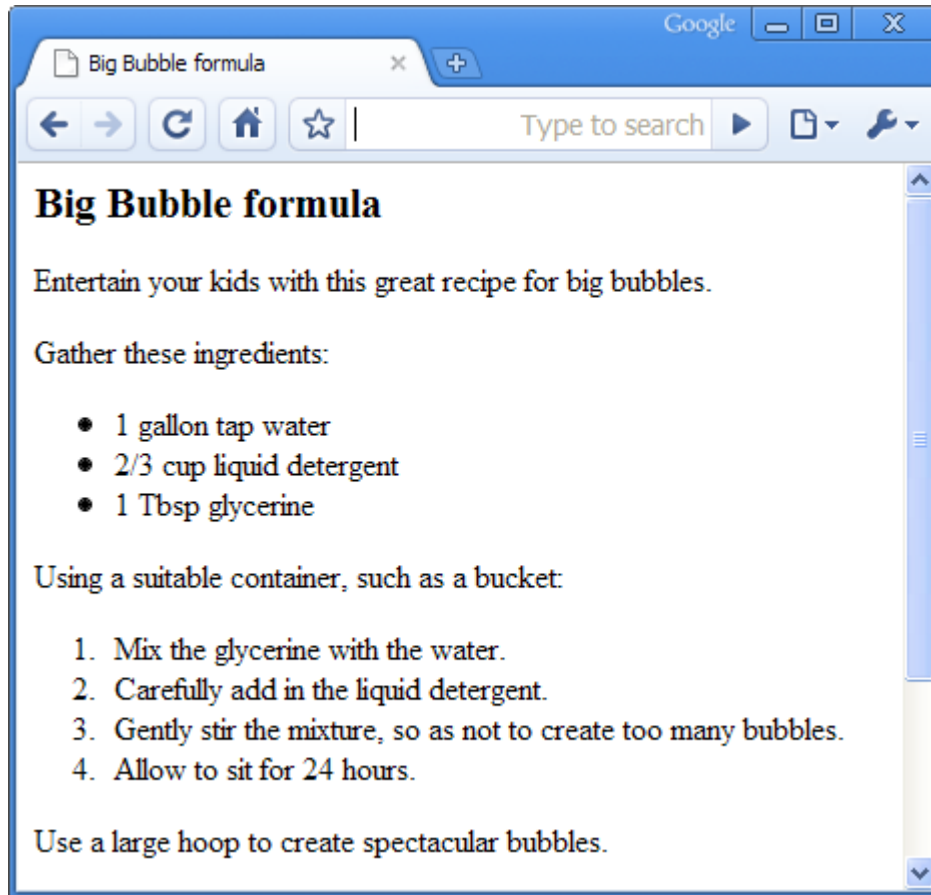
```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE task PUBLIC "-//OASIS//DTD DITA Task//EN"
"../dtd/task.dtd">
<task id="bubbles" xml:lang="en-us">
  <title>Big Bubble formula</title>
  <shortdesc>Entertain your kids with this great recipe for big bubbles.</shortdesc>
  <taskbody>
    <prereq>Gather these ingredients:
    <ilist>
      <ingr>1 gallon tap water</ingr>
      <ingr>2/3 cup liquid detergent</ingr>
      <ingr>1 Tbsp glycerine</ingr>
    </ilist></prereq>
    <context>
      <p>Using a suitable container, such as a bucket:</p>
    </context>
    <steps>
      <step><cmd>Mix the glycerine with the water.</cmd></step>
      <step><cmd>Carefully add in the liquid detergent.</cmd></step>
      <step><cmd>Gently stir the mixture, so as not to create too many bubbles.</cmd></step>
      <step><cmd>Allow to sit for 24 hours.</cmd></step>
    </steps>
    <postreq>Use a large hoop to create spectacular bubbles.</postreq>
  </taskbody>
  <related-links>
    <link href="../concepts/waterhose.xml" format="dita" type="concept">
      <linktext>Water hose</linktext>
    </link>
  </related-links>
</task>
```

Edit ditaot/samples/hierarchy.ditamap and add a new <topicref> element that references the new bubbleformula.xml file (shown in bold):

```
...
<topicref href="tasks/washingthecar.xml" type="task"/>
<topicref href="tasks/bubbleformula.xml" type="task"/>
</topicref>
```



Run the DITA OT and check the log file for errors. Now you can view your specialized topic:



Now you can apply all that you've learned in this paper to create CSS and XSL transform rules to handle `<ulist>` and `<ingr>` uniquely.

Summary

Although the default XHTML output from the DITA OT is quite plain, there are many ways to customize its appearance so that it looks the way you want it. The most dramatic changes can be achieved by modifying the CSS. Creating custom headers and footers adds a note of consistency to your documents.

Modifying the XSL transforms can add dynamic changes that are driven by many different external and internal influences. To make your DITA OT changes much more portable, create a DITA plugin. Specializations help you to make your topics much more content specific.

Now, get in there and hack!

About the author

With over 30 years experience in technical publications, Simon Bate has acquired extensive knowledge in writing, managing, production, book design, template design, and document conversions of all sorts. Simon takes great delight in programming and scripting. His motto is: “let the computer do the work.” He also enjoys teaching and sharing his knowledge with others.

Simon divides his time at Scriptorium between tools development and training. He has worked on a number of DITA OT projects, including transforming XML documents to DITA, modifying or hacking the DITA Java API Reference specialization, and creating XSL:FO output transforms for the DITA Java API Reference specialization.

About Scriptorium

Scriptorium Publishing provides expert advice on how to develop, deploy, and manage content. Our typical customer has thousands of pages of information, which needs to be delivered in print, PDF, HTML, and other media, often in dozens of languages. Our mission is to automate formatting and production tasks, usually through XML technologies, so that authors can write more efficiently.

Our consultants have experience in traditional publishing workflows, including typesetting, book design, copyfitting, and production editing. This understanding influences our approach to creating state-of-the-art publishing systems with modern tools and technologies, such as XML, HTML, DITA, the DITA Open Toolkit, XSLT, XSL-FO, FrameMaker, Ant, Perl, FrameScript, Flash, InDesign, XMetaL, oXygen, and many more.

Our customers include federal and state government as well as companies in defense, consumer electronics, telecommunications, health care, pharmaceutical, and other industries.

If you are facing a difficult publishing challenge, we want to hear from you. Contact us after the conference at info@scriptorium.com or 919-481-2701 x105.

Scriptorium Publishing is based in the Research Triangle area of North Carolina and has been in business since 1997.

Scriptorium Publishing Services, Inc.
PO Box 12761
Research Triangle Park, NC 27709-2761
USA
info@scriptorium.com
919-481-2701
www.scriptorium.com

