

DITA Open Toolkit 1.5.4

Contents

List of Figures.....	7
List of Tables.....	9
 Chapter 1: Getting Started.....	 11
Introduction to the DITA Open Toolkit and Ant.....	12
Overview.....	12
About DITA-OT Toolkit Roles.....	12
What is Ant?.....	13
What is DITA-OT?.....	13
When Should I Use DITA-OT?.....	13
Running DITA Open Toolkit.....	14
Generating Documents with Ant.....	14
Writing Ant Build Files DITA-OT.....	15
Generating Documents with command-line tool.....	17
Debugging DITA-OT Transformations.....	17
Introducing Document Generation.....	17
Best Practices.....	19
 Chapter 2: DITA Open Toolkit User Guide.....	 21
Release notes.....	22
General Enhancements and Changes.....	22
Migration from previous releases.....	23
SourceForge trackers.....	24
DITA 1.2 Specification Support.....	26
Distribution packages.....	27
Installing the DITA Open Toolkit.....	28
Installing DITA-OT Full Easy Install package in Windows.....	28
Installing DITA-OT Full Easy Install package in Linux or OS X.....	28
Set up DITA Open Toolkit in Windows.....	29
Set up DITA Open Toolkit in Linux or OS X.....	30
Using DITA transforms.....	32
Running DITA-OT from Ant.....	33
Running Ant.....	34
Ant tasks and scripts.....	34
Ant argument properties for DITA-OT.....	35
Running DITA-OT from command-line tool.....	43
Command-line tool arguments for DITA-OT.....	44
Configuration.....	50
Available DITA-OT Transforms.....	50
DITA to XHTML.....	51
DITA to Eclipse help.....	51
DITA to TocJS.....	51
DITA to PDF (PDF2).....	51
DITA to HTML Help (CHM).....	51
DITA to ODT (Open Document Type).....	52
DITA to Docbook.....	52
DITA to Troff.....	52

DITA to Word output transform.....	52
DITA to Eclipse Content.....	53
DITA to legacypdf (Deprecated).....	53
Migrating HTML to DITA.....	53
Problem determination and log analysis.....	57
Troubleshooting.....	57

Chapter 3: Developer Reference.....59

DITA Open Toolkit Architecture.....	60
Processing modules in the DITA-OT.....	61
Processing order within the DITA-OT.....	62
DITA-OT pre-processing architecture.....	63
Generating XHTML with navigation.....	67
PDF output pipeline.....	68
ODT Transform type (Open Document Format).....	69
Extending the DITA Open Toolkit.....	69
Installing DITA-OT plug-ins.....	70
Creating DITA-OT plug-ins.....	70
Plug-in configuration file.....	71
Extending the XML Catalog.....	72
Adding new targets to the Ant build process.....	72
Adding Ant targets to the pre-process pipeline.....	73
Integrating a new transform type.....	74
Override styles with XSLT.....	74
Adding new generated text.....	75
Passing parameters to existing XSLT steps.....	76
Adding Java libraries to the classpath.....	77
Adding diagnostic messages.....	77
Managing plug-in dependencies.....	78
Version and support information.....	79
Creating a new plug-in extension point.....	79
Example plugin.xml file.....	80
Implementation dependent features.....	80
Extended functionality.....	81
Code reference processing.....	81
Topic merge.....	81
Creating Eclipse help from within Eclipse.....	83

Appendix A: DITA-OT release history.....97

DITA Open Toolkit Release 1.5.4.....	98
General Enhancements and Changes.....	98
Migration from previous releases.....	99
SourceForge trackers.....	100
DITA Open Toolkit Release 1.5.3.....	102
DITA Open Toolkit Release 1.5.2.....	106
DITA OT Release 1.5.1.....	108
DITA OT release 1.5.....	110
DITA OT release 1.4.3.....	113
DITA OT release 1.4.2.1.....	114
DITA OT release 1.4.2.....	114
DITA OT release 1.4.1.....	115
DITA OT release 1.4.....	116
DITA OT release 1.3.1.....	118
DITA OT release 1.3.....	118

DITA OT release 1.2.2.....	120
DITA OT release 1.2.1.....	121
DITA OT release 1.2.....	121
DITA OT release 1.1.2.1.....	123
DITA OT release 1.1.2.....	123
DITA OT release 1.1.1.....	124
DITA OT release 1.1.....	124
DITA OT release 1.0.2.....	125
DITA OT release 1.0.1.....	126
DITA OT release 1.0.....	126
DITA history on developerWorks (pre-Open Source).....	126

Appendix B: Project Management..... 129

Goals and objectives of the DITA Open Toolkit.....	130
DITA Open Toolkit Development Process.....	130
Roles and Responsibilities.....	130
Process life cycle.....	131
Procedures.....	131
DITA Open Toolkit Contribution Policy.....	132
DITA-OT Contribution Questionnaire Form 1.2.....	133

Appendix C: developerWorks articles..... 137

Introduction to the Darwin Information Typing Architecture.....	138
Executive summary.....	138
DITA overview.....	140
DITA delivery contexts.....	141
DITA typed topic specializations (infotyped topics).....	141
DITA vocabulary specialization (domains).....	142
DITA common structures.....	143
Elements designed for specialization.....	143
The values of specialization.....	143
Specializing topic types in DITA.....	144
Architectural context.....	145
Specializing information types.....	145
Specialization example: Reference topic.....	146
Specialization example: API description.....	148
Working with specialization.....	150
Specializing with schemas.....	153
Summary.....	154
Appendix: Rules for specialization.....	154
Specializing domains in DITA.....	157
Introducing domain specialization.....	157
Understanding the base domains.....	158
Combining an existing topic and domain.....	159
Creating a domain specialization.....	161
Considerations for domain specialization.....	164
Generalizing a domain.....	164
Summary.....	165
How to define a formal information architecture with DITA map domains.....	166
Formal information architecture.....	166
Specializing topics and maps.....	167
The how-to collection.....	167
Map specialization.....	168
Implementing a map domain.....	168

Declaring the map domain entities.....	168
Defining the map domain module.....	168
Assembling the shell DTD.....	170
Creating a collection with the domain.....	170
Summary.....	171

List of Figures

Figure 1: Layers in the Darwin Information Typing Architecture.....	140
Figure 2: Relationship of specialized communities to the base architecture.....	144
Figure 3: A more specialized information type, API description.....	149

List of Tables

Table 1: Common DITA-OT parameters.....	35
Table 2: XHTML and related parameters.....	37
Table 3: PDF parameters.....	39
Table 4: ODT related parameters.....	40
Table 5: EclipseContent properties.....	41
Table 6: Properties for the "xhtml" transform type.....	41
Table 7: Common DITA-OT parameters.....	44
Table 8: XHTML and related parameters.....	45
Table 9: PDF parameters.....	47
Table 10: ODT related parameters.....	48
Table 11: EclipseContent options.....	48
Table 12: Options for the "xhtml" transform type.....	48
Table 13: Table of supported parameters.....	54
Table 14: Table of supported parameters.....	56
Table 15: Relationships between topic and a specialization based on it.....	147
Table 16: Summary of APIdesc specialization.....	149
Table 17: Summary of specialization rules.....	154

Chapter

1

Getting Started

Topics:

- *[Introduction to the DITA Open Toolkit and Ant](#)*
- *[Running DITA Open Toolkit](#)*
- *[Debugging DITA-OT Transformations](#)*
- *[Best Practices](#)*

Introduction to the DITA Open Toolkit and Ant

Understanding the role of DITA Open Toolkit and Ant for technical writers.

This chapter further describes the role of DITA Open Toolkit, Ant, and when DITA-OT is a logical choice for your documentation team.

Overview

Introducing the most important authoring tool since FrameMaker.

DITA-based writers gain numerous advantages using this powerful, single-source, document solution. Indeed, I think of DITA as the Holy Grail of technical documentation, the object for which many a manager has sent me on long, fruitless searches in the past, only to return with half-hearted recommendations for a combination of tools that required hours of manual tweaking to reproduce a document in just one alternative format.

Today, I can tell my manager that the Grail has been found, and I can produce a handful of different document output types simultaneously. This is a breakthrough technology for technical writers. In industry jargon, "single source" has previously meant writing in FrameMaker, then importing your source into another expensive application to produce a second output format, typically online help. To gain just a second format from the source often required tedious hours, sometimes days, massaging the text after it had been "translated" into an online help system. DITA-OT allows technical writers to produce seven output formats at the same time.

However, several technologies make this magic happen. Motivated, or just curious, writers will want a more advanced understanding of what makes DITA tick. To do so, you will need to learn how to write Ant build scripts for the DITA-OT and invoke them from the command line.

- [Ant](#)
- [DITA Open Toolkit](#)

Hopefully, this guide will motivate you to study DITA-OT further and encourage your publications team to implement a single-source, DITA-based documentation solution.

About DITA-OT Toolkit Roles

Descriptions of the following user roles for the toolkit: CSS Customizers, Build Scripters, and potentially End Users of third-party authoring tools.

If you don't know what a build script is, or whether your authoring tool should or does use DITA OpenToolkit, you may be reading the right document in the in the DITA-OT documentation suite. User Roles are a useful way to present and approach the documentation, enabling the reader to bypass information that is not relevant to the specific task she is performing when this document is read. The followige roles are addressed in this guide:

- CSS Stylesheet Customizers
- Ant Build Scripters
- Online Help developers

Many readers may be unaware of the toolkit's presence, but it is the "engine" of whatever writing tool you use to write and maintain XML-based, DITA-compliant content. CSS Stylesheet Customizers read this guide when they are unable to specify custom stylesheets, or .css files from their third-party authoring application. Build Scripters consult this guide when the toolkit is unable to process an Ant build script. The Quick Start Guide features an up-to-date tables of all supported parameters for both Ant and the Java command line interface, useful information if your documentation is generated automatically as part of an automated daily build. As my first manager warned me long, long ago, 'The howling hordes descend when the build breaks'.

If you need to fix a broken build, time is of the essence. If you know how to edit CSS stylesheessor run Ant build scripts, then read on. If you don't recognize CSS and Ant, you're likely an End User and you should refer to your third-party documentation. If like the author, you also enjoy hacking with your tools this is definitely the guide for you. Although the Quick StartGguide currently provides more information for build scripters than css customizers, revisions will describe the use of XSLT and the new plugin architecture for this audience. When your third-party authoring tool breaks down, this guide is the mechanic's manual you didn't know you had. If you're unafraid of a

command line and willing to "get your hands dirty" under the hood, an up-to-date, and maintained table of Ant build properties can be the difference between a broken build and a bad day or a gently purring build in a background `chron` job that never demands attention. If the build has already broken by the time you read this, you may be the "build hero" who magically "fixes" the build which no one cares about, except that it is usually only then that the fire-breathing IT dragon returns to its cave and everyone can relax again until the next "emergency".

What is Ant?

Learning about the blurred line between code and documentation and why technical writers need to learn Ant.

If your DITA authoring tool uses the Open DITA Toolkit to generate your documents, you're already using Ant. So, what is Ant, anyway? Ant is a *build tool*, a program used to compile other programs. If you work as a writer in the enterprise software industry, you know that software engineers regularly produce several versions of whatever software they are working on before they release it to the public. Each compilation is called a *build*. Dozens, sometimes hundreds, of builds are compiled before the RTM (release to manufacturing) or GA (general acceptance) build is certified as the official release build. You can often determine the release build of whatever software you are using by reading the Help->About dialog box. For example, my version of XMetal is 5.5.0.219. This means that build 219 was the official release build for XMetal, version 5.5.

Writers also draft, write, revise, and rewrite their documents many times before releasing a document to the public. We tend to call these drafts, rather than builds. You probably saved drafts of your documents in a document repository or CMS, a content management system, in the past, but I doubt you thought of your draft, even though it was versioned by the repository, as a software build that either compiled or failed to compile. A successful document "build" meant only that a document opened in your authoring tool the next day, not that all the related documents also opened successfully and "compiled" together to produce a version, albeit incomplete, of the documentation that will eventually make its way to your readers. Hence, the "build" metaphor did not extend beyond the programming code in the engineers' cubicles to the documents crafted by the writers.

DITA changes that forever; the build metaphor is as relevant to you as to the engineers. Behind the user interface of your authoring tool, the DITA Open Toolkit uses Ant to compile a build every time you try to generate your single-source documents. If you want to customize the way DITA-OT generates your documents, you will need to open the hood, so to speak, and get your hands dirty with the internals of the Toolkit and Ant.

What is DITA-OT?

What is the DITA Open Toolkit, anyway?

The DITA Open Toolkit is a popular, free, open-source tool used to transform DITA documents and maps into the output document formats you desire. In fact, most of the proprietary authoring tools use the DITA-OT to transform DITA documentation projects, so you aren't wasting your time learning about how it works. Many errors are more quickly fixed if you understand what the toolkit is doing "beneath the hood" of your authoring tool.

DITA-OT uses Ant to generate your documents. The primary ant script is `build.xml`, which imports several other build scripts to initialize, validate, and transform your `.dita` documents.

See [Introducing Document Generation](#) on page 17 for more information about how DITA-OT processes documents.

When Should I Use DITA-OT?

Determining when DITA OT makes sense for your team.

There are several scenarios where using DITA-OT is the appropriate choice for your documentation team, and describing them all is beyond the scope of this guide. However, here are three simple criteria where DITA-OT provides the best solution:

- Your documentation suite contains a lot of content that is reusable for different documents and audiences.
- Your documentation suite contains documentation for developers using the Eclipse IDE.
- Your documentation suite includes both Microsoft HTML Help and PDF-based documents.

DITA-OT is the only publication tool capable of producing both PDF and Eclipse-based documentation from the same source. Eclipse is the industry-standard IDE for many Java developers, and DITA-OT generates the content

and plugin file required for this environment. If your developer audience uses Eclipse, you can easily add IDE-specific online help to your documentation suite.

If your primary audience is end users, rather than software developers and system administrators, you likely need to provide Microsoft HTML Help for them, in addition to PDF-based documentation. DITA-OT is the only tool that produces both from the same source files.

Running DITA Open Toolkit

Learn how to run DITA-OT to generate published output documents.

Generating Documents with Ant

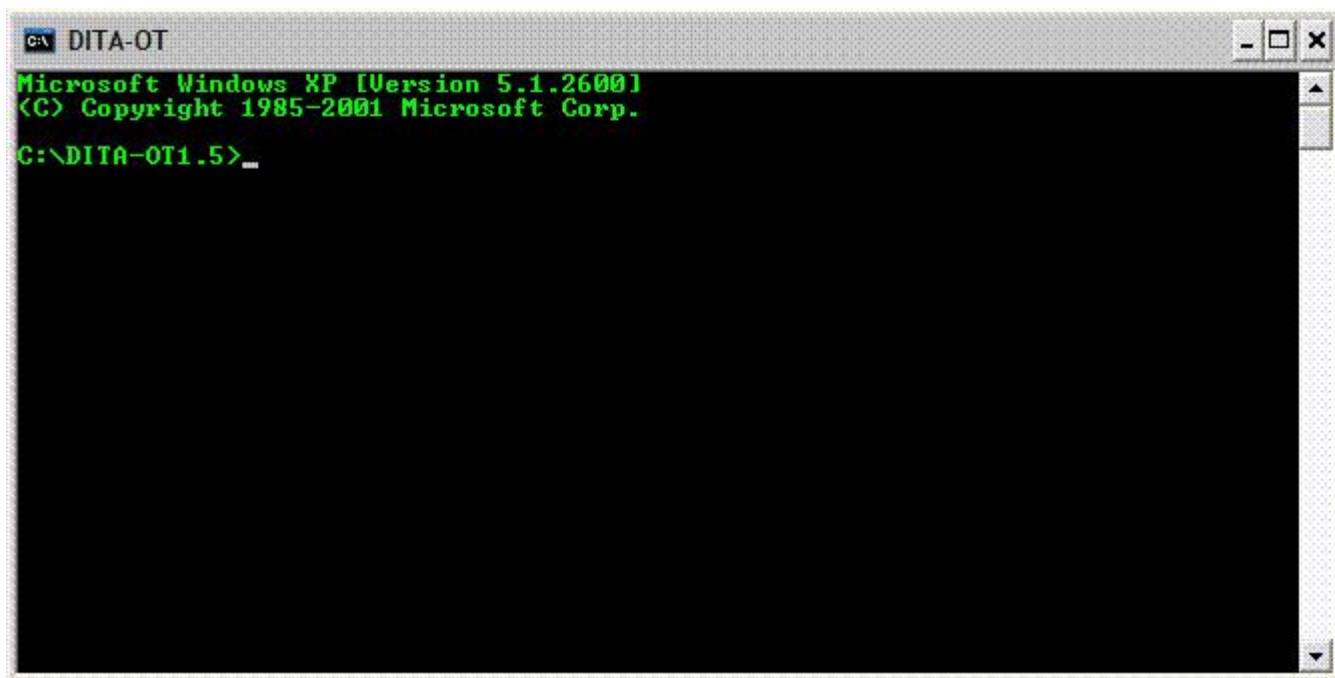
How to generate documents from the command line with Ant.

1. Open a command prompt.
2. Change directories to where the DITA-OT is installed on your machine.
3. Set up the processing environment.

Enter the following command:

```
startcmd.bat
```

Another command prompt appears with DITA-OT in the title bar, as shown in the following figure:



4. Run a conversion to a transformation output type.

Enter the following command and press the Enter key:

```
ant -Dargs.input=source -Dtranstype=transtype
```

The following table describes this command.

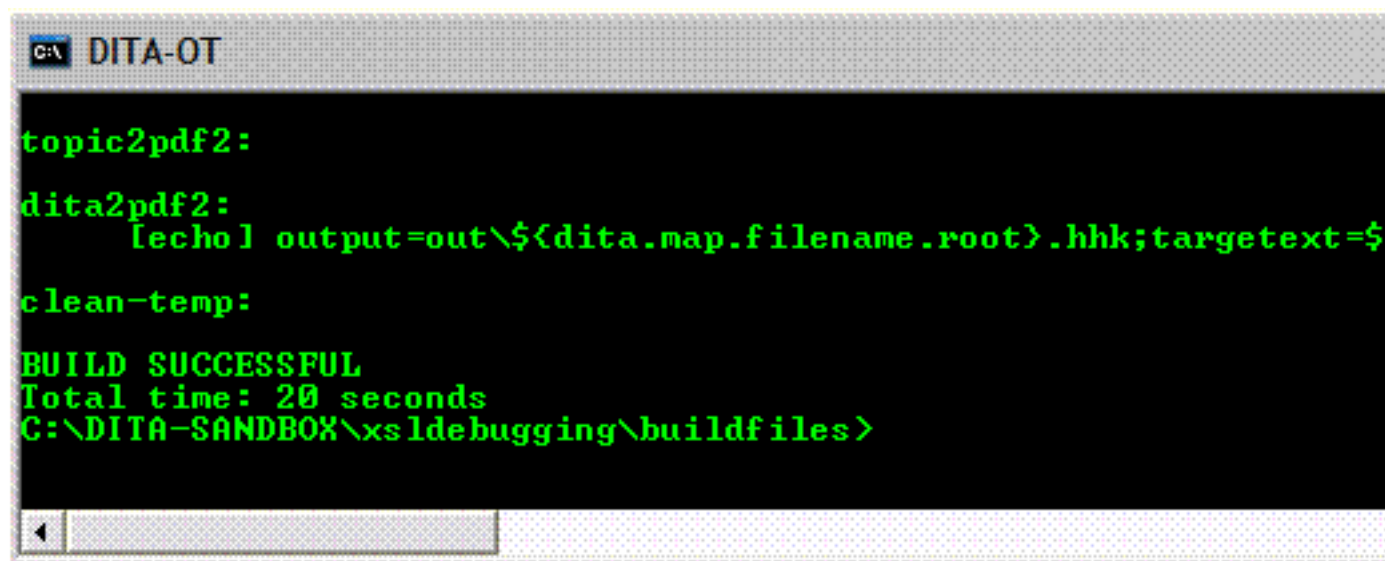
Syntax	Description
ant	Starts the Ant build tool installed as part of DITA-OT.

Syntax	Description
<code>-Dargs.input=source</code>	The <code>source</code> specifies the DITA topic or map to process. If the build file is not in the current directory, you must specify the path to the file.
<code>-Dtranstype=format</code>	The <code>format</code> specifies the transformation output type to generate.

To build XHTML output for the sample DITA map `samples\hierarchy.ditamap`, run the command:

```
ant -Dargs.input=samples\hierarchy.ditamap -Dtranstype=xhtml
```

DITA-OT displays a lot of output in the console window, including whether the build failed or succeeded at the end of the output.



When your build is unsuccessful, the error message may be difficult to find in the copious output. If you have not configured your console window most of the early output may have already scrolled off the screen. If you add an Ant property, `-l log-file` to the command line invocation, DITA-OT will save the output to a log file that you can study after the build finishes.

Writing Ant Build Files DITA-OT

Learning how buildfiles tell Ant what and how to compile.

The sample Ant build scripts provided by the DITA-OT may not be adequate to meet the needs of your documentation project. This topic describes how to customize the default scripts and write your own.

Customize the Default Ant Script

The DITA Open Toolkit contains sample build files for both the DITA-OT and sample documentation. Writers new to the toolkit use the `sample_all.xml` Ant build script to create all the sample documents that come with DITA-OT. The toolkit also contains build scripts for individual output types, such as `sample_pdf.xml`. You can modify just one or two Ant properties in these scripts for your own documentation.

Here is the Ant project definition from `template_pdf.xml`.

```
<project name="@PROJECT.NAME@_pdf" default="@DELIVERABLE.NAME@2pdf"
  basedir=".">
```

```

<property name="dita.dir" location="${basedir}${file.separator}..
${file.separator}.."/>

<target name="@DELIVERABLE.NAME@2pdf">
  <ant antfile="${dita.dir}${file.separator}build.xml">
    <property name="args.input" location="@DITA.INPUT@" />
    <property name="output.dir" location="@OUTPUT.DIR@" />
    <property name="transtype" value="pdf" />
  </ant>
</target>

</project>

```

You simply change the values of the following properties to match the values used in your project:

- Project name: The root element in an Ant build file.
- Target name: Must be one of the supported DITA-OT transtypes.

However, the toolkit's scripts assume that your input files are located in same directory structure used by the DITA-OT samples.

Write Your Own Ant Script

The default build script may not meet the needs of your project for a range of reasons:

- You want to add additional Ant properties not used in the sample template, such as XSL and DTD properties to assist your debugging efforts.
- Your content files may not have the same directory structure as the samples.
- You want to place the output files in a different directory.

You need to customize or write your own build file for these use cases. For example, each target for this guide's build script uses a separate value for `dita.temp.dir` to assist debugging for a specific output types.

Here is an example Ant script that can be used to produce this document:

```

<project name="antquickstartguide" default="dita2pdf" basedir=".">

  <property environment = "env"/>
  <property name="DITA_DIR" value="${env.DITA_DIR}"/>
  <property name="args.logdir" value="../logs"/>
  <property name="dita.input.valfile" value="../QSG.ditaval"/>

  <property name="dita.extname" value=".dita"/>
  <property name="args.fo.img.ext" value=".gif"/>

  <property name="output.dir" location="../output"/>
  <property name="outdir" location = "../output"/>
  <property name="clean.temp" value="no"/>

  <property name="args.indexshow" value="no"/>

  <target name="dita2pdf">
    <ant antfile="${DITA_DIR}/build.xml">
      <property name="transtype" value="pdf"/>
      <property name="args.input" value="../quickstartguide.ditamap"/>
      <property name="dita.temp.dir" value="${outdir}/temp_pdf"/>
      <property name="output.dir" value="${outdir}/pdf"/>
      <property name="outer.control" value="quiet"/>
      <property name="clean.temp" value="yes"/>
    </ant>
  </target>

</project>

```


[Ant argument properties for DITA-OT](#) on page 35 contains a list of the most basic Ant properties used by DITA-OT. Use these properties to customize your document's build script for your needs.

Generating Documents with command-line tool

How to generate documents from the command line with the DITA-OT command-line tool.

The DITA Open Toolkit provides a command-line tool to run document conversions. However, the command-line tool is a wrapper for the Ant interface, so you still must install Ant. In addition, only a subset of the Ant properties are supported by the command-line tool

1. Open a command prompt.
2. Change directories to where you installed the DITA Open Toolkit.
3. Set up the processing environment.

Enter the following command:

```
startcmd.bat
```

4. Run a conversion to a transformation output type.

Enter the following command:

```
java -jar lib/dost.jar [arguments]
```

Three arguments are required:

/i:source	defines the location of the .ditamap file for your document
/outdir:output-dir	defines the director where the output resides after DITA-OT finishes processing your project
/transtype:format	defines the type of document you want to generate for the project.

For example, the following command instructs DITA-OT to build the `samples/sequence.ditamap` as a PDF in the `out` directory:

```
java -jar lib/dost.jar /i:samples/sequence.ditamap /outdir:out /
transtype:pdf
```

Debugging DITA-OT Transformations

Transforming your DITA-compliant XML into documents.

Understanding the Role of the FO PlugIn. Debugging FO-generated tranformation files.

Introducing Document Generation

Learning the mechanics of document generation with DITA OT.

Your documentation project uses an Ant build script, which calls a target in another Ant build script in the DITA-OT root directory, which imports another Ant build script, which itself imports several more Ant build scripts. Sound confusing? This topic explains this interaction and explains how to identify targets in these scripts related to errors in your document generation.

Each target in the build script for this Quick Start Guide contains the following code snippet.

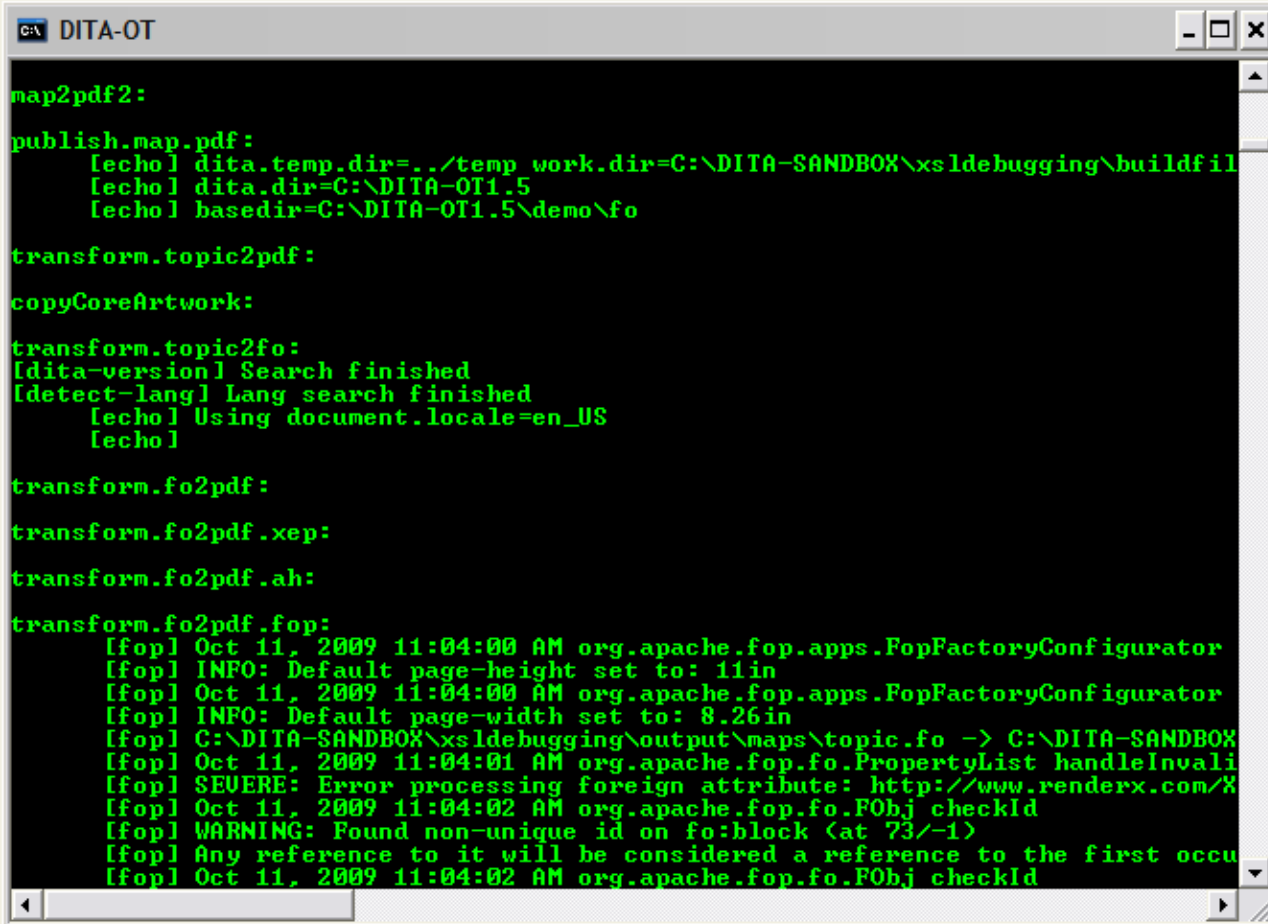
```
<ant antfile="${dita.dir}/build.xml" target="init">
```

The `toolkit_dir` directory is the root directory where you installed DITA-OT.

The build file you should understand is `build.xml`, located in folder bound to `dita.dir` property. The Ant targets defined and imported into this script are the same targets that you see on the console as your build script runs.

DITA-OT Build Script	Description
<code>build_init.xml</code>	Starts the document transformation, initializes the DITA-OT logger, verifies that the toolkit can locate the files and directories that you specified in your build file, and prints these values to the console and the log file, if you have specified one.
<code>build_preprocess.xml</code>	Validates your content files, generates lists of input files, including internal elements distributed across all content, such as index and conref entries. Moves copies of these files and elements into the the directory specified by <code>output.dir</code> property in your build script.
<code>build_general.xml</code> , <code>build_dita2wordrtf.xml</code> , <code>build_dita2xhtml.xml</code> , <code>build_dita2eclipsehelp.xml</code> , <code>build_dita2javahelp.xml</code> , <code>build_dita2htmlhelp.xml</code> , <code>build_dita2pdf</code>	Output specific Ant files which are not intended to be run directly.

Your console displays the name of each Ant target called inside the build scripts, including the output-specific script. For example, the following screen shot displays the names of Ant targets contained in the `build.xml` script when PDF transformation type is used.



```

C:\ DITA-OT

map2pdf2:
publish.map.pdf:
    [echo] dita.temp.dir=../temp work.dir=C:\DITA-SANDBOX\xsldebugging\buildfil
    [echo] dita.dir=C:\DITA-OT1.5
    [echo] basedir=C:\DITA-OT1.5\demo\fo

transform.topic2pdf:
copyCoreArtwork:
transform.topic2fo:
[ dita-version ] Search finished
[ detect-lang ] Lang search finished
    [echo] Using document.locale=en_US
    [echo]

transform.fo2pdf:
transform.fo2pdf.xep:
transform.fo2pdf.ah:
transform.fo2pdf.fop:
    [fop] Oct 11, 2009 11:04:00 AM org.apache.fop.apps.FopFactoryConfigurator
    [fop] INFO: Default page-height set to: 11in
    [fop] Oct 11, 2009 11:04:00 AM org.apache.fop.apps.FopFactoryConfigurator
    [fop] INFO: Default page-width set to: 8.26in
    [fop] C:\DITA-SANDBOX\xsldebugging\output\maps\topic.fo -> C:\DITA-SANDBOX
    [fop] Oct 11, 2009 11:04:01 AM org.apache.fop.fo.PropertyList handleInvali
    [fop] SEVERE: Error processing foreign attribute: http://www.renderx.com/X
    [fop] Oct 11, 2009 11:04:02 AM org.apache.fop.fo.FObj checkId
    [fop] WARNING: Found non-unique id on fo:block (at 73/-1)
    [fop] Any reference to it will be considered a reference to the first occu
    [fop] Oct 11, 2009 11:04:02 AM org.apache.fop.fo.FObj checkId
  
```

When you see an error in the output, you should read the Ant target that generated it for clues to solve the problem. To learn more about what caused the INFO, SEVERE, and WARNING errors in the image above, you should read the `transform.fo2pdf.fo` Ant target to learn what the Toolkit was doing when the error occurred and which xsl file generated the error.



Note: The DITA-OT build scripts sometimes continue to run even if they are unable to generate a temporary file for one of your content files. The build later displays an error message stating that a DITA-OT build script cannot find a generated file. This error is often misleading; the problem may be that your content file contains an error other than XML validation, which would stop the DITA-OT build from proceeding.

DITA-OT uses a separate set of Ant targets to process your PDF if you specify a value for the `args.fo.userconfig` property in your document's build script.

Best Practices

Tips and tricks for working directly with the DITA OT.

Create targets only for document types that you need.

DITA-OT's most attractive feature is its ability to produce so many different types of documents from the same source files. However, you may find that you need to tweak the targets in your Ant build file to get a document to meet your customization and style guide requirements. Although the sample documents for DITA-OT ship with every available target, there is no point in ironing out the details of a `dita2rtf` target in your build file if your documentation set doesn't require Word-based documents. If you're not providing JavaHelp, troff, or .rtf, then don't create targets for them.

Place all content inside or within the map directory if HTML Help is one of your output types.

The HTML Help Compiler cannot compile the files generated by DITA-OT for source files that reside outside the folder where your `.ditamap` file resides. If your documentation suite contains HTML Help, you should place all your source files in or below this directory.

For advanced debugging, use a different temp folder for each document type within the same build.

The Ant build script for the DITA-OT samples uses a unique folder for each build. However, many builds will include multiple targets, and some of these targets generate overlapping intermediate files. Specify a unique temp directory for each target within the same build to be sure that the intermediate files that you are reading were generated for the target you're debugging. See the build file for this document for an example.

Chapter

2

DITA Open Toolkit User Guide

Topics:

- [DITA Open Toolkit Release 1.5.4](#)
- [DITA 1.2 Specification Support](#)
- [Tested platforms and tools](#)
- [Distribution packages](#)
- [Installing the DITA Open Toolkit](#)
- [Using DITA transforms](#)
- [Running DITA-OT from Ant](#)
- [Running DITA-OT from command-line tool](#)
- [Configuration](#)
- [Available DITA-OT Transforms](#)
- [Problem determination and log analysis](#)
- [Troubleshooting](#)

The DITA Open Toolkit is an implementation of the OASIS DITA Technical Committee's specification for DITA DTDs and Schemas. The Toolkit transforms DITA content (maps and topics) into deliverable formats, such as XHTML, PDF, Eclipse Help, HTML Help, and JavaHelp.

This set of documentation contains some basic setup and overview information for the DITA Open Toolkit. The latest information about the toolkit, including plans for upcoming or future releases, can be found at dita.xml.org: [The DITA Open Toolkit](#).

DITA Open Toolkit Release 1.5.4

General Enhancements and Changes

Configuration file for defaults

In previous versions, `lib/configuration.properties` was generated by integration process. Integration has been changed to generate `lib/org.dita.dost.platform/plugin.properties` and the role of the old `lib/configuration.properties` has been changed to contain defaults and configuration options, such as default language.

Plug-in extension point for file extension configuration

New plug-in extension points have been added allow configuring DITA-OT behaviour based on file extensions.

Extension point	Description	Default values
<code>dita.topic.extension</code>	DITA topic	<code>.dita, .xml</code>
<code>dita.map.extensions</code>	DITA map	<code>.ditamap</code>
<code>dita.html.extensions</code>	HTML file	<code>.html, .htm</code>
<code>dita.resource.extensions</code>	Resource file	<code>.pdf, .swf</code>

Both HTML and resource file extensions are used to determine if a file in source is copied to output.

New plug-in extension point has been added to allow declaring transtypes as print types.

Extension point	Description
<code>dita.transtype.print</code>	Declare transtype as a print type.

Strict integration mode

Two modes have been added to integration process: lax and strict. In strict mode the processing will fail if any errors are encountered. In lax mode an error message may be thrown for an error and the integration process will try to run to the end, even if there are errors that were unrecoverable. The default mode is lax.



Note: In lax mode, even if the process runs to the end and reports a successful result, DITA-OT may not be able to function correctly because of e.g. corrupted plug-in files or unprocessed template files.

Code reference charset support

Encoding of the code reference target file can be set using the `format` attribute, for example

```
<coderef href="unicode.txt" format="txt; charset=UTF-8"/>
```

Plugin URI scheme

Support for plugin URI scheme has been added to XSLT stylesheets. Plug-ins can refer to files in other plug-ins without hard coding relative paths, for example

```
<xsl:import href="plugin:org.dita.pdf2:xsl/fo/topic2fo_1.0.xsl"/>
```

PDF

Support for mirrored page layout has been added. The default is the unmirrored layout.

The `args.bookmap-order` property has been added to control how front and back matter are processed in bookmarks. The default is to reorder the frontmatter content as in previous releases.

A new extension point has been added to add mappings to the PDF configuration catalog file.

Extension point	Description
<code>org.dita.pdf2.catalog.relative</code>	Configuration catalog includes.

Support for the following languages has been added:

- Finnish
- Hebrew
- Romanian
- Russian
- Swedish

PDF processing no longer copies images or generates XSL FO to output directory. Instead, the temporary directory is used for all temporary files and source images are read directly from source directory. The legacy processing model can be enabled by setting `org.dita.pdf2.use-out-temp` to `true` in configuration properties; support for the legacy processing model may be removed in future releases.

Support for FrameMaker index syntax has been disabled by default. To enable FrameMaker index syntax, set `org.dita.pdf2.index.frame-markup` to `true` in configuration properties.

A configuration option has been added to disable I18N font processing and use stylesheet defined fonts. To disable I18N font processing, set `org.dita.pdf2.i18n.enabled` to `false` in configuration properties

XHTML

Support for the following languages has been added:

- Indonesian
- Kazakh
- Malay

Migration from previous releases

The `print_transtypes` property in `integrator.properties` has been deprecated in favor of `dita.transtype.print` plug-in extension point.

The `dita.plugin.org.dita.*.dir` properties have been changed to point to DITA-OT base directory.

PDF

Support for mirrored page layout was added and the following XSLT configuration variables have been deprecated:

- `page-margin-left`
- `page-margin-right`

The following variables should be used instead to control page margins:

- `page-margin-outside`
- `page-margin-inside`

XSLT Parameters `customizationDir` and `fileProfilePrefix` have been removed in favor of `customizationDir.url` parameter.

A new shell stylesheet has been added for FOP and other shell stylesheets have also been revised. Plug-ins which have their own shell stylesheets for PDF processing should make sure all required stylesheets are imported.

Font family definitions in stylesheets have been changed from Sans, Serif, and Monospaced to sans-serif, serif, and monospace, respectively. The I18N font processing still uses the old logical names and aliases are used to map the new names to old ones.

SourceForge trackers

Feature requests

- 3333697 Add strict mode processing (Milestone 1)
- 3336630 Add resource file extension configuration (Milestone 1)
- 3323776 Base HTML stylesheets (Milestone 1)
- 3355860 Enable defining code ref target encoding (Milestone 1)
- 3393969 Make default TocJS output more usable (Milestone 3)
- 3394708 cfg/catalog.xml should be an extension point (Milestone 4)
- 3411030 Add args.fo.userconfig to PDF2 (Milestone 5)
- 3411961 Change margin-* to space-* property (Milestone 5)
- 3412144 Add FOP specific shell to PDF2 (Milestone 5)
- 3413215 Add schemas for PDF2 configuration files (Milestone 5)
- 3414416 Support bookmap order in PDF2 front and back matter (Milestone 5)
- 3413933 Fix inconsistencies in PDF2 page headers (Milestone 5)
- 3418877 Mechanism to refer to other plug-ins in XSLT (Milestone 5)
- 3411476 Add extension point for print type declaration (Milestone 6)
- 3392891 Copy the graphic files to the temporary folder (Milestone 6)
- 3429290 Remove unused Apache Commons Logging JAR (Milestone 6)
- 3434640 Add XHTML NLS support for Indonesian, Malay, Kazakh (Milestone 6)
- 3435528 Add base configuration file (Milestone 7)
- 3432219 Refactor dita.list read and write (Milestone 7)
- 3401849 PDF2: runtime switch for localization post-processing (Milestone 7)
- 3438361 Add "tocjs" transform to demo script (Milestone 7)
- 3341648 Clean HTML and XHTML stylesheets (Milestone 8)
- 3343562 Java clean-up (Milestone 8)
- 3346094 Improve test coverage (Milestone 8)
- 3372147 Improve logging (Milestone 8)
- 3373416 Refactor PDF attribute sets (Milestone 8)
- 3376114 Improve PDF page layout configuration (Milestone 8)
- 3415269 Support for more languages in the PDF transform (Milestone 8)
- 3412211 Refactor PDF index stylesheet for XSL 1.1 support (Milestone 8)
- 3425838 General PDF2 improvements (Milestone 8)
- 3428152 General I18N improvements (Milestone 8)
- 3429390 General XHTML improvements (Milestone 8)
- 3438790 Clean up build_demo script (Milestone 8)
- 3440826 Dutch patch for feature request 3415269 (Milestone 8)
- 1785391 Make Java code thread-safe (in progress)

Patches

- 2963037 PDF changes to fix index rendering of colon (bug 2879196) (Milestone 7)

Bugs

- 2714699 FO plug-in doesn't support specialized index elements (Milestone 1)
- 2848636 Duplicate key definitions should produce info messages (Milestone 1)
- 3353955 Frontmatter child order is not retained in PDF2 (Milestone 1)
- 3354301 XRef with conreffed phrases not properly generate HTML link (Milestone 1)
- 3281074 Bad attribute being applied to fo:bookmark-title element (Milestone 2)
- 3344142 Conref Push order of validation (Milestone 2)
- 3358377 Cryptic error message when DITA Map has "bookmap" extension (Milestone 3)
- 3384673 ODF transtype no longer embeds images in output (OT 1.5.3) (Milestone 3)
- 3394000 TocJS needs cleanup for several minor bugs (Milestone 3)
- 3392718 TOCJS sample should not require ant target (Milestone 3)
- 3389277 DocBook transform redundantly nests Related Links (Milestone 3)
- 3105339 '<' and '>' characters in a title cause tocjs trouble (Milestone 3)
- 3104497 tocjs JavaScripts don't work in Japanese environment (Milestone 3)
- 3394130 Remove outdated developer documentation (Milestone 3)
- 3397165 chunk on topichead not honored (Milestone 4)
- 3397501 Custom reltable column headers are reversed (Milestone 4)
- 3397495 Relcolspec with <title> does not generate link group headers (Milestone 4)
- 3399030 <ph> Elements not flagged with alt-text in HTML output (Milestone 4)
- 3396884 NPE in EclipseIndexWriter.java<Merges,setLogger for AbstractIndexWriters (Milestone 4)
- 3398004 -d64 flag to JVM not allowed for Windows JVMs (Milestone 4)
- 3401323 Fix PDF nested variable handling (Milestone 4)
- 3401721 Processing broken for <topicsetref> elements (Milestone 4)
- 3404049 Setting of clean_temp is backwards (Milestone 4)
- 3386590 Product name repeated hundreds of times in PDF (Milestone 4)
- 3405417 Shortdesc output twice when using abstract (Milestone 4)
- 3402165 wrong image output dir if using generate.copy.outer=2 (Milestone 4)
- 2837095 Positions of index and TOC in bookmaps are ignored (Milestone 5)
- 3414826 DITA OT not handling image path with chunking turned on (Milestone 5)
- 3411767 Not so meaningful messages given by ImgUtils (Milestone 5)
- 3405851 Incorrect entry@colname in merged XML with row and colspan (Milestone 5)
- 3406357 Custom profiling issue (Milestone 5)
- 3413203 Remove references to OpenTopic in PDF2 (Milestone 5)
- 3414270 @props specialization not used in map (Milestone 5)
- 3383618 Attribute 'link-back' cannot occur at element 'fo:index-key (Milestone 5)
- 3418953 Scale computation for XHTML uncorrectly looks up images (Milestone 6)
- 3413229 onlytopic.in.map & symlink (Milestone 6)
- 3423537 Additional line breaks in <menucascade> should be ignored (Milestone 6)
- 3423672 Problems with refs to images outside the DITA Map directory (Milestone 6)
- 2879663 indexterm/keyword causees NullPointerException (Milestone 7)
- 2879196 Colon character in <indexterm> causes nesting in output (Milestone 7)
- 3179018 Indexterm with only nested subelement results in NPE (Milestone 7)
- 3432267 Task example title processing incorrect for PDF (Milestone 7)
- 3430302 Unitless images sizes in throw errors (Milestone 7)
- 3429845 No variables for Warning (Milestone 7)
- 3428871 topicmerge gives incomplete topicref when reference or topic (Milestone 7)
- 3132976 Duplicate index text in index page (Milestone 7)
- 2795649 Java topicmerge ignores xml:lang (Milestone 7)
- 3431798 Relative CSS paths incorrectly computed for @copy-of (Milestone 7)
- 3438421 Remove transtype default (Milestone 7)

- 2866342 Nested see also is ignored (Milestone 8)
- 1844429 PDF2: Non-DITA link broken unless marked external (Milestone 8)
- 3270616 "lcTime" not displayed in PDF output (Milestone 8)
- 3388668 Data in figure captions not suppressed in xrefs (Milestone 8)
- 3429824 topicmerge gives wrong topicref with nested topics (Milestone 8)
- 3414332 PDF2 variable string translations missing (Milestone 8)
- 3323806 Improve Java logging and exception handling (Milestone 8)
- 3426920 Image files not copied or referenced correctly for eclipse (Milestone 8)
- 3445159 entry/@colname has been removed! (Milestone 8)
- 3447732 Bug in handling of longdesc (Milestone 8)
- 3452510 Ant parameter customization.dir not documented anywhere (Milestone 8)
- 3451621 Revisions on <plentry> use wrong image for nested <pd> (Milestone 8)

DITA 1.2 Specification Support

DITA Open Toolkit 1.5.4 supports the DITA 1.2 specification. Initial support for this specification was added in version 1.5 of the toolkit; versions 1.5.1 and 1.5.2 contain minor modifications to keep up with the latest drafts. The specification itself was approved at approximately the same time as DITA-OT 1.5.2, which contains the final versions of the DTD and Schemas.

Earlier versions of the DITA Open Toolkit contained a subset of the specification material, including descriptions of each DITA element. This material was shipped in source, CHM and PDF format. This was possible in part because versions 1.0 and 1.1 of the DITA Specification contained two separate specification documents: one for the architectural specification, and one for the language specification.

In DITA 1.2, each of these has been considerably expanded, and the two have been combined into a single document. The overall document is much larger, and including the same set of material would double the size of the DITA-OT package. Rather than include that material in the package, we've provided the links below to the latest specification material.

Highlights of DITA 1.2 support in the toolkit include:


- Processing support for all new elements and attributes
- Link redirection and text replacement using keyref
- New processing-role attribute in maps to allow references to topics that will not produce output artifacts
- New conref extensions, including the ability to reference a range of elements, to push content into another topic, and to use keys for resolving a conref attribute.
- The ability to filter content with controlled values and taxonomies, using the new Subject Scheme Map
- Processing support for both default versions of task (original, limited task, and the general task with fewer constraints on element order)
- Acronym and abbreviation support with the new <abbreviated-form> element
- New link grouping abilities available with headers in relationship tables
- OASIS Subcommittee specializations from the learning and machine industry domains (note that the core toolkit contains only basic processing support for these, but can be extended to produce related artifacts such as SCORM modules)

To find detailed information about any of these features, see the specification documents at OASIS. The DITA Adoption Technical Committee has also produced several papers to describe individual new features. In general, the white papers are geared more towards DITA users and authors, while the specification is geared more towards tool implementors, though both may be useful for either audience. The DITA Adoption papers can be found from that TC's main web page.

Tested platforms and tools

See which tools and platforms have been used in testing the DITA processing system.

The DITA processing system has been tested against the following platforms and tools:

Platform or tool	Tested version
OS	<ul style="list-style-type: none"> Windows XP Windows 7 OS X 10.6 SLES 10
XSLT processor	<ul style="list-style-type: none"> Xalan-J 2.6 Xalan-J 2.7 Xalan-J 2.7.1 Saxon 6.5 Saxon 9 Saxon-B 9.1 Saxon-HE/EE 9.3 <p> Note: XSLT 2.0 standard is not officially required in DITA-OT code, due to the reliance by some users on Xalan.</p>
JDK	<ul style="list-style-type: none"> IBM 1.5 IBM 1.6 Oracle 1.5 Oracle 1.6
Ant	<ul style="list-style-type: none"> Ant 1.7.1 Ant 1.8.2

Distribution packages

DITA-OT is available in three three binary packages, as well as in a source package.

Minimal package (DITA-OT1.5.4_minimal_bin.zip).

This package is primarily for use by vendors that embed the toolkit within their products. It contains all of the core processing code: CSS and XSLT, Ant build scripts, Java code (dost.jar), resource files, DTDs and Schemas. Users will need to have their own version of Ant and other libraries, and will need to set up environment variables for each library. The only external files inside the minimal package are the OASIS DTDs and Schemas, along with the following open source libraries:

- Apache Commons Codec 1.4
- Apache Catalog Resolver 1.1

Standard package (DITA-OT1.5.4_bin.zip)

This package contains everything in the minimal package, plus documentation, existing demo code (such as legacy support for the old bookmap), sample Ant scripts, and sample DITA files. This package is appropriate for those who want the core toolkit function, demos, and samples, but already have local installed copies of Ant and other required tools. The standard package includes only the following open source libraries:

- Apache Commons Codec 1.4
- Apache Catalog Resolver 1.1

Full Easy Install package (DITA-OT1.5.4_full_easy_install_bin.zip)

This package contains everything in the Standard package, plus common Apache tools used for builds so that you do not need to set them up separately; the only core tool that is missing is Java. This package also contains a batch file to set up a build environment using those tools. This package is appropriate for novice users, those testing the toolkit, or those who do not want to maintain local copies of other Apache tools. The following external libraries are included in version 1.5.4 of this package:

- Apache Ant 1.7.1
- Saxon 9.1
- Apache Commons Codec 1.4
- Apache Catalog Resolver 1.1
- IBM ICU for Java 3.4.4
- Apache FOP 1.0

Source package (DITA-OT1.5.4_src.zip)

In addition to the binary versions of the toolkit, there is a source package available that contains all of the source code used in each of the three packages. The demo files, samples, and documentation are not part of the source package. The source package contains the Apache Commons Logging and Apache Catalog Resolver libraries.

Installing the DITA Open Toolkit

This topic explains how to install the DITA Open Toolkit processing environment.

Installing DITA-OT Full Easy Install package in Windows

The software that DITA-OT depend on are redistributed. The Full Easy Install package of DITA-OT after 1.3 includes the software, so the configuration process is streamlined.

Before installing DITA Open Toolkit full distribution, you need to complete the following steps:

- Download and configure JRE properly.
- (Optional) Download and configure HTMLHelp Compiler properly for HTMLHelp transformation.
- (Optional) Download and configure JavaHelp Compiler properly for JavaHelp transformation.

Then, you need to complete the following steps:

1. Download the DITA-OT Full Easy Install package.
2. Unzip DITA-OT1.5.4_full_easy_install_bin.zip into the installation directory.
3. Run the batch file "startcmd.bat" to set up the necessary environment variables.
A new **Command Prompt** window will open up, with the environment variables already set to enable DITA-OT to run within that shell.
4. Run the transformation in the Command Prompt window by using Ant or command-line tool.

Installing DITA-OT Full Easy Install package in Linux or OS X

The software that DITA-OT depend on are redistributed. The Full Easy Install package of DITA OT after 1.3 includes the software, so the configuration process is streamlined.

Before installing DITA Open Toolkit full distribution, you need to complete the following steps:

- Download and configure JRE properly.
- (Optional) Download and configure JavaHelp Compiler properly for JavaHelp transformation.

Then, you need to complete the following steps:

1. Download the DITA-OT Full Easy Install package.
2. Extract `DITA-OT1.5.4_full_easy_install_bin.tar.gz` into the installation directory.
3. Run shell script "`startcmd.sh`" to set up the necessary environment variables.
A new **Terminal** window will open up, with the environment variables already set to enable DITA-OT to run within that shell.
4. Run the transformation in the Terminal window by using Ant or command-line tool.

Set up DITA Open Toolkit in Windows

The recommended use of the DITA Open Toolkit components is inside of the Java environment because its *pre-process architecture* needs Java and Java-based tools. Therefore, before installing the DITA Open Toolkit processing environment, ensure that you have installed the following prerequisite tools :



Note: See *Tested platforms and tools* for detailed information about versions of these tools that have been successfully tested with the current toolkit release.

Java runtime or development environment 1.5

Provides the basic environment for most tools used in this toolkit.

You can download and install the Java Runtime Environment (JRE) 1.5 (or greater) (available on <http://www.oracle.com/technetwork/java/javase/overview/index.html>) into a directory of your choice.

XSLT 1.0 compliant transformation engine

Provides the main transformation services via the advanced XSLT processor Saxon 9.1 (or greater) or Xalan-J 2.7.1 (or greater) in the toolkit.

You can download and extract Saxon (available at <http://saxon.sourceforge.net/>) or the Xalan-J (available at <http://xml.apache.org/xalan-j/downloads.html>) into a directory of your choice.

Ant 1.7.1 build tool

Provides the standard setup and sequencing of processing steps.

The following steps guide you to set up the DITA Open Toolkit processing environment.

1. Download the DITA Open Toolkit package file from [SourceForge](#) .



Note:

- It is recommended to download the latest version of the DITA Open Toolkit for stable usage.
- If you use DITA Open Toolkit full distribution, follow instructions in *Installing DITA-OT Full Easy Install package in Windows* on page 28.

2. Unzip the package file into a installation directory of your choice.

For example `C:\pkg\DITA-OT1.5.4`

3. Verify that the environment variable `JAVA_HOME` has been set.

```
set JAVA_HOME=<JRE_dir>
```

4. Verify that the environment variable `ANT_HOME` has been set.

```
set ANT_HOME=<Ant_dir>
```

5. Verify that the environment variable `PATH` includes Java and Ant executables.

```
set PATH=%JAVA_HOME%\bin;%ANT_HOME%\bin;%PATH%
```

6. Set up DITA_HOME environment variable to point to DITA-OT installation directory.

```
set DITA_HOME=<DITA-OT_dir>
```

7. Set up your environment variable CLASSPATH.

```
set CLASSPATH=%DITA_HOME%\lib\dost.jar;%CLASSPATH%
set CLASSPATH=%DITA_HOME%\lib;%CLASSPATH%
set CLASSPATH=%DITA_HOME%\lib\resolver.jar;%CLASSPATH%
set CLASSPATH=%DITA_HOME%\lib\commons-codec-1.4.jar;%CLASSPATH%
```

8. Set up the XSLT processor.

- If you use the Saxon, set up CLASSPATH to include Saxon JAR files.

```
set CLASSPATH=<saxon_dir>\saxon9.jar;<saxon_dir>\saxon9-dom.jar;
%CLASSPATH%
```

Set up ANT_OPTS.

```
set ANT_OPTS=%ANT_OPTS% -
Djavadoc.xml.transform.TransformerFactory=net.sf.saxon.TransformerFactoryImpl
```

- If you use the Xalan, set up CLASSPATH to include Xalan JAR files.

```
set CLASSPATH=<xalan_dir>\xalan.jar;%CLASSPATH%
```

9. Optional: If you need JavaHelp output, set up your environment variable JHHOME.

```
set JHHOME=<javahelp_dir>
```

10. Optional: If you need Compiled HTML Help output, add Microsoft HTML Help Workshop installation directory to local.properties as hhc.dir property.

```
hhc.dir=C:\\Program Files (x86)\\HTML Help Workshop
```

11. Optional: If you use FOP for PDF processing, add FOP installation directory to local.properties as fop.home property.

```
fop.home=C:\\Program Files\\fop
```

12. Optional: If you use RenderX for PDF processing, add RenderX installation directory to local.properties as xep.dir property.

```
xep.dir=C:\\Program Files\\xep
```

13. Optional: If you use AntennaHouse Formatter for PDF processing, add AH Formatter installation directory to local.properties as axf.path property.

```
axf.path=C:\\Program Files\\AHFormatterV6
```

14. Test the DITA-OT installation with the demo conversions.

Run all demos in the DITA Open Toolkit directory.

```
C:\pkg\DITA-OT1.5.4>ant -f samples\ant_sample\sample_all.xml
```

Set up DITA Open Toolkit in Linux or OS X

The following steps guide you to set up the DITA Open Toolkit processing environment in Linux or OS X.

The recommended use of the DITA Open Toolkit components is inside of the Java environment because its *pre-process architecture* needs Java and Java-based tools. Therefore, before installing the DITA Open Toolkit processing environment, ensure that you have installed the following prerequisite tools :



Note: See *Tested platforms and tools* for detailed information about versions of these tools that have been successfully tested with the current toolkit release.

Java runtime or development environment 1.5

Provides the basic environment for most tools used in this toolkit.

You can download and install the Java Runtime Environment (JRE) 1.5 (or greater) (available on <http://www.oracle.com/technetwork/java/javase/overview/index.html>) into a directory of your choice.

XSLT 1.0 compliant transformation engine

Provides the main transformation services via the advanced XSLT processor Saxon 9.1 (or greater) or Xalan-J 2.7.1 (or greater) in the toolkit.

You can download and extract Saxon (available at <http://saxon.sourceforge.net/>) or the Xalan-J (available at <http://xml.apache.org/xalan-j/downloads.html>) into a directory of your choice.

Ant 1.7.1 build tool

Provides the standard setup and sequencing of processing steps.

1. Download the DITA Open Toolkit package file from [SourceForge](#) .



Note:

- It is recommended to download the latest version of the DITA Open Toolkit for stable usage.
- If you use DITA Open Toolkit full distribution, follow instructions in *Installing DITA-OT Full Easy Install package in Linux or OS X* on page 28.

2. Extract the package file into a installation directory of your choice.



Note: You can extract all package files and toolkits either to your private home directory for exclusive usage or to `/usr/local/share/` directory for sharing.

3. Verify that the environment variable `JAVA_HOME` has been set.

```
export JAVA_HOME=<JRE_dir>
```

4. Verify that the environment variable `ANT_HOME` has been set.

```
export ANT_HOME=<Ant_dir>
```

5. Verify that the environment variable `PATH` includes Java and Ant executables.

```
export PATH=$JAVA_HOME/bin:$ANT_HOME/bin:$PATH
```

6. Set up `DITA_HOME` environment variable to point to DITA-OT installation directory.

```
export DITA_HOME=<DITA-OT_dir>
```

7. Set up your environment variable `CLASSPATH`.

```
export CLASSPATH=$DITA_HOME/lib/dost.jar:$CLASSPATH
export CLASSPATH=$DITA_HOME/lib:$CLASSPATH
export CLASSPATH=$DITA_HOME/lib/resolver.jar:$CLASSPATH
export CLASSPATH=$DITA_HOME/lib/commons-codec-1.4.jar:$CLASSPATH
```

8. Set up the XSLT processor.

- If you use the Saxon, set up CLASSPATH to include Saxon JAR files.

```
export CLASSPATH=<saxon_dir>/saxon9.jar:<saxon_dir>/saxon9-dom.jar:
$CLASSPATH
```

Set up ANT_OPTS.

```
export ANT_OPTS=$ANT_OPTS -
Djavax.xml.transform.TransformerFactory=net.sf.saxon.TransformerFactoryImpl
```

- If you use the Xalan, set up CLASSPATH to include Xalan JAR files.

```
export CLASSPATH=<xalan_dir>/xalan.jar:$CLASSPATH
```

9. Optional: If you need JavaHelp output, set up you environment variable JHHOME.

```
export JHHOME=<javahelp_dir>
```

10. Optional: If you use FOP for PDF processing, add FOP installation directory to local.properties as fop.home property.

```
fop.home=/usr/share/java/fop
```

11. Optional: If you use RenderX for PDF processing, add RenderX installation directory to local.properties as xep.dir property.

```
xep.dir=/usr/share/java/xep
```

12. Optional: If you use AntennaHouse Formatter for PDF processing, add AH Formatter installation directory to local.properties as axf.path property.

```
axf.path=/usr/share/java/AHFormatterV6
```

13. Test the DITA-OT installation with the demo conversions.

Run all demos in the DITA Open Toolkit directory.

```
/usr/local/share/DITA-OT1.5.4$ ant -f samples/ant_sample/sample_all.xml
```

Using DITA transforms

The core transforms of the DITA Open Toolkit represent an implementation of all processing defined by OASIS in the DITA specification.

Pre-process

A pre-process is done before the main transformation. The input of the pre-process is dita files (maps and topics) and the output of the pre-process is often referred to as "normalized" dita files. The pre-process stage resolves several common DITA behaviors, such as resolving conref attributes, resolving keyref values, and adding links based on the hierarchy and relationship tables in a map. The normalized dita files are in a temporary directory. Most DITA transforms use this common pre-process setup.

Available core transforms

A core DITA transform is the basic set of templates that support all the elements of a topic. This set is the basis for the following processing of any specialized element. Core transforms handle one topic instance, or nested set of topics, at a time. The DITA Open Toolkit provides several core transforms:

PDF	PDF output is based on a plug-in that produces XSL-FO output. The XSL-FO output may be converted to PDF using an open source tools (Apache FOP) or using commercial tools such as RenderX or Antenna House Formatter. This transformation type replaced an older demo PDF transform, and is often called "PDF2".
XHTML	XHTML output is supported together with a variety of XHTML-based navigation types. Navigation is not required. The XHTML output contains class values based on the DITA elements so CSS files may be used to customize the output style.
Eclipse Help	Eclipse output is an XHTML based format that also produces navigation and index files for use with Eclipse information centers.
TocJS	The TocJS transformation type includes XHTML output along with JavaScript based frames for navigation, including TOC sections that expand and collapse.
HTML Help	Microsoft Compiled HTML Help output produces a compiled help (.chm) file with the XHTML topics, table of contents, and index.
Java Help	Java Help includes a table of contents and index for use with the Java Help platform.
OpenDocument (ODT)	ODT is a document format based on the ODF standard, for use with tools like Open Office. Support for ODT was added in DITA-OT 1.5.2.
Rich Text Format (RTF)	RTF output is supported for basic content, but complex markup and some advanced features of DITA may not be supported.
troff	troff based man pages are supported; one man page is generated for each input topic. Note that tables are not supported by this transformation type.

Invoke the complete transformation

The complete transformation including pre-process can be executed by an Ant script. There are some examples of simple Ant scripts in the directory `samples/ant_sample`.

Running DITA-OT from Ant

Ant is an open tool that the DITA Open Toolkit uses to manage builds.

Introduction to Ant

Ant is a Java-based, open source tool provided by the Apache Foundation to declare a sequence of build actions. It is well suited for development builds as well as document builds. The "Full Easy Install" version of the toolkit ships with a copy of Ant.

DITA provides a set of XSLT scripts for producing different types of documentation, such as help output in Eclipse, Java Help and HTML Help, XHTML pages, and PDF. The DITA-OT uses Ant to manage these scripts, as well as to manage additional intermediate steps written in Java.

Running Ant

After setting up the Ant environment, you can build the DITA output by running the `ant` command.

Here are some samples to explain how to use Ant to build sample output in the DITA directory.



Note: To run the Ant demo properly, you should switch to the **DITA-OT installation directory** under the command prompt. If you are using the "Full Easy Install" package, running the `startcmd.bat` batch file in that directory will give you a prompt that is already set up for the following commands.

To build XHTML output for the sample DITA map `samples/hierarchy.ditamap`, run the command:

```
ant -Dargs.input=samples/hierarchy.ditamap -Doutput.dir=out/
samples/web -Dtranstype=xhtml
```

This will generate the output into `out/samples/web` directory. The `samples` directory also contains dedicated Ant build files for various transformation types and the same output can be generated with the `sample_xhtml.xml`:

```
ant -f samples/ant_sample/sample_xhtml.xml
```

You can build all samples in the DITA directory:

```
ant -f build_demo.xml all
```

The building process will create an `/out/` directory and put the output files in subdirectories that parallel the source directory.



Note: To find out the complete list of targets you can clean and build, check the name attributes for the target elements within the `build_demo.xml` file. Or, input `ant -f build_demo.xml -projecthelp` for a full list information.

You can also build your own selections using a prompted build.

```
ant -f build_demo.xml
```

Ant will prompt you for the input file, output directory, and transform type. Values on these parameters are case sensitive.



Note: To troubleshoot problems in setting up Java, Ant, Saxon, or FOP, you will get better information from the communities for those components rather than the communities for the DITA. Of course, if you find issues relevant to the DITA XSLT scripts (or have ideas for improving them), you are encouraged to engage the DITA community.

Ant tasks and scripts

This topic describes detailed Ant tasks and scripts.

The build process including pre-process can be called by using an Ant script. The most important Ant scrip file is called `build.xml`, it defines and combines common pre-processing and output transformation routines, as well as extension points that allow DITA-OT Plug-ins to add to this common processing.

Sample ant script

These ant scripts are in `samples/ant_sample` directory. They are simple and easy to learn. From these files, you can learn how to write your own Ant script to build your own process.

Here is a sample template for writing an Ant script that executes transformation to XHTML in `samples/ant_samples` directory:

```
<?xml version="1.0" encoding="UTF-8" ?>
<project name="@PROJECT.NAME@_xhtml" default="@DELIVERABLE.NAME@2xhtml"
  basedir=".">

  <property name="dita.dir" location="${basedir}/../.."/>

  <target name="@DELIVERABLE.NAME@2xhtml">
    <ant antfile="${dita.dir}${file.separator}build.xml">
      <!-- please refer to the toolkit's document for supported parameters,
and
      specify them base on your needs -->
      <property name="args.input" location="@DITA.INPUT@" />
      <property name="output.dir" location="@OUTPUT.DIR@" />
      <property name="transtype" value="xhtml" />
    </ant>
  </target>

</project>
```

To use this template, modify the following items:

- Replace `@PROJECT.NAME@` with the name of your project, such as "MyDocs".
- Replace `@DELIVERABLE.NAME@` with the name of your deliverable, such as "installDocs".
- Replace `@DITA.INPUT@` with the name of your input file (using either a full path or a relative path from the location of this template).
- Replace `@OUTPUT.DIR@` with the desired output directory (using either a full path or a relative path from the location of this template).

Once you have updated these items, you can run your build with the following command:

```
ant -f samples/ant_sample/template_xhtml.xml
```

The build will convert your input file to XHTML. Note that the build directly calls the Ant script `build.xml`, which is a common entry point for DITA-OT builds; it in turn imports all of the scripts mentioned above.

Ant argument properties for DITA-OT

Reference list of DITA-specific Ant properties.

DITA-OT processes your documentation project as an Ant project, which allows several Ant build properties specific to DITA-OT and your project. These properties can be divided into three categories:

- Properties specific to your documentation project
- Properties specific to the DITA Open Toolkit that you may override
- Properties specific to the DITA Open Toolkit that you should never override





The following tables describes the first group of properties, grouped by transformation type.

Parameters available to all transforms

The following common parameters are available for use by all DITA-OT builds.

Table 1: Common DITA-OT parameters

Project Ant Property	Description
<code>args.debug</code>	Specifies that DITA-OT print debugging information for your project. Allowed values are "yes" and "no". Default value is "no".

Project Ant Property	Description
<code>args.draft</code>	<p>Indicates whether draft-comment and required-cleanup elements are included in the generated file. Corresponds to XSLT parameter DRAFT in most XSLT modules. Allowed values are "yes" and "no". Default value is "no".</p> <p> Tip: For PDF output, setting <code>\${args.draft}</code> to "yes" will also cause the contents of <code><titlealts></code> to appear below the title.</p>
<code>args.figurelink.style</code>	<p>Specifies how cross references to figures are styled. Allowed values are "NUMBER" and "TITLE". NUMBER results in "Figure 5", TITLE results in the title of the figure. Corresponds to the XSLT parameter FIGURELINK.</p> <p> Note: This parameter is available for all except the PDF transform.</p>
<code>args.grammar.cache</code>	<p>Specifies whether to use the grammar caching feature of the XML parser. Allowed values are "yes" and "no". Default value is "yes".</p> <p> Note: For most users, this is an important option that dramatically speeds up processing time. However, there is a known problem with using this feature for documents that use XML Entities. If your build fails with parser errors about entity resolution, try setting this parameter to "no".</p>
<code>args.input</code>	<p>Typically defines the location of the .ditamap file for your documentation project. However, the property can be set to a .dita file, as well. DITA-OT reads this file to find the .dita files that comprise the content for the documentation project.</p>
<code>args.logdir</code>	<p>Defines the location where DITA-OT places log files for your project.</p>
<code>args.outext</code>	<p>Specifies the file extension for HTML files in your project's output. Corresponds to XHTML parameter OUTEXT. Default values is ".html".</p>
<code>args.tablelink.style</code>	<p>Specifies how cross references to tables are styled. Allowed values are "NUMBER" or "TITLE". Default is "NUMBER", which produces results such as "Table 5". TITLE results in the title of the table. Corresponds to the XSLT parameter TABLELINK.</p> <p> Note: This parameter is available for all except the PDF transform.</p>
<code>basedir</code>	<p>The directory where your project's ant build script resides. The DITA-OT will look for your .dita files relative to this directory. DITA-OT's default build script sets this as an attribute of the project, but you can also define it as a project property.</p>
<code>dita.ext</code>	<p>Specifies an extension to use for DITA topics; All DITA topics will use this single extension in the temp directory. Corresponds to XSLT parameter DITAEXT. Default value is ".xml"</p>

Project Ant Property	Description
<code>dita.input.valfile</code>	Defines the location of your project's filter file. Filter files end with the <code>.ditaval</code> suffix and are used to filter, include and exclude, content in the generated document. Alternatively, you can create multiple versions of your document by creating a different <code>.ditamap</code> file for each version.
<code>output.dir</code>	The location of the directory to hold output from your documentation project.
<code>transtype</code>	<p>Defines the output type for a specific Ant target. Plug-ins may add new values for this option; by default, the following values are available:</p> <ul style="list-style-type: none"> • pdf • xhtml • htmlhelp • eclipsehelp • eclipsecontent • odt • troff • rtf • javahelp • legacypdf • docbook
<code>validate</code>	Specifies whether DITA-OT should validate your content files. Allowed values are "yes" and "no". Default value is "yes".



Parameters available for all XHTML based transforms


The following parameters are available for all output types that are based on the XHTML transform type, including:

- XHTML
- HTMLHelp
- JavaHelp
- eclipsehelp

Table 2: XHTML and related parameters

Project Ant Property	Description
<code>args.artlbl</code>	Adds a label to each image containing the image's filename. Allowed values are "yes" and "no". Default value is "no".
<code>args.breadcrumbs</code>	Specifies whether to generate breadcrumb links. Corresponds to the XSLT parameter <code>BREADCRUMBS</code> . Allowed values are "yes" and "no". Default value is "no".
<code>args.copycss</code>	Indicates whether you want to copy your own <code>.css</code> file to the output directory.
<code>args.css</code>	The name of your custom <code>.css</code> file.
<code>args.csspath</code>	The location of your copied <code>.css</code> file relative to the output directory. Corresponds to XSLT parameter <code>CSSPATH</code> .
<code>args.cssroot</code>	The directory that contains your custom <code>.css</code> file. DITA-OT will copy the file from this location.


Project Ant Property	Description
<code>args.ftr</code>	<p>Specifies the location of a well-formed XML file containing your custom running-footer for the document body. Corresponds to XSLT parameter FTR.</p> <p> Note: The fragment must be valid XML, with a single root element, common practice is to place all content into <div>.</p>
<code>args.gen.default.meta</code>	<p>Specifies whether to generate extra metadata that targets parental control scanners, meta elements with name="security" and name="Robots". Allowed values are "yes" and "no". Default value is "no". Corresponds to the XSLT parameter genDefMeta.</p>
<code>args.gen.task.lbl</code>	<p>Specifies whether to generate locale-based default headings for sections within task topics. Allowed values are "YES" and "NO". Default value is "NO". Corresponds to the XSLT parameter GENERATE-TASK-LABELS.</p>
<code>args.hdf</code>	<p>Specifies the location of a well-formed XML file to be placed in the document head.</p>
<code>args.hdr</code>	<p>Specifies the location of a well-formed XML file containing your custom running-header for the document body. Corresponds to XSLT parameter HDR.</p> <p> Note: The fragment must be valid XML, with a single root element, common practice is to place all content into <div>.</p>
<code>args.hide.parent.link</code>	<p>Specifies whether to hide links to parent topics in the rendered XHTML. Corresponds to the XSLT parameter NOPARENTLINK. Allowed values are "yes" and "no". Default value is "no".</p>
<code>args.indexshow</code>	<p>Indicates whether <code>indexterm</code> element should appear in the output. Allowed values are "yes" and "no". Default value is "no".</p>
<code>args.xhtml.toc.class</code>	<p>String for a CSS class name attribute applied to the TOC (x)HTML output's <body> element. Found in <code>map2htmltoc.xsl</code>.</p>
<code>args.xhtml.classattr</code>	<p>Specifies whether to include DITA class ancestry inside generated XHTML elements. Allowed values are "no" and "yes"; the default is "yes" in release 1.5.2 (it was "no" in 1.5 and 1.5.1). For example, the <code>prereq</code> element in a task (which is specialized from section) would generate <code>class="section prereq"</code>. Corresponds to the XSLT parameter PRESERVE-DITA-CLASS.</p>
<code>args.xsl</code>	<p>Specifies an XSL file that is used rather than the default XSL transform, located in <code>toolkitdir\xsl\dita2xhtml.xsl</code>. Property must specify the full path and XSL file name.</p>
<code>generate.copy.outer</code>	<p>Specifies whether to generate files for content files that are not located in or beneath the directory containing your <code>ditmap</code> file. Supported values are:</p> <ul style="list-style-type: none"> • "1" (default) – do not generate outer files • "2" – generate outer files

Project Ant Property	Description
<code>onlytopic.in.map</code>	<ul style="list-style-type: none"> "3" – old solution. <p>Specifies whether files that are linked to, or referenced with a <code>conref</code> attribute, should generate output. If set to "yes", only files that are referenced directly from the map will generate output files.</p>
<code>outer.control</code>	<p>Specifies whether content files are located in or below the directory containing your <code>.ditamap</code> file. Supported values are:</p> <ul style="list-style-type: none"> "fail" – fail quickly if files are going to be generated/copied outside of that directory "warn" (default) – complete if files will be generated/copied outside, but log a warning "quiet" – quietly finish with only those files (no warning or error). <p>The <code>gen-list-without-flagging</code> Ant task generates a harmless warning for content outside the map directory; you can suppress these warnings by setting the <code>outer.control</code> property to "quiet".</p> <p> Warning: Microsoft HTML Help Compiler cannot produce HTMLHelp for documentation projects that use outer content. Your content files must reside in or below the directory containing the <code>.ditamap</code> file, and the map file cannot specify <code>..</code> at the start of <code>href</code> attributes for <code>topicref</code> elements.</p>

PDF-specific Ant Properties

The following table describes Ant properties that are specific to the PDF transformation type.

Table 3: PDF parameters

Project Ant Property	Description
<code>args.fo.include.rellinks</code>	<p>Specifies which links to include in the PDF file. Values are:</p> <ul style="list-style-type: none"> "none" (default) – no links are included. "all" – all links are included. "nofamily" – hard coded links and reltable-based links are included. Parent, child, next, and previous links are not included.
<code>args.fo.output.rel.links</code>	<p>Specifies whether to show links in your project's output. Values are "yes" (include all links) and "no" (the default, include no links). If <code>\${args.fo.include.rellinks}</code> is specified, this parameter is ignored.</p> <p> Note: This parameter is deprecated in favor of <code>\${args.fo.include.rellinks}</code>.</p>
<code>args.gen.task.lbl</code>	<p>Specifies whether to generate locale-based default headings for sections within task topics. Allowed values are "YES" and "NO". Default value is "NO". Corresponds to the XSLT parameter <code>GENERATE-TASK-LABELS</code>.</p>

Project Ant Property	Description
<code>args.xml.pdf</code>	Specifies an XSL file that is used in place of the default XSL transform at <code>demo\fo\xsl\fo\topic2fo_shell.xml</code> . You must specify the full path and XSL file name.
<code>publish.required.cleanup</code>	Indicates whether draft-comment and required-cleanup elements are included in the generated file. Allowed values are "yes" and "no". Default value is value of <code>args.draft</code> property. Deprecated, <code>args.draft</code> parameter should be used instead. Corresponds to XSLT parameter <code>publishRequiredCleanup</code> .
<code>args.fo.userconfig</code>	The parameter to specify the user configuration file for FOP.
<code>custom.xep.config</code>	The parameter to specify the user configuration file for RenderX.
<code>retain.topic.fo</code>	Specifies whether to leave the generated FO file for a PDF project.
<code>args.bookmap-order</code>	Specify if frontmatter and backmatter content order is retained in bookmap. Values are "retain" and "discard" (default).
<code>customization.dir</code>	Specifies the customization directory path.
<code>pdf.formatter</code>	<p>Specified the XSL processor to use. Supported values are:</p> <ul style="list-style-type: none"> • "fop" (default) – Apache FOP • "ah" – Antenna House Formatter • "xep" – RenderX XEP Engine <p>The Full Easy Install distribution package comes with Apache FOP installed, other XSL processors need to be separately installed.</p>

ODT-specific Ant Properties

The ODT transform, which produces a document using the Open Document Format, is available in the 1.5.2 version of the DITA-OT.

Table 4: ODT related parameters

Project Ant Property	Description
<code>args.odt.img.embed</code>	Determines whether images are embedded as binary objects within the ODT file.
<code>args.odt.include.rellinks</code>	<p>Specifies which links to include in the ODT file. Values are:</p> <ul style="list-style-type: none"> • "none" (default) – no links are included. • "all" – all links are included. • "nofamily" – hard coded links and reltable-based links are included. Parent, child, next, and previous links are not included.

EclipseContent-specific Ant Properties

The "eclipsecontent" transform type produces normalized DITA files, along with Eclipse TOC and project files.

Table 5: EclipseContent properties

Project Ant Property	Description
<code>args.eclipsecontent.toc</code>	Specifies the name of the TOC file for an Eclipse Content project.

XHTML-specific Ant Properties



Parameters in this section are used by the "xhtml" transtype, but not by other XHTML based transforms.


Table 6: Properties for the "xhtml" transform type

Project Ant Property	Description
<code>args.xhtml.contenttarget</code>	Specifies the content frame name where links from TOC are opened.
<code>args.xhtml.toc</code>	Specifies the name of the entry point for an XHTML project. Default value is <code>index.html</code>

EclipseHelp-specific Ant Properties

The following table describes Ant properties that are specific to the EclipseHelp transformation type, which is an XHTML based output for use with the Eclipse Help System.

Project Ant Property	Description
<code>args.eclipsehelp.toc</code>	Specifies the name of the TOC file.
<code>args.eclipse.country</code>	Specifies the more specific region for the language specified with <code>args.eclipse.language</code> . For example, US, CA and GB would clarify a value of "en" for <code>args.eclipse.language</code> . The content will be moved into the appropriate directory structure for an Eclipse fragment.
<code>args.eclipse.language</code>	Specifies the base language for translated content, such as "en" for English. This parameter is a prerequisite for <code>args.eclipse.country</code> . The content will be moved into the appropriate directory structure for an Eclipse fragment.
<code>args.eclipse.provider</code>	Specifies the name of the person or organization providing an Eclipse Help project. Default value is DITA.  Tip: The toolkit ignores the value of this property when processing an Eclipse Collection Map, <code>eclipse.dtd</code> .
<code>args.eclipse.version</code>	Specifies the version number to include in the output. Default value is 0.0.0.  Tip: The toolkit ignores the value of this property when processing an Eclipse Collection Map, <code>eclipse.dtd</code> .
<code>args.eclipse.symbolic.name</code>	Specifies the symbolic name (aka plugin ID) in the output for an Eclipse Help project. The @id value from the DITA map or the Eclipse map collection (Eclipse help specialization) is the symbolic name for the plugin in Eclipse. By default, the value <code>org.sample.help.doc</code> .

Project Ant Property	Description
	 Tip: The toolkit ignores the value of this property when processing an Eclipse Collection Map, <code>eclipse.dtd</code> .

HtmlHelp-specific Ant Properties

The following table describes Ant properties that are specific to the HTML Help compiled help transformation target.

Project Ant Property	Description
<code>args.htmlhelp.includefile</code>	Specifies the name of a file that you want included in an HTMLHelp project.

JavaHelp-specific Ant Properties

The following table describes Ant properties that are specific to the JavaHelp transformation target.

Project Ant Property	Description
<code>args.javahelp.map</code>	Specifies the name of the ditamap file for a JavaHelp project.
<code>args.javahelp.toc</code>	Specifies the name of the file containing the TOC in your JavaHelp output. Default value is the name of the ditamap file for your project.

Other Toolkit Ant Properties

The following table describes additional Ant properties specific to the DITA Open Toolkit that you may override. You should not override a DITA-OT Ant property if it does not appear in this table or one of the tables above.

DITA Ant Property	Description
<code>args.dita.locale</code>	Specifies the language locale file to use for sorting index entries. The JavaHelp transformation type also uses this parameter.
<code>clean.temp</code>	Specifies whether DITA-OT should delete the files in the temporary directory, <code>dita.temp.dir</code> , when it finishes a build. Allowed values are "yes" and "no". Default value is "yes".
<code>dita.dir</code>	The location of your DITA-OT installation. Verify that your project's build script points to the correct location.
<code>dita.extname</code>	Deprecated, <code>dita.ext</code> parameter should be used instead. Defines the file extension for content files in the directory specified with the <code>dita.temp.dir</code> property. Allowed values are ".xml" and ".dita"; Default value is ".xml".
<code>dita.preprocess.reloadstylesheet</code>	Instructs the toolkit to reload the XSL stylesheets used for transformation. Allowed values are "true" and "false". Default value is "false". Tip: Set the value to true if you want to use more than one set of stylesheets to process a group of topics. The parameter is also useful for writers of toolkit build scripts who experience Java memory problems during transformation due to large Ant
<code>dita.preprocess.reloadstylesheet.conref</code>	
<code>dita.preprocess.reloadstylesheet.mapref</code>	
<code>dita.preprocess.reloadstylesheet.mappull</code>	
<code>dita.preprocess.reloadstylesheet.maplink</code>	
<code>dita.preprocess.reloadstylesheet.topicpull</code>	

DITA Ant Property	Description
	projects. Alternatively, you can adjust the size of your Java memory heap if setting <code>dita.preprocess.reloadstylesheet</code> for this reason.
<code>dita.temp.dir</code>	Defines the directory where DITA-OT will create a temporary directory to place temporary files generated during the transformation process.

Running DITA-OT from command-line tool

The DITA Open Toolkit provides a command-line tool as an alternative for users with little knowledge of Ant. Most parameters available to the Ant builds are also available using the command-line tool.



Important: The command-line tool interface is simply a wrapper around the Ant interface; it takes the simplified parameters as input, converts them to Ant parameters, and then runs an Ant build. This means that applications embedding the toolkit should always invoke Ant directly. For individual builds, the additional Java overhead is minimal, but for repeated or server based builds, it the extra memory usage will become more of an issue.

Running command-line tool

If you are using the "Full Easy Install" package, running the startcmd batch file will set up a build environment for you and put you in the correct directory. If you are not using this method, you must set up all of your tools (Ant, XSLT, FOP, etc) before running the build.

1. Change into the DITA Open Toolkit installation directory.
2. On the command-line, enter the following command:

```
java -jar lib/dost.jar /i:samples/sequence.ditamap /outdir:out /
transtype:xhtml
```

This particular example calls Ant to build the sample `sequence.ditamap` file to XHTML. The output is placed in the `out/` directory.

Note:

1. In this example, the character slash preceded by a space is the separator for each parameter.
2. Currently, the parameters `/filter`, `/ftr`, `/hdr`, and `/hdf` require an absolute path.
3. The properties file is saved in the `${args.logdir}` directory. The following command provides an example using this properties file:

```
ant -propertyfile ${args.logdir}/property.temp
```

4. To see a list of all supported parameters from the command-line tool, run the following command with no additional parameters:

```
java -jar lib/dost.jar
```

Supported parameters

See [Command-line tool arguments for DITA-OT](#) on page 44 for supported command-line tool arguments. To get a full list of arguments, run

```
java -jar lib/dost.jar -help
```

Command-line help

You can find the version of toolkit and the usage of the command-line from the command line help by using the following commands:

```
java -jar lib/dost.jar -version
java -jar lib/dost.jar -h
```

You can see the brief description of the supported parameters in the command-line window when you type a specified command.



Command-line tool arguments for DITA-OT

Reference list of DITA-specific command line tool arguments.

Parameters available to all transforms

The following common parameters are available for all transformation output types.

Table 7: Common DITA-OT parameters

Argument	Description
/debug	Specifies that DITA-OT print debugging information for your project. Allowed values are "yes" and "no". Default value is "no".
/draft	<p>Indicates whether draft-comment and required-cleanup elements are included in the generated file. Corresponds to XSLT parameter DRAFT in most XSLT modules. Allowed values are "yes" and "no". Default value is "no".</p> <p> Tip: For PDF output, setting <code>\${args.draft}</code> to "yes" will also cause the contents of <code><titlealts></code> to appear below the title.</p>
/grammarchache	<p>Specifies whether to use the grammar caching feature of the XML parser. Allowed values are "yes" and "no". Default value is "yes".</p> <p> Note: For most users, this is an important option that dramatically speeds up processing time. However, there is a known problem with using this feature for documents that use XML Entities. If your build fails with parser errors about entity resolution, try setting this parameter to "no".</p>
/i	Typically defines the location of the .ditamap file for your documentation project. However, the property can be set to a .dita file, as well. DITA-OT reads this file to find the .dita files that comprise the content for the documentation project.
/logdir	Defines the location where DITA-OT places log files for your project.
/outext	Specifies the file extension for HTML files in your project's output. Corresponds to XHTML parameter OUTEXT. Default values is ".html".

Argument	Description
/basedir	The directory where your project's ant build script resides. The DITA-OT will look for your .dita files relative to this directory. DITA-OT's default build script sets this as an attribute of the project, but you can also define it as a project property.
/filter	Defines the location of your project's filter file. Filter files end with the .ditaaval suffix and are used to filter, include and exclude, content in the generated document. Alternatively, you can create multiple versions of your document by creating a different .ditamap file for each version.
/outdir	The location of the directory to hold output from your documentation project.
/transtype	Defines the output type for a specific Ant target. Plugins may add new values for this option; by default, the following values are available: <ul style="list-style-type: none"> pdf xhtml htmlhelp eclipsehelp eclipsecontent odt troff rtf javahelp legacypdf docbook
/validate	Specifies whether DITA-OT should validate your content files. Allowed values are "yes" and "no". Default value is "yes".



Parameters available for all XHTML based transforms


The following parameters are available for all output types that are based on the XHTML transform type, including:

- XHTML
- HTMLHelp
- JavaHelp
- eclipsehelp

Table 8: XHTML and related parameters

Argument	Description
/artlbl	Adds a label to each image containing the image's filename. Allowed values are "yes" and "no". Default value is "no".
/copycss	Indicates whether you want to copy your own .css file to the output directory.
/args.css	The name of your custom .css file.


Argument	Description
<code>/csspath</code>	The location of your copied .css file relative to the output directory. Corresponds to XSLT parameter CSSPATH.
<code>/cssroot</code>	The directory that contains your custom .css file. DITA-OT will copy the file from this location.
<code>/ftr</code>	<p>Specifies the location of a well-formed XML file containing your custom running-footer for the document body. Corresponds to XSLT parameter FTR.</p> <p> Note: The fragment must be valid XML, with a single root element, common practice is to place all content into <div>.</p>
<code>/usetasklabels</code>	<p>Specifies whether to generate locale-based default headings for sections within task topics. Allowed values are "YES" and "NO". Default value is "NO". Corresponds to the XSLT parameter GENERATE-TASK-LABELS.</p>
<code>/hdf</code>	Specifies the location of a well-formed XML file to be placed in the document head.
<code>/hdr</code>	<p>Specifies the location of a well-formed XML file containing your custom running-header for the document body. Corresponds to XSLT parameter HDR.</p> <p> Note: The fragment must be valid XML, with a single root element, common practice is to place all content into <div>.</p>
<code>/indexshow</code>	Indicates whether <code>indexterm</code> element should appear in the output. Allowed values are "yes" and "no". Default value is "no".
<code>/xhtmlclass</code>	Specifies whether to include DITA class ancestry inside generated XHTML elements. Allowed values are "no" and "yes"; the default is "yes" in release 1.5.2 (it was "no" in 1.5 and 1.5.1). For example, the <code>prereq</code> element in a task (which is specialized from section) would generate <code>class="section prereq"</code> . Corresponds to the XSLT parameter PRESERVE-DITA-CLASS.
<code>/xsl</code>	Specifies an XSL file that is used rather than the default XSL transform, located in <code>toolkitdir\xsl\dita2xhtml.xsl</code> . Property must specify the full path and XSL file name.
<code>/generateouter</code>	<p>Specifies whether to generate files for content files that are not located in or beneath the directory containing your <code>ditmap</code> file. Supported values are:</p> <ul style="list-style-type: none"> • "1" (default) – do not generate outer files • "2" – generate outer files • "3" – old solution.
<code>/onlytopicinmap</code>	Specifies whether files that are linked to, or referenced with a <code>conref</code> attribute, should generate output. If set to

Argument	Description
<code>/outercontrol</code>	<p>"yes", only files that are referenced directly from the map will generate output files.</p> <p>Specifies whether content files are located in or below the directory containing your <code>.ditamap</code> file. Supported values are:</p> <ul style="list-style-type: none"> "fail" – fail quickly if files are going to be generated/copied outside of that directory "warn" (default) – complete if files will be generated/copied outside, but log a warning "quiet" – quietly finish with only those files (no warning or error). <p>The <code>gen-list-without-flagging</code> Ant task generates a harmless warning for content outside the map directory; you can suppress these warnings by setting the <code>outer.control</code> property to "quiet".</p> <p> Warning: Microsoft HTML Help Compiler cannot produce HTMLHelp for documentation projects that use outer content. Your content files must reside in or below the directory containing the <code>.ditamap</code> file, and the map file cannot specify <code>".."</code> at the start of <code>href</code> attributes for <code>topicref</code> elements.</p>

PDF-specific command line tool options

The following table describes command line tool options that are specific to the PDF transformation type.

Table 9: PDF parameters

Argument	Description
<code>/foincluderellinks</code>	<p>Specifies which links to include in the PDF file. Values are:</p> <ul style="list-style-type: none"> "none" (default) – no links are included. "all" – all links are included. "nofamily" – hard coded links and reltable-based links are included. Parent, child, next, and previous links are not included.
<code>/fooutputrellinks</code>	<p>Specifies whether to show links in your project's output. Values are "yes" (include all links) and "no" (the default, include no links). If <code>\${args.fo.include.rellinks}</code> is specified, this parameter is ignored.</p> <p> Note: This parameter is deprecated in favor of <code>\${args.fo.include.rellinks}</code>.</p>
<code>/xslpdf</code>	<p>Specifies an XSL file that is used in place of the default XSL transform at <code>demo\fo\xsl\fo\topic2fo_shell.xsl</code>. You must specify the full path and XSL file name.</p>

Argument	Description
/fouserconfig	The parameter to specify the user configuration file for FOP.
/retaintopicfo	Specifies whether to leave the generated FO file for a PDF project.

ODT-specific command line tool options

The ODT transform, which produces a document using the Open Document Format, is available in the 1.5.2 version of the DITA-OT.

Table 10: ODT related parameters

Argument	Description
/odtimgembed	Determines whether images are embedded as binary objects within the ODT file.
/odtincluderellinks	Specifies which links to include in the ODT file. Values are: <ul style="list-style-type: none"> "none" (default) – no links are included. "all" – all links are included. "nofamily" – hard coded links and reltable-based links are included. Parent, child, next, and previous links are not included.

EclipseContent-specific command line tool options

The "eclipsecontent" transform type produces normalized DITA files, along with Eclipse TOC and project files.

Table 11: EclipseContent options

Argument	Description
/eclipsecontenttoc	Specifies the name of the TOC file for an Eclipse Content project.

XHTML-specific command line tool options

Parameters in this section are used by the "xhtml" transtype, but not by other XHTML based transforms.



Table 12: Options for the "xhtml" transform type

Argument	Description
/xhtmltoc	Specifies the name of the entry point for an XHTML project. Default value is <code>index.html</code>

EclipseHelp-specific command line tool options

The following table describes command line tool options that are specific to the EclipseHelp transformation type, which is an XHTML based output for use with the Eclipse Help System.

Argument	Description
/eclipsehelptoc	Specifies the name of the TOC file.

Argument	Description
/provider	Specifies the name of the person or organization providing an Eclipse Help project. Default value is DITA.  Tip: The toolkit ignores the value of this property when processing an Eclipse Collection Map, <code>eclipse.dtd</code> .
/version	Specifies the version number to include in the output. Default value is 0.0.0.  Tip: The toolkit ignores the value of this property when processing an Eclipse Collection Map, <code>eclipse.dtd</code> .

HTMLHelp-specific command line tool options

The following table describes command line tool options that are specific to the HTML Help compiled help transformation target.

Argument	Description
/htmlhelpincludefile	Specifies the name of a file that you want included in an HTMLHelp project.

JavaHelp-specific command line tool options

The following table describes command line tool options that are specific to the JavaHelp transformation target.

Argument	Description
/javahelpmap	Specifies the name of the ditamap file for a JavaHelp project.
/javahelptoc	Specifies the name of the file containing the TOC in your JavaHelp output. Default value is the name of the ditamap file for your project.

Other Toolkit command line tool options

The following table describes additional command line tool options specific to the DITA Open Toolkit that you may override.

Argument	Description
/ditalocale	Specifies the language locale file to use for sorting index entries. The JavaHelp transformation type also uses this parameter.
/cleantemp	Specifies whether DITA-OT should delete the files in the temporary directory, <code>dita.temp.dir</code> , when it finishes a build. Allowed values are "yes" and "no". Default value is "yes".
/ditadir	The location of your DITA-OT installation. Verify that your project's build script points to the correct location.
/ditaext	Deprecated, <code>dita.ext</code> parameter should be used instead. Defines the file extension for content files in the

Argument	Description
<code>/tempdir</code>	<p>directory specified with the <code>dita.temp.dir</code> property. Allowed values are ".xml" and ".dita"; Default value is ".xml".</p> <p>Defines the directory where DITA-OT will create a temporary directory to place temporary files generated during the transformation process.</p>

Configuration

Reference list of DITA-OT configuration properties.

The `lib/configuration.properties` file is used to store configuration properties for DITA-OT and plug-ins. The configuration properties are available to both Ant and Java processes, but unlike argument properties, they cannot be set at run-time.

The `lib/org.dita.dost.platform/plugin.properties` file is used to store configuration properties set by the integration process. The file is regenerated from plug-in configuration each time the integration process is run and should not be edited manually.

Common properties

<code>default.language</code>	Default language. Allowed values are those defined in IETF BCP 47, Tags for the Identification of Languages .
-------------------------------	---

PDF properties

<code>org.dita.pdf2.index.frame-markup</code>	<p>FrameMaker index syntax processing. Allowed values are</p> <ul style="list-style-type: none"> "true" — FrameMaker index syntax processing "false" (default) — normal DITA 1.2 index syntax processing
<code>org.dita.pdf2.i18n.enabled</code>	<p>I18N font processing. Allowed values are</p> <ul style="list-style-type: none"> "true" (default) — I18N processing is enabled "false" — I18N processing is disabled
<code>org.dita.pdf2.use-out-temp</code>	<p>Legacy temporary file mode for generating <code>topic.fo</code> and graphics into output directory. Allowed values are</p> <ul style="list-style-type: none"> "true" — use output directory for XSL FO file processing "false" (default) — use tempory folder for XSL FO processing

Available DITA-OT Transforms

The primary purpose of the DITA Open Toolkit is to transform DITA maps and topics into other formats. The toolkit ships with several native transforms, and can be extended to support any other output format with by adding a new plug-in.

DITA to XHTML

The XHTML transform was the first transform created for the DITA Open Toolkit; it converts DITA topics into XHTML documents. In addition to the XHTML output, this transform also returns a simple table of contents file named `index.html`, which is based on the structure of the input map file.

XHTML output is always associated with the default DITA-OT CSS stylesheet `"commonltr.css"` (or `"commonrtl.css"` for right-to-left languages). Parameters are available to override the default CSS styling.

To run the default XHTML transform, set the transform type parameter to `"xhtml"`. Many of the other transform types run the same conversion to XHTML, followed by additional routines to create new navigation files.



Note: When running the demo script `build_demo.xml`, setting the output type to `"web"` will run the default XHTML transform.

DITA to Eclipse help

Eclipse output is an XHTML based output format used by the Eclipse Help system.

To create a documentation plug-in for the Eclipse platform, set the transform type to `"eclipsehelp"`. In addition to the standard XHTML and CSS files, this transform will generate the following output files:

- `plugin.xml`, the required control file for any Eclipse plug-in.
- An Eclipse table of contents, based on the name of the input map (`mapname.ditamap` will generate `mapname.xml`).
- An Eclipse index file with any index entries from the content, in a file named `index.xml`.
- Additional control files for Eclipse named `plugin.properties` and `META-INF/MANIFEST.MF`.

DITA to TocJS

The `"tocjs"` transform type is an XHTML based output type that also generates a frameset and a JavaScript based table of contents with expandable and collapsible entries. The `tocjs` transform type was originally created by Shawn McKenzie as a plug-in to the toolkit, and later bundled with the default distribution.

The original `tocjs` output required an Ant build with several parameters in order to function properly; it also required a separate transform to XHTML in order to produce content. Beginning with DITA-OT release 1.5.4, the `tocjs` transform type was updated to always produce XHTML output and to use a default frameset when one is not already specified.

In release 1.5.4 the `tocjs` transform was also added to the `build_demo.xml` script as an option for the output type.

DITA to PDF (PDF2)

The DITA-OT transform to PDF was originally created as a plug-in and maintained outside of the main toolkit code. It was created as a more robust alternative to the demo PDF transform in the original toolkit, and thus was known as PDF2. The plug-in was bundled into the default toolkit distribution with release 1.4.3, and is run when setting the transform type to `"pdf"` or `"pdf2"`.

DITA to HTML Help (CHM)

The `"htmlhelp"` transform type will generate HTML output, along with the control files needed to produce a Microsoft® HTML Help file.

The HTML Help output produces HTML files rather than XHTML files. In addition to the HTML output and CSS files, this transform type will return the following files, all based on the name of the input map:

- Table of Contents (`mapname.hhc`)
- Sorted index (`mapname.hhk`)
- HTML Help project file (`mapname.hhp`)
- If the HTML Help compiler is located on the system, the project will be compiled to create `mapname.chm`

DITA to ODT (Open Document Type)

The "odt" transform type produces output files that use the Open Document format, which is used by tools such as Open Office.

This transform returns an ODT document, which is a zip file that contains the ODF XML file (`content.xml`), referenced images, and default styling (in the file `styles.xml`).

DITA to Docbook

The Docbook output routine (transform type "docbook") was converts DITA maps and topics into a Docbook output file. Complex DITA markup may not be supported by this process, but the transform does support most common DITA structures.

DITA to Troff

The "troff" transform produces output files for use with the Troff viewer on many Unix-style platforms, particularly for programs such as the Man page viewer. Each DITA topic document generally corresponds to one troff output file.

The troff output works for most common DITA structures, but does not support table or simbletable elements. Most testing of troff output was done against the Cygwin Linux emulator.

DITA to Word output transform

The "wordrtf" transform type will produce an RTF file for use by Microsoft® Word.

The whole structure of the output file is the same as the structure designed in the ditamap file of the DITA source files. To avoid losing files in the final output, make sure the ditamap file contains all topics which are cross referenced from within any individual topics.

Limitations

1. You can change the styles of the output file by using tools in Microsoft® Word rather than specifying the styles before transforming.
2. If there is a cross reference referring to an URL in the DITA source file, the link should be completed defined with the proper internet protocol. For example, specify `http://www.ibm.com` instead of `www.ibm.com`.
3. Flagging, revision bar and filtering are not supported in Word RTF output.
4. Style attributes for table are not supported in Word RTF output.
5. Complex cases dealing with tables in lists are not supported in Word RTF.
6. There may be no output style applied on contents of some tags in Word RTF output compared with other output.

Transforming DITA to Word with Ant script

1. Specify a directory where you want to put the output files. For example, `e:/output/dita2word`.
2. Open an ant script file from the default model template. You can find the file "template_wordrtf.xml" under the "ant" directory.
3. Modify the transformation type to Word.
The default file name is "ant.xml". If you want to save the template file, make sure to save it as another file name. For example, `antfile.xml`.
4. On the command line, enter `ant -f <ant.xml>`.
5. After processing and generating, a single output file in ".rft" format occurs in the specified directory, such as in `e:/output/diat2word`.
6. You can use Microsoft® Word to open the output file.
7. You can also further edit the output file by using tools in Microsoft® Word.

Transforming DITA to Word with Java™ command

1. On the command line, enter the command `java -jar lib\dost.jar /i:<input> /outdir:<out> /transtype:wordrtf`.

<input> means the name of ditamap file to be transformed, and <out> means the output directory.

2. After processing and generating, a single output file in .rft format occurs in the specified directory, such as in `e:/output/diat2word..`
3. You can use Microsoft® Word to open the output file.
4. You can also further edit the output file by using tools in Microsoft® Word.

DITA to Eclipse Content

The "eclipsecontent" transform type was originally designed to work with an Eclipse plug-in for dynamically rendering DITA content. It returns Eclipse control files along with normalized DITA files (with preprocessing complete and default attributes included).

The normalized DITA files, which use an extension of ".xml", may also be used by other scripts that work with DITA; they have all pre-processing resolved, including modifications such as:

- Map-based links are added to the topics
- Link text is resolved (based on empty elements like `<xref href="othertopic.dita"/>`)
- DTD or Schema reference is removed
- Class attributes, which are defaulted in the DTD / Schema, are made explicit in the topics
- Map attributes that cascade are made explicit on child elements

DITA to legacypdf (Deprecated)

The first few versions of the toolkit came with a demo PDF build, which was eventually replaced by the much more robust PDF Plug-in (also known as PDF2). This code is no longer maintained by the DITA-OT developers.

The demo PDF transform was deprecated in DITA-OT release 1.4.3, when it was replaced by the PDF Plug-in. The older script is still included in order to support older customizations or build scripts that extended the code; however, the transform type must be set to "legacypdf".

Migrating HTML to DITA

The HTML to DITA migration tool ships in the `demo/` directory of the toolkit, and does not make use of the common toolkit processing for DITA content.

The DITA Open Toolkit release 1.2 or above provides a HTML to DITA migration tool, which migrates HTML files to DITA files. This migration tool originally comes from the developerWorks publication of Robert D. Anderson's how-to articles with the original h2d code. This migration tool is under `demo\h2d` directory. You can use it separately because it is not integrated into the main transformation of toolkit. The version in the toolkit is more recent, but the articles should be referenced for information on details of the program, as well as for information on how to extend it. There are links to the articles at the bottom of this page.

Preconditions

The preconditions to be considered before using the migration tool are listed below:

- The HTML file content must be divided among concepts, tasks, and reference articles. If not, the HTML files should be reworked before migrating.
- This migration tool is intended for topics. The HTML page should contain a single section without any nested sections.
- DITA architecture is focused on topics, information that is written for books needs to be redesigned in order to fit into a topic-based architecture.
- This migration utility only works with valid XHTML files, HTML files must be cleaned up using HTML Tidy or other utility before processing.

Post conditions

There are also some post conditions to consider after processing:

- In some case, the tool cannot determine the correct way to migrate, it places the contents in a <required-cleanup> element, you should fix such elements in the output DITA files.
- Check the output DITA files. Compare them with the source HTML files and check if both contents are equivalent.

Known limitations

1. Since Xalan doesn't allow to set the public and system IDs dynamically using a variable, when Xalan is used as the default XSLT processor, the output will contain:

```
<!DOCTYPE topic PUBLIC "{$publicid}" "{$systemid}">
```

Suggest to use Saxon as the processor to fix this problem. For other information on this problem, see the section "Other general migration notes" in the first developerWorks article.

Extension points

The HTML2DITA migration tool helps extension in the following listed ways:

- The `genidattribute` template can be overridden to change the method for creating the topic ID.
- The `gentitlealts` template can be overridden to change the ways of title generation.
- Override respond section in the tool to preserve the semantic of source, in case if the <div> or element is used in regular structures.
- You can also migrate to another specialized DTD by overriding the original template base on the specific DTD and your required output.

Migrating HTML to DITA with Ant script

Running example

1. Start the command window.
2. Navigate to the directory of the migration tool.
3. Use ant script to run the migration, on the command line, enter the following command:

```
ant -Dargs.input={file|direcotry} -Dargs.output={direcotry} -  
Dargs.infotype={topic|concept|task|reference}
```



Note: The namespace problem listed in [Known Limitations](#) has been fixed by adding a new preprocess step in the script in release 1.2.1.



You can also add other parameters to the command. See the following table for details.

Supported Parameters

The following table lists the supported parameters that you can set with the ant script.

Table 13: Table of supported parameters

Parameter	Descriptoin	Required
args.input	The input of the migration. It can be a file or directory. Default is current directory.	No.
args.include.subdirs	The parameter to specify if sub directories under the input directory is included. "yes" and "no" are valid values. Default is "no".	No.

Parameter	Description	Required
	 Note: Any value that is not "yes" is regarded as "no".	
args.output	The output directory of generated DITA files. Default is the current directory.	No.
args.infotype	The infotype of generated DITA files, topics, concept, task, and reference are valid values. Default is topic.	No.
args.dita.ext	The extension of generated DITA files. This extension also used to convert links that go to other DITA topic. ".dita" and ".xml" are valid values. Default is ".dita".	No.
args.xsl	The xsl file to replace the default xsl file.	No.
args.lang	The default language of output DITA files. Default is "en-us".  Note: For supported language, please refer to <code>strings.xml</code> under the directory <code>\${ditaot_dir}/xsl/common</code> .	No.

Migrating HTML to DITA with Java command

Running example

1. Start the command window.
2. Navigate to the directory of the migration tool.
3. (Optional) If the input HTML file contains namespace, you can remove it by hand, or running the command below:

enter the following command when using Saxon:

```
java com.icl.saxon.StyleSheet mytask.htm preprocess.xsl > mytask.htm
```

enter the following command when using Xalan:

```
java org.apache.xalan.xslt.Process -in mytask.htm -xsl preprocess.xsl -out mytask.htm
```

4. Use Saxon or Xalan directory to run the migration, on the command line,

enter the following command when using Saxon:

```
java com.icl.saxon.StyleSheet mytask.htm h2d.xsl infotype=task > mytask.dita
```

enter the following command when using Xalan:

```
java org.apache.xalan.xslt.Process -in mytask.htm -xsl h2d.xsl -out
mytask.dita -param infotype task
```

You can also add other parameters to this properties file. See the following table for details.



Note: The output directory of the generated DITA file should exist, since the XSLT processor can't create it automatically.

Supported parameters

The following table lists the supported parameters that you can set with the java command.

Table 14: Table of supported parameters

Parameter	Description	Required
infotype	The infotype of generated DITA files. Topic, concept, task, and reference are valid values. Default is topic.	No.
dita-extension	<p>The extension for links that go to other DITA topics. ".dita" and ".xml" are valid values. Default is ".dita".</p> <p> Note: The extension of the generated DITA file can't specified by this parameter, it only can be specified along with the output filename.</p>	No.
FILENAME	<p>It is used to determine the main topic's ID.</p> <p> Note: The FILENAME should ends with '.htm' or '.html'. Invalid ID characters, including all numbers, will be replaced with letters.</p>	No.
default-lang	<p>The default language of output DITA files. Default is "en-us".</p> <p> Note: For supported language, please refer to <code>strings.xml</code> under the directory <code>\${ditaot_dir}/xsl/common</code>.</p>	No.

Problem determination and log analysis

Introduction

The DITA-OT command-line tool has a logging method to log messages both on the screen and into the log file. The messages on the screen present user with the status information, warning, error, and fatal error messages. By analyzing these messages, user can know what cause the problem and how to solve it. In the command-line tool the logger use is specified internally, so you do not need to specify it.

When running DITA-OT directly with Ant, different loggers provided by Ant can be used to get different log output. See Ant documentation for more information.

Analyze messages on the screen

During the building process, some information or messages occur on the screen to tell you about the status, warnings, errors, or fatal errors. You need to analyze the messages to solve the problems.

- If the build succeeded with some warning messages on the screen, it means that there are something incorrect within the user input parameters or source DITA files; but you can still get the correct output.
- If the build succeeded with some error messages on the screen, it means that there are something incorrect within the user input parameters or source DITA files; the output maybe not correct.
- If the build failed with fatal error message on the screen, it means that there are something illegal or invalid within the user input parameters or source DITA files; you may get no output, or wrong output.

Analyze messages in the log file

A log file in plain text format is generated in the log directory, which has a name combined with both input file name and transform type. You can open it and find more detailed information, which are helpful for solving problems. You can use the same way introduced above to analyze the messages and solve the problems.

The log directory can be specified by:

- using Ant, with argument `-logfile=log-file`
- using command-line tool, the parameter `/logdir:log-dir`.

Turn on debug mode

Under debug mode, diagnostic information, such as: environment variables, stack trace, will be logged into the log file. These information can help the user or developer to go deep into the problems and find the root cause.

By default, the debug mode is disabled. To turn on the debug mode on, you need to follow the usage below:

- Append `-v` and `-Dargs.debug=yes` in Ant command.
- Append `/d` or `/debug` in command-line tool.

About message file

The message file is used to store the detailed log messages, these messages are read dynamically from this file. To ensure those messages can be read correctly during the transform process, the message file should be located properly. In some situations, the toolkit may fails to load the message file due to some exceptions thrown. Please refer to [Troubleshooting](#) on page 57 for detailed information.

For high level users and developers, there is a property `args.message.file` in the toolkit's ant script, it is used to config the message file, you can override it in your ant script.

Troubleshooting

This section is used for identifying problems when installing and executing the DITA Open Toolkit.

Out of Memory Error

In some cases, you might receive a message stating the build has failed due to an "Out of Memory" error. In many cases this can be solved by switching from Xalan to Saxon as the default processor. Recent versions of the "Full Easy Install" toolkit distribution ship with Saxon instead of Xalan.

If that does not work, please follow the steps below to fix this problem:

1. For Windows, type `set ANT_OPTS=%ANT_OPTS% -Xmx256M` in the command prompt before running a build. Alternatively, you can add the value `-Xmx256M` to the `ANT_OPTS` environment variable.

For Linux, type `export ANT_OPTS=$ANT_OPTS -Xmx256M` in the command prompt before running a build.

2. Run the transformation again.

java.io.IOException: Can't store Document

In some cases, when you run the JavaHelp transformation, you might receive the exception above. This problem is caused by some HTML files unrelated with the current JavaHelp transformation were found under the output directory. Please follow the steps below to fix this problem:

1. Change into the output directory.
2. Clean the output directory.
3. Run the JavaHelp transformation again.

Failed to load message file

In some situations, the toolkit may fails to load the message file `messages.xml` and begin to throw exceptions.

To fix this problem, you need to check if the files `resource/messages.xml` and `resource/messages.dtd` exist in the toolkit. If not, please copy them from the toolkit's root directory.

Commas in file names

Commas in file names will cause trouble during the processing because Ant uses commas as the delimiters when processing batch files in a list. Please prevent using commas in the name of DITA files.

Stack Overflow

Sometimes, you will receive an error during the transformation which says the stack memory overflows. Please follow the steps below to fix the problem:

1. For Windows, type `set ANT_OPTS=%ANT_OPTS% -Xms512M` in the command prompt, you can also choose to add a new option `-Xms512M` to the `ANT_OPTS` environment variable.

For Linux, type `export ANT_OPTS=$ANT_OPTS -Xms512M` in the command prompt.

2. Run the transformation again.

Chapter

3

Developer Reference

Topics:

- [*DITA Open Toolkit Architecture*](#)
- [*Extending the DITA Open Toolkit*](#)
- [*Installing DITA-OT plug-ins*](#)
- [*Creating DITA-OT plug-ins*](#)
- [*Implementation dependent features*](#)
- [*Extended functionality*](#)
- [*Topic merge*](#)
- [*Creating Eclipse help from within Eclipse*](#)

DITA Open Toolkit Architecture

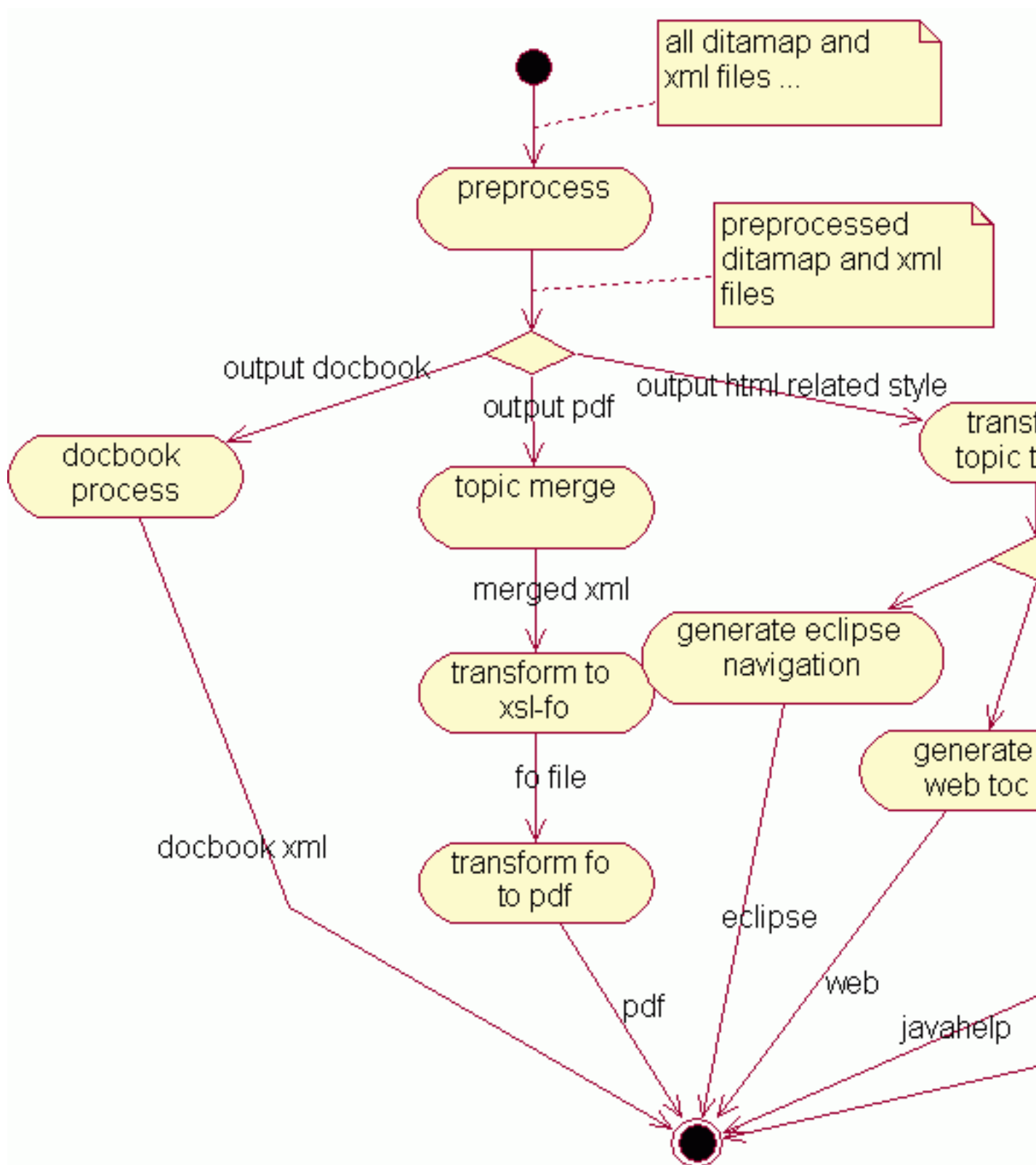
The DITA Open Toolkit is an open source implementation of the OASIS DITA Technical Committee's specification for Darwin Information Typing Architecture (DITA) DTDs and schemas. The toolkit uses open source solution of ANT, XSLT (currently 1.0 for most processing, with XSLT 2.0 for PDF) and Java to implement transformation functions from DITA content (maps and topics) into different deliverable formats.

Processing structure

The DITA Open Toolkit implements a multi-stage, map-driven architecture to process DITA content. Each step in the process examines some or all of the content; some steps result in temporary files used by later steps, while others result in updated copies of the DITA content. Most of the processing takes place in a temporary working directory (the source files themselves are never modified).

Transformations in the toolkit are designed like a pipeline. Most of the pipeline is common between all output formats, and is known as the "pre-process" stage. In general, any DITA process begins with this common set of pre-processing routines. Once the pre-processing is completed, the pipeline diverges based on the desired output format. Some processing is still common to multiple output formats; for example, Eclipse Help and HTML Help both use the same routines to generate XHTML topics, after which the two pipelines branch to create a different set of navigation files.

The following image illustrates how the pipeline works for some common output types.



Processing modules in the DITA-OT

The DITA-OT processing pipeline is implemented using Ant. Individual modules within the Ant script are generally implemented in either Java or XSLT, depending on a variety of factors, such as performance or requirements for

customization. Virtually all Ant and XSLT modules are extensible by adding a plug-in to the toolkit; new Ant targets may be inserted before or after common processing, and new rules may be imported into common XSLT modules to override default processing.

XSLT modules are all set up using shell files. Typically, each shell file begins by importing common rules that apply to all topics. This set of common processing may in turn import additional common modules, such as those used for reporting errors or determining the document locale. After the common files, additional imports may be included in order to support processing for DITA Specializations.

For example, XHTML processing is controlled by the `dita2xhtml.xsl` file inside the `xsl\` directory. The shell begins by importing common rules applicable to all general topics (`xslhtml\dita2htmlImpl.xsl`). After that, additional XSLT overrides are imported for specializations that require modified processing. For example, an override for reference topics is imported in order to add default headers to property tables. Additional modules are imported for tasks, for the highlighting domain, and for several other standard specializations. After the standard XSLT overrides, plug-ins may add in additional processing for local styles or for additional specializations.

Java modules are typically used when XSLT is a poor fit, such as for processes that work directly with the file system to copy files, or which make use of standard Java libraries (like those used for index sorting). Java modules are also used in many cases where a step involves copying files, such as the initial process where source files are parsed and copied to a temporary processing directory.

Processing order within the DITA-OT

The order of processing is often significant when evaluating DITA content. Although the DITA specification does not mandate a specific order for processing, the toolkit has over time found that the current order best meets user expectations. Switching the order of processing, while legal, may give different results.

For example, if `conref` is evaluated before filtering, it is possible to reuse content that will later be filtered out of its original location. However, we have found that filtering first provides several benefits. For example, the following `<note>` element uses `conref`, but also contains a `product` attribute:

```
<note conref="documentA.dita#doc/note" product="MyProd"/>
```

If the `conref` attribute is evaluated first, then `documentA` must be parsed in order to retrieve the note content. That content is then stored in the current document (or in a representation of that document in memory). However, if all content with `product="MyProd"` is filtered out, then that work is all discarded later in the build.

However, if the filtering is done first as in the toolkit, this element is discarded immediately, and `documentA` is never examined. This provides several important benefits:

- Time is saved simply by discarding unused content as early as possible; all future steps can load the document without this extra content.
- More significant time is saved in this case by not evaluating the `conref` attribute; in fact, `documentA` does not even need to be parsed.
- Any user reproducing this build does not need `documentA`. If the content is sent to a translation team, that team can reproduce an error-free build without `documentA`; this means `documentA` can be kept back from translation, preventing accidental translation and increased costs.

If the order of these two steps is reversed, so that `conref` is evaluated first, it is possible that results will differ. For example, on the sample above, the `product` attribute will override a `product` setting on the referenced note. Now assume that the note in `documentA` is defined as follows:

```
<note id="note" product="SomeOtherProduct">This is an important note!</note>
```

A process that filters out `product="SomeOtherProduct"` will remove the target of the original `conref` before that `conref` is ever evaluated -- resulting in a broken reference. Evaluating `conref` first would resolve the reference, and only later filter out the target of the `conref`. While some use cases can be found where this is desirable, benefits such as those described above resulted in the current processing order.

DITA-OT pre-processing architecture

This topic describes the set of steps commonly known as the pre-processing stage of a DITA build. These steps typically run at the start of any build using the DITA-OT, regardless of the final output format.

Each step described corresponds to one Ant target in the build pipeline. The general Ant target "preprocess" will call all of the targets described here.

Generate lists (gen-list)

The `gen-list` step examines the input files and creates lists of topics, images, document properties, or other content. These lists are used by later steps in the pipeline. For example, one list includes all topics that make use of the `conref` attribute; only those files are processed during the `conref` stage of the build. This step is implemented in Ant and Java.

The result of this list is a set of several list files in the temporary directory, including `dita.list` and `dita.xml.properties`.

List file property	List file	List property	Usage
canditopicfile	canditopic.list	canditopiclist	
chunkedditamapfile	chunkedditamap.list	chunkedditamaplist	
chunkedtopicfile	chunkedtopic.list	chunkedtopiclist	
codereffile	coderef.list	codereflist	topics with coderef
conreffile	conref.list	conreflist	Documents that contains conref attribute that need to be resolved in preprocess.
conrefpushfile	conrefpush.list	conrefpushlist	
conreftargetsfile	conreftargets.list	conreftargetslist	
copytosourcefile	copytosource.list	copytosourcelist	
copytotarget2sourcemapfile	copytotarget2sourcemap.list	copytotarget2sourcemaplist	
flagimagefile	flagimage.list	flagimagelist	
fullditamapandtopicfile	fullditamapandtopic.list	fullditamapandtopiclist	All of the ditamap and topic files that are referenced during the transformation. These may be referenced by href or conref attributes.
fullditamapfile	fullditamap.list	fullditamaplist	All of the ditamap files in dita.list
fullditatopicfile	fullditatopic.list	fullditatopiclist	All of the topic files in dita.list
hrefditatopicfile	hrefditatopic.list	hrefditatopiclist	All of the topic files that are referenced with an href attribute
hreftargetsfile	hreftargets.list	hreftargetslist	link targets
htmlfile	html.list	htmllist	resource files
imagefile	image.list	imagelist	Images files that are referenced in the content

List file property	List file	List property	Usage
keyfile	key.list	keylist	List of keys. The format is: <pre>keyname "=" link "(" source ")"</pre> Both href and source URLs are relative to base directory.
keyreffile	keyref.list	keyreflist	Topics and maps which have key references.
outditfilesfile	outditfiles.list	outditfileslist	
relflagimagefile	relflagimage.list	relflagimagelist	
resourceonlyfile	resourceonly.list	resourceonlylist	
skipchunkfile	skipchunk.list	skipchunklist	
subjectschemefile	subjectscheme.list	subjectschemelist	
subtargetsfile	subtargets.list	subtargetslist	
tempdirToinputmapdir.relative.value			
uplevels			
user.input.dir			Absolute input directory path
user.input.file.listfile			Input file list file
user.input.file			Input file path, relative to input directory

Debug and filter (debug-filter)

The `debug-filter` stage processes all referenced DITA content, and creates copies in a temporary directory for use during the remainder of the build. Several modifications are made during this process.

As the files are copied, the following modifications are made:

- The files are filtered according to entries in any specified DITAVAL file.
- Debug information is inserted into each element (using the `xtrf` and `xtrc` attributes). These values allow messages later in the build to reliably indicate the original source of the error — for example, a message may trace back to the fifth `<ph>` element in a specific source document. Without these attributes, that count may no longer be available due to filtering and other processing.
- Adjust column names in tables to use a common naming scheme. This is done only to simplify later conref processing; for example, if a table row is pulled into another table, this ensures that a reference to "column 5 properties" will continue to work in the fifth column of the new table.

This step is implemented in Java.

Copy related files (copy-files)

The `copy-files` step copies related non-DITA resources to the output directory, such as HTML files referenced in a map or images referenced by DITAVAL files.

Conref push (conrefpush)

The `conrefpush` step resolves "conref push" references. The conref push feature was added in the DITA 1.2 specification, and the associated processing is available in DITA-OT version 1.5 and later. This step only processes documents that use conref push (or that are updated due to the push action). The step is implemented in Java.

Conref (conref)

The `conref` step resolves traditional `conref` attributes, processing only the documents that use the `conref` attribute. Each map or topic is processed with XSLT to resolve the attributes.

As part of the process, IDs within referenced content are changed as they are pulled into the new location. This is done in order to ensure that IDs within the original (referencing) topic remain unique.

If an element with an ID is pulled into a new context along with a cross reference that references the target, both the ID and the reference are updated so that they remain valid in the new location. For example, a referenced topic may include a section as in the following example.

```
<topic id="referenced_topic">
  <title>...</title>
  <body>
    <section id="sect"><title>Sample section</title>
      <p>Look at the next figure <xref href="#referenced_topic/fig">here</xref>.</p>
      <fig id="fig"><title>Sample</title>
        <p>This is a rather useless figure, but it
          illustrates a point.</p>
      </fig>
    </section>
  </body>
</topic>
```

If the section is referenced with a `conref` attribute, the ID on the `<fig>` element will be modified to ensure it remains unique inside the new topic. At the same time, the `<xref>` element will also be modified so that after the `conref` is resolved, it remains valid as a local reference. If the topic pulling in a new copy of the section has the id "new_topic", then the pulled copy of the section may look something like this in the intermediate document.

```
<section><title>Sample section</title>
  <p>Look at the next figure <xref href="#new_topic/d1e25">here</xref>.</p>
  <fig id="d1e25"><title>Sample</title>
    <p>This is a rather useless figure, but it
      illustrates a point.</p>
  </fig>
</section>
```

In this case, the ID of the figure has been changed to a generated value of "d1e25". At the same time, the `<xref>` element has been updated to use that new generated ID, so that the reference stays local in the updated topic.

Move metadata (move-meta-entries)

The `move-meta-entries` step pushes metadata back and forth between maps and topics. For example, index entries and copyrights in the map are pushed into affected topics, so that topics may be processed later in isolation while retaining all relevant metadata.

This step is implemented in Java.

Resolve keyref (keyref)

The `keyref` step examines all keys defined in the source material, and updates key references appropriately. Links that make use of keys are updated so that any `href` value is replaced by the appropriate target; key based text replacement is also evaluated. The `keyref` mechanism was defined as part of the DITA 1.2 standard, and is available in DITA-OT 1.5 and later.

This step is implemented in Java.

Resolve code references (coderef)

The `coderef` module resolves references made with the `<coderef>` element, which was added in DITA 1.2. This module is available in DITA-OT 1.5 and later.

The `<coderef>` element is used inside of `<codeblock>` to reference code stored externally in non-XML documents. During the pre-process step, this Java module pulls the referenced content into the `<codeblock>` element.

Resolve map references (mapref)

The `mapref` module resolves references from one map to another.

Maps may reference other maps using markup similar to the following:

```
<topicref href="other.ditamap" format="ditamap" />
```

The DITA 1.2 standard added a new element that allows this sort of reference without setting the `format` attribute:

```
<mapref href="other.ditamap" />
```

In either case, the element that references the other map is replaced by the topic references from the other map. Relationship tables are pulled into the referencing map as a child of the root element (`<map>` or a specialization of `<map>`).

This step is implemented in XSLT.

Pull content into maps (mappull)

The `mappull` step pulls content from referenced topics into maps, and cascades data within maps.

This step uses XSLT to make the following changes to the map:

- Pull titles from referenced DITA topics. This step replaces the navigation title specified on the `topicref`. If the `locktitle` attribute is set to "yes", the value in the map is not changed.
- The `<linktext>` element is set based on the title of the referenced topic, unless it is already specified locally.
- The `<shortdesc>` element is set based on the short description of the referenced topic, unless it is already specified locally.
- When a local DITA topic is referenced, the `type` attribute is set on the `topicref` based on the type of topic referenced. For example, a reference to a task topic will end up with `type="task"`.
- Inheritable attributes, such as `toc` or `print`, are made explicit on child `topicref` elements. This allows any future step to work with the attributes directly, without reevaluating the cascade behavior.

Chunk topics (chunk)

The `chunk` step is a Java module that breaks apart and assembles referenced DITA content based on the `chunk` attribute in maps.

The following values are recognized on the `chunk` attribute, based on definitions provided in the DITA specification. These values were initially defined in the DITA 1.1 specification, with significant clarifications in the DITA 1.2 specification.

- `select-topic`
- `select-document`
- `select-branch`
- `by-topic`
- `by-document`
- `to-content`
- `to-navigation`.

Map based linking (maplink and move-links)

These two steps work together to create links based on a map and move those links into referenced topics. The links are created based on hierarchy (parent/child), the `collection-type` attribute (sequential or family links), and relationship tables.

The `maplink` module first runs an XSLT program that evaluates the map, and places all generated links into a single file in the temporary processing directory. Once that file is created, the `move-links` module runs a Java program that pushes the generated links into the proper topics.

Pull content into topics (`topicpull`)

The `topicpull` module pulls content into `<xref>` and `<link>` elements (if needed).

For `<xref>` elements, if the `<xref>` does not contain link text, the target is examined and link text is pulled. For example, a reference to a topic will pull the title of the topic; a reference to a list item will pull the number of the item. If the `<xref>` element references a topic that has a short description, and the `<xref>` element does not already contain a child `<desc>` element, a `<desc>` element is created with the short description of the target.

The process is similar for `<link>` elements. If the `<link>` does not have a child `<linktext>` element, one is created with the appropriate link text. Similarly, if the `<link>` element does not have a child `<desc>` element, and the short description of the target can be determined, a `<desc>` is created with the short description of the target.

This step is implemented in XSLT.

Generating XHTML with navigation

The toolkit ships with several varieties of XHTML output, each of which follows roughly the same path through the processing pipeline. All XHTML builds begin with the same call to the preprocess routine, after which they generate XHTML files and then branch to create navigation files.

Once the preprocess runs, XHTML based builds each run a common series of Ant targets. Navigation may be created before or after this set of common routines.

- When the CSS parameter is passed to the build to add a CSS file, the `copy-css` target copies that CSS file from its source location to the proper relative location in the output directory.
- When a DITAVAL file is used, the `copy-revflag` target copies the default start and end revision flags into the output directory.
- Two targets names `dita.inner.topics.xhtml` and `dita.outer.topics.xhtml` are used to convert DITA topics into XHTML documents. At this point after the preprocess has completed, each DITA topic document in the temporary directory corresponds to one XHTML output document. The "inner" template is used to process documents that are in the map directory (or subdirectories of that directory). The "outer" template is used to process documents that are outside of the scope of the map, and may end up outside of the designated output directory. Parameters to the build control how documents processed by the "outer" target are handled.

Default XHTML output

The `dita.map.xhtml` target is called by default xhtml builds. This target generates a TOC file called `index.html`, which may be loaded into an independent frameset.

Eclipse help output (transform type "eclipsehelp")

Eclipse help is an XHTML based output format intended to create a plug-in for the Eclipse Help system. Once the normal XHTML process has run, the `dita.map.eclipse` target is used to create a set of several control files and navigation files for Eclipse.

Eclipse relies on several different files to control the plug-in. Some of these are generated by the build, while others may be created by hand. The Ant targets used to control this process are:

- `dita.map.eclipse.init` sets up various default properties for processing Eclipse output.
- `dita.map.eclipse.toc` creates the XML file that defines an Eclipse table of contents.
- `dita.map.eclipse.index` creates the sorted XML file that defines an Eclipse index.
- `dita.map.eclipse.plugin` creates the `plugin.xml` file that controls the behavior of an Eclipse plug-in.
- `dita.map.eclipse.plugin.properties` creates a Java properties file that sets properties for the plug-in (such as name and version information).
- `dita.map.eclipse.manifest.file` creates a `MANIFEST.MF` file with additional information used by Eclipse.
- `copy-plugin-files` checks for the presence of several control files in the source directory, and copies those found to the output directory.

- `dita.map.eclipse.fragment.language.init`, `dita.map.eclipse.fragment.language.country.init`, and `dita.map.eclipse.fragment.error` all work together to control Eclipse fragment files (used for versions of a plug-in created for a new language or locale).

Several of the targets listed above have matching templates for processing content that is located outside of the scope of the map directory (such as `dita.out.map.eclipse.toc`).

TocJS output path

The TocJS transform type was originally created as a plug-in distributed outside of the toolkit, but now ships bundled in the default packages. This XHTML based output type creates a JavaScript based frameset with TOC entries that expand and collapse.

A few Ant targets control most of the TocJS processing:

- `tocjsInit` is used to set up default properties. This template detects whether builds have already set a name for the JavaScript control file; if not, the default name `toctree.js` is used.
- `map2tocjs` calls `dita.map.tocjs`, which generates the contents frame for TocJS output.
- `tocjsDefaultOutput` was added to the process in version 1.5.4 of the DITA-OT. If scripts are missing some required information, such as a name for the default frameset, this template will copy default style and control files. It also ensures that the XHTML process runs (earlier versions of TocJS created only the JavaScript control file by default).

Compiled Help (CHM) output

The transform type "htmlhelp" is used to create HTML Help control files. If the build runs on a system that has the HTML Help compiler installed, the control files will be compiled into a CHM file.

Once the preprocess and XHTML process is complete, most of the HTML Help processing is handled by the `dita.map.htmlhelp` target. This target creates several files:

- The HHP file is the control file for the HTML Help project.
- The HHC file contains the HTML Help table of contents.
- The HHK file contains the HTML Help index. This file is sorted based on the language of the map.

The `dita.htmlhelp.convertlang` is a post-processor for the content to ensure that it can be processed correctly by the compiler, and that the appropriate codepages and languages are used.

Finally, `compile.HTML.Help` attempts to detect the HTML Help compiler; if found, it compiles the full project into a single CHM file.

Javahelp output

The "javahelp" transform type runs several additional Ant targets after the XHTML process is completed in order to create control files for JavaHelp output.

There are two primary targets in the Ant JavaHelp code.

- `dita.map.javahelp` creates all of the files needed to compile Javahelp, including a table of contents, sorted index, and help map file.
- `compile.Java.Help` searches for a Javahelp compiler on the system; if found, it will compile the help project.

PDF output pipeline

The PDF process (formerly known as PDF2) runs the preprocess routine, followed by a series of additional targets. These steps work together to create a merged set of content, convert that to XSL-FO, and then format the FO file to PDF.

The PDF process adds many new Ant targets. During a typical conversion from map to PDF, the following targets are most significant.

- `map2pdf2` creates a merged file by calling a common Java merge module. It then calls `publish.map.pdf` to do the remainder of the work.

- The `publish.map.pdf` target does some initialization, and then calls `transform.topic2pdf` to do the remainder of processing. That target runs all of the following steps.
 - `transform.topic2fo` is used to convert the merged file to an XSL-FO file. This process is composed of several Ant targets.
 - `transform.topic2fo.index` runs a Java process to set up index processing, based on the document language. This step generates the file `stage1.xml` in the temporary processing directory.
 - `transform.topic2fo.flagging` sets up preprocessing for flagging based on a DITAVAL file. This step generates the file `stage1a.xml` in the temporary processing directory.
 - `transform.topic2fo.main` does the bulk of the conversion from DITA to XSL-FO. It runs the XSLT based process that creates `stage2.fo` in the temporary processing directory.
 - `transform.topic2fo.i18n` does additional localization processing on the FO file; it runs a Java process that converts `stage2.fo` into `stage3.fo`, followed by an XSLT process that converts `stage3.fo` into `topic.fo`.
 - `transform.fo2pdf` converts the `topic.fo` file into PDF using the available FO processor (Antenna House, XEP, or Apache FOP).
 - `delete.fo2pdf.topic.fo` deletes the `topic.fo` file, unless otherwise specified by setting an Ant property or command line option.

ODT Transform type (Open Document Format)

The "odt" transform type creates a binary file using the OASIS standard Open Document Format.

The "odt" transform path begins with the preprocess, as with other builds. It then runs the Ant target `dita.odt.package.topic` (if the input file is a topic) or `dita.odt.package.map` (if the input file is a map). This description focuses on the map process, which is made up of the following targets.

- `dita.map.odt` creates the `content.xml` portion of the ODT output file. This is done by converting the map into a merged XML file using the Java `topicmerge` program. An XSLT process is then used to convert the merged file into `content.xml`.
- `dita.map.odt.stylesfile` is a target that reads the input map, and uses XSLT to create a `styles.xml` file in the temporary directory.
- `dita.out.odt.manifest.file` creates the `manifest.xml` portion of the ODT output file.
- Once the three previous targets run, the generated files are zipped up together with other required files to create the output ODT file.

Extending the DITA Open Toolkit

There are several methods that can be used to extend the toolkit; not all of them are recommended or supported. The best way to create most extensions is with a plug-in; extended documentation for creating plug-ins is provided in the next section.

- Creating a plug-in can be very simple to very complex, and is generally the best method for changing or extending the toolkit. Plug-ins can be used to accomplish almost any modification that is needed for toolkit processing, from minor style tweaks to extensive, complicated new output formats.
- The PDF process was initially developed independently of the toolkit, and created its own extension mechanism using customization directories. Many (but not quite all) of the capabilities available through PDF customization directories are now available through plug-ins.
- Using a single XSL file as an override by passing it in as a parameter. For example, when building XHTML content, the XSL parameter allows users to specify a single local XSL file (inside or outside of the toolkit) that is called in place of the default XHTML code. Typically, this code imports the default processing code, and overrides a couple of processing routines. This approach is best when the override is very minimal, or when the style varies from build to build. However, any extension made with this sort of override is also possible with a plug-in.

- Editing DITA-OT code directly may work in some cases, but is not advised. Modifying the code directly significantly increases the work and risk involved with future upgrades. It is also likely that such modifications will break plug-ins provided by others, limiting the function available to the toolkit.

Installing DITA-OT plug-ins

Plug-ins are generally distributed as zip files. There are two steps to installing a plug-in: unzipping and integrating.

It is possible to define a plug-in so that it may be installed anywhere, although most expect to be placed in either the `demo/` or `plugins/` directory inside of the DITA-OT. Most plug-ins do not require a specific install directory and can go in either of the default locations, but some may come with instructions for a particular install directory. The first step to installing a plug-in is to unzip to the desired location.

The remaining step is to integrate the new plug-in. This can be accomplished in a few ways:

1. From the toolkit directory, you can run the following command to integrate all installed plug-ins:

```
ant -f integrator.xml
```

The integration process has two modes, lax and strict. In the strict mode the integration process will immediately fail if it encounters errors in plug-in configurations or installation process. In the lax mode, the integration process will continue to finish regardless of errors; the lax mode does not imply error recovery and may leave the DITA-OT installation into a broken state. The default mode is lax due to backwards compatibility, to run the integration in strict mode:

```
ant -f integrator.xml strict
```

To get more information about the integration process, run Ant in verbose mode:

```
ant -f integrator.xml -verbose strict
```

2. Any build that uses the Java command line interface automatically runs the integrator before processing begins.
3. Ant based builds may import the `integrator.xml` file, and add `integrate` to the start of the dependency chain for the build.



Note: The integration process is considered part of the installation process and running it before each conversion will incur a performance penalty.

Creating DITA-OT plug-ins

The DITA Open Toolkit comes with a built in mechanism for adding in extensions through plug-ins. These plug-ins may do a wide variety of things, such as adding support for specialized DITA DTDs or Schemas, integrating processing overrides, or even providing entirely new output transforms. Plug-ins are the best way to extend the toolkit in a way that is consistent, easily sharable, and easy to preserve through toolkit upgrades.

A plug-in consists of a directory, typically stored directly within the `demo/` or `plugins/` directory inside of the DITA-OT. Every plug-in is controlled by a file named `plugin.xml`, located in the plug-in's root directory.

Benefits of extending the toolkit through plug-ins include:

- Plug-ins are easily sharable with other users, teams, or companies; typically, all that is needed is to unzip and run a single integration step. With many builds, even that integration step is automatic.
- Allows overrides or customizations to grow from simple to complex over time, with no increased complexity to the extension mechanism.
- Plug-ins can be moved from version to version with an upgraded toolkit simply by unzipping again, or by copying the directory from one install to another; there is no need to re-integrate code based on updates to the core processing.

- Plug-ins can build upon each other. If you like a plug-in provided by one user, simply install that plug-in, and then create your own that builds on that extension. The two plug-ins can then be distributed to your team as a unit, or you can even share your own extensions with the original provider.

Plug-in configuration file

The `plugin.xml` controls all aspects of a plug-in, making each extension visible to the rest of the toolkit. The file uses pre-defined extension points to locate changes, and integrates those changes into the core code.

The root element of the `plugin.xml` file is `<plugin>`, and must specify an `id` attribute. The `id` attribute is used to identify the plugin, as well as to identify whether pre-requisite plugins are available. The `id` attribute should follow the syntax rules:

```
id ::= token('.'token)*
token ::= ( [0..9] | [a..zA..Z] | '_' | '-' )+
```

The `<plugin>` element supports the following child elements:

- `<feature>` defines an *extension* to contribute to a defined *extension point*. The following attributes are supported:

Attribute	Description	Required
extension	extension point identifier	yes
value	comma separated string value of the extension	either value or file
file	file path value of the extension, relative to <code>plugin.xml</code>	either value or file
type	type of the value attribute	no

- `<require>` defines plug-in dependencies. The following attributes are supported:

Attribute	Description	Required
plugin	vertical bar separated list of plug-ins that are required	yes
importance	flag whether plug-in is required or optional	no

- `<template>` defines files that should be treated as *templates*. The following attributes are supported:

Attribute	Description	Required
file	file path to the template, relative to <code>plugin.xml</code>	yes

- `<meta>` defines metadata. The following attributes are supported:

Attribute	Description	Required
type	metadata name	yes
value	metadata value	yes

Any extension that is not recognized by the DITA-OT is ignored; all elements other than `<plugin>` are optional. Since version 1.5.3 multiple extension definitions within a plugin configuration file are combined; in older versions only the last extension definition is used.

Extending the XML Catalog

The XML Catalogs extension point is used to update the XML Catalogs used to resolve DTD or Schema document types, or to add URI mappings. This is required in order to support DITA specializations or new DITA document type shells.

To do this, first create a catalog with only your new values, using the OASIS Catalog format, and place that in your plug-in. Local file references in the catalog should be relative to the location of the catalog. The following extension points are available to work with catalogs.

dita.specialization.catalog.relative
dita.specialization.catalog

Adds the content of the catalog file defined in `file` attribute to main DITA-OT catalog file.



Remember: The `dita.specialization.catalog` extension is deprecated. Use `dita.specialization.catalog.relative` instead.

org.dita.pdf2.catalog.relative

Adds the content of the catalog file defined in `file` attribute to main PDF plug-in catalog file.

Example

This example assumes that "catalog-dita.xml" contains an OASIS catalog for any DTDs or Schemas inside this plug-in. The catalog entries inside of `catalog-dita.xml` are relative to the catalog itself; when the plug-in is integrated, they will be added to the core DITA-OT catalog (with the correct path).

```
<plugin id="com.example.catalog">
  <feature extension="dita.specialization.catalog.relative"
    file="catalog-dita.xml"/>
</plugin>
```

Adding new targets to the Ant build process

The Ant conductor extension point is used to make new targets available to the Ant processing pipeline. This may be done as part of creating a new transform, extending pre-processing, or simply to provide Ant targets for the use of other plug-ins.

dita.conductor.target.relative
dita.conductor.target

Add Ant import to main Ant build file.



Remember: The `dita.conductor.target` extension is deprecated. Use `dita.conductor.target.relative` instead.

Example

To extend And processing, first place your extensions in an Ant project file within your plug-in, such as `myAntStuff.xml`. Next, create a small wrapper file `myAntStuffWrapper.xml` in the same directory:

```
<dummy> <import file="myAntStuff.xml"/> </dummy>
```


Then create the following feature:

```
<plugin id="com.example.ant">
  <feature extension="dita.conductor.target.relative"
    file="myAntStuffWrapper.xml"/>
</plugin>
```

When the plug-in is integrated, the imports from `myAntStuffWrapper.xml` will be copied into `build.xml` (using the correct path). This makes targets in `myAntStuff.xml` available to any other processing.

Adding Ant targets to the pre-process pipeline

Every step in the pre-process pipeline defines an extension point before and after the step, to allow plug-ins to integrate additional processing. This allows a plug-in to insert a new step before any pre-processing step, as well as before or after the entire preprocess pipeline.

The group of preprocessing steps defines extension points before and after the full preprocessing chain.

<code>depend.preprocess.pre</code>	Preprocessing pre-target; extending this target runs your Ant target before the full preprocess routine begins.
<code>depend.preprocess.post</code>	Preprocessing post-target; extending this target runs your Ant target after the full preprocess routine completes.

In addition, there are extension points to execute an Ant target before individual preprocessing steps.

<code>depend.preprocess.clean-temp.pre</code>	Clean temp pre-target
<code>depend.preprocess.gen-list.pre</code>	Generate list pre-target
<code>depend.preprocess.debug-filter.pre</code>	Debug and filter pre-target
<code>depend.preprocess.conrefpush.pre</code>	Content reference push pre-target
<code>depend.preprocess.move-meta-entries.pre</code>	Move meta entries pre-target
<code>depend.preprocess.conref.pre</code>	Content reference pre-target
<code>depend.preprocess.coderef.pre</code>	Code reference pre-target
<code>depend.preprocess.mapref.pre</code>	Map reference pre-target
<code>depend.preprocess.keyref.pre</code>	Resolve key reference pre-target
<code>depend.preprocess.mappull.pre</code>	Map pull pre-target
<code>depend.preprocess.chunk.pre</code>	Chunking pre-target
<code>depend.preprocess.maplink.pre</code>	Map link pre-target
<code>depend.preprocess.move-links.pre</code>	Move links pre-target
<code>depend.preprocess.topicpull.pre</code>	Topic pull pre-target
<code>depend.preprocess.copy-files.pre</code>	Copy files pre-target
<code>depend.preprocess.copy-image.pre</code>	Copy images pre-target
<code>depend.preprocess.copy-html.pre</code>	Copy HTML pre-target
<code>depend.preprocess.copy-flag.pre</code>	Copy flag pre-target
<code>depend.preprocess.copy-subsidiary.pre</code>	Copy subsidiary pre-target

depend.preprocess.copy-generated-files.pre

Copy generated files pre-target

Example

The following feature adds "myAntTargetBeforeChunk" Ant target to be executed before the chunk step in preprocessing. It assumes that an Ant file defining that target has already been integrated.

```
<plugin id="com.example.extendchunk">
  <feature extension="depend.preprocess.chunk.pre"
    value="myAntTargetBeforeChunk" />
</plugin>
```

When integrated, the Ant target "myAntTargetBeforeChunk" will be added to the Ant dependency list so that it always runs immediately before the Chunk step.

Integrating a new transform type

Plug-ins may integrate an entire new transform type. The new transform type can be very simple, such as an XHTML build that creates an additional control file; it can also be very complex, adding any number of new processing steps.

The transtype extension point is used to define a new "transtype", or transform type, which makes use of targets in your Ant extensions. When a transform type is defined, the build expects Ant code to be integrated to define the transform process. The Ant code must define a target based on the name of the transform type; if the transform type is "mystuff", the Ant code must define a target named dita2mystuff.

dita.conductor.transtype.check

Add new value to list of valid transformation type names.

dita.transtype.print

Declare transtype as a print type.

Example

The following feature defines a transform type of "newtext" and declares it as a print type; using this transform type will cause the build to look for a target dita2newtext, defined in a related Ant extension from the third feature:

```
<plugin id="com.example.newtext">
  <feature extension="dita.conductor.transtype.check"
    value="newtext" />
  <feature extension="dita.transtype.print" value="newtext" />
  <feature extension="dita.conductor.target.relative"
    file="antWrapper.xml" />
</plugin>
```

Override styles with XSLT

The XSLT import extension points are used to override various steps of XSLT processing. For this, the extension attribute indicates the step that the override applies to; the value attribute is a relative path to the override within the current plugin; if specified, the (optional) type attribute should be set to "file". The plugin installer will add an XSL import statement to the default code so that your override becomes a part of the normal build.

The following XSLT steps are available to override in the core toolkit:

dita.xsl.xhtml

Overrides default (X)HTML output (including HTML Help and Eclipse Help). The referenced file is integrated directly into the XSLT step that generates XHTML.

dita.xsl.xslfo	Overrides default PDF output (formerly known as PDF2). The referenced file is integrated directly into the XSLT step that generates XSL-FO for PDF.
dita.xsl.docbook	Overrides default DocBook output.
dita.xsl.rtf	Overrides default RTF output.
dita.xsl.eclipse.plugin	Overrides the step that generates plugin.xml for Eclipse.
dita.xsl.conref	Overrides the preprocess step that resolves conref.
dita.xsl.topicpull	Overrides the preprocess step "topicpull" (the step that pulls text into <xref> elements, among other things).
dita.xsl.mapref	Overrides the preprocess step "mapref" (the step that resolves references to other maps).
dita.xsl.mappull	Overrides the preprocess step "mappull" (the step that updates navtitles in maps and causes attributes to cascade).
dita.xsl.maplink	Overrides the preprocess step "maplink" (the step that generates map-based links).
dita.xsl.fo	Override the (now deprecated) original PDF output, which is still available with the transform type "legacypdf".

Example

The following two files represent a complete, simple style plug-in. The `plugin.xml` file declares an XSLT file that extends XHTML processing; the XSLT file overrides default header processing to provide a (theoretical) banner.

```

plugin.xml:
<?xml version="1.0" encoding="UTF-8"?>
<plugin id="com.example.brandheader">
  <feature extension="dita.xsl.xhtml" file="xsl/header.xsl"/>
</plugin>

xsl/header.xsl:
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/
Transform">
  <xsl:template name="gen-user-header">
    <div></div>
  </xsl:template>
</xsl:stylesheet>

```

Adding new generated text

Generated text is the term for strings that are automatically added by the build, such as "Note" before the contents of a <note> element.

The generated text extension point is used to add new strings to the default set of Generated Text.

dita.xsl.strings	Add new strings to generated text file.
-------------------------	---

Example

First copy the file `xsl/common/strings.xml` to your plug-in, and edit it to contain the languages that you are providing translations for ("en-us" must be present). The new strings file will look something like this:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Provide strings for my plug-in; this plug-in supports
      English, Icelandic, and Russian. -->
<langlist>
  <lang xml:lang="en"      filename="mystring-en-us.xml" />
  <lang xml:lang="en-us"   filename="mystring-en-us.xml" />
  <lang xml:lang="is"      filename="mystring-is-is.xml" />
  <lang xml:lang="is-is"   filename="mystring-is-is.xml" />
  <lang xml:lang="ru"      filename="mystring-ru-ru.xml" />
  <lang xml:lang="ru-ru"   filename="mystring-ru-ru.xml" />
</langlist>
```

Next, copy the file `xsl/common/strings-en-us.xml` to your plug-in, and replace the content with your own strings (be sure to give them unique name attributes). Do the same for each language that you are providing a translation for. For example, the file `mystring-en-us.xml` might contain:

```
<?xml version="1.0" encoding="utf-8"?>
<strings xml:lang="en-us">
  <str name="String1">English generated text</str>
  <str name="Another String">Another String in English</str>
</strings>
```

Use the following extension code to include your strings in the set of generated text:

```
<plugin id="com.example.strings">
  <feature extension="dita.xsl.strings" file="xsl/my-new-
strings.xml" />
</plugin>
```

The string is now available to the "getString" template used in many DITA-OT XSLT files. For example, if processing in a context where the `xml:lang` value is "en-us", the following call would return "Another String in English":

```
<xsl:call-template name="getString">
  <xsl:with-param name="stringName" select="'Another String'"/>
</xsl:call-template>
```

Passing parameters to existing XSLT steps

Plug-ins can define new parameters to be passed from the Ant build into existing XSLT pipeline stages, usually to have those parameters available as global `<xsl:param>` values within XSLT overrides.

To create new parameters, create a file `insertParameters.xml` which contains one or more Ant `<param>` elements. It also needs a `<dummy>` wrapper element around the parameters. For example, the following parameter will be passed in to the XSLT file with a value of `${antProperty}`, but only if that parameter is defined:

```
<dummy>
  <!-- Any Ant code allowed in xslt task is possible. Common example: -->
  <param name="paramNameinXSLT" expression="${antProperty}"
    if="antProperty"/>
</dummy>
```

Pass the value using the following extensions:

dita.conductor.html.param	Pass parameters to HTML and HTML Help XSLT
dita.conductor.xhtml.param	Pass parameters to XHTML and Eclipse Help XSLT
dita.preprocess.conref.param	Pass parameters to conref XSLT
dita.preprocess.mapref.param	Pass parameters to mapref XSLT
dita.preprocess.mappull.param	Pass parameters to mappull XSLT
dita.preprocess.maplink.param	Pass parameters to maplink XSLT
dita.preprocess.topicpull.param	Pass parameters to topicpull XSLT

Example

The following plug-in will pass the parameters defined inside of `insertParameter.xml` as input to the XHTML process. Generally, an additional XSLT override will make use of the parameter to do something new with the generated content.

```
<plugin id="com.example.newparam">
  <feature extension="dita.conductor.xhtml.param"
    file="insertParameters.xml"/>
</plugin>
```

Adding Java libraries to the classpath

If your Ant or XSLT extensions require additional Java libraries in the classpath, you can add them to the global Ant classpath with the following feature.

dita.conductor.lib.import	Add Java libraries to DITA-OT classpath.
----------------------------------	--

Example

The following plug-in adds the compiled Java code from `myJavaLibrary.jar` into the global Ant classpath. XSLT or Ant code can then make use of the added code.

```
<plugin id="com.example.addjar">
  <feature extension="dita.conductor.lib.import"
    file="myJavaLibrary.jar"/>
</plugin>
```

Adding diagnostic messages

Plug-in specific warning and error messages can be added to the set of messages supplied by the DITA-OT. These messages can then be used by any XSLT override.

dita.xsl.messages	Add new messages to diagnostic message file.
--------------------------	--

Example

To add your own messages, create the new messages in an XML file such as `myMessages.xml`:

```
<dummy>
  <!-- See resource/messages.xml for the details. -->
  <message id="DOTXmy-msg-numW" type="WARN">
    <reason>Message text</reason>
    <response>How to resolve</response>
```

```
</message>
</dummy>
```

There are three components to the message ID:

1. The prefix DOTX is used by all DITA-OT XSLT transforms, and must be part of the ID.
2. This is followed by the message number ("my-msg-num" in the sample above).
3. Finally, a letter corresponds to the severity. This should be one of:
 - I = Informational, used with type="INFO"
 - W = Warning, used with type="WARN"
 - E = Error, used with type="ERROR"
 - F = Fatal, used with type="FATAL"

Once the message file is defined, it is incorporated with this extension:

```
<plugin id="com.example.newmsg">
  <feature extension="dita.xsl.messages" file="myMessages.xml" />
</plugin>
```

XSLT modules can then generate the message using the following call:

```
<xsl:call-template name="output-message">
  <xsl:with-param name="msgnum">my-msg-num</xsl:with-param>
  <xsl:with-param name="msgsev">W</xsl:with-param>
</xsl:call-template>
```

Managing plug-in dependencies

The `<require>` element in a `plugin.xml` file is used to create a dependency on another plug-in. The `<require>` element requires the `plugin` attribute in order to reference the dependency.

If the current plug-in requires a plug-in with `id="plugin-id"` before it can be installed, it would include the following:

```
<require plugin="plugin-id">
```

Prerequisite plug-ins are integrated before the current plug-in is integrated. This does the right thing with respect to XSLT overrides. If your plug-in is a specialization of a specialization, it should `<require>` its base plug-ins, in order from general to specific.

If a prerequisite plug-in is missing, a warning will be printed during integration. To suppress this, but keep the integration order if both plug-ins are present, add `importance="optional"` to the `<require>` element.

If your plug-in can depend on any one of several optional plug-ins, separate the plug-in ids with a vertical bar. This is most useful when combined with `importance="optional"`:

Example

The following plug-in will only be installed if the plug-in with `id="com.example.primary"` is available. If that one is not available, a warning will be generated during the integration process.

```
<plugin id="com.example.builds-on-primary">
  <!-- ...extensions here -->
  <require plugin="com.example.primary" />
</plugin>
```

The following plug-in will only be installed if either the plug-in with id="pluginA" or the plug-in with id="pluginB" are available. If neither of those are installed, the current plug-in will be ignored.

```
<plugin id="pluginC">
  <!-- ...extensions here -->
  <require plugin="pluginA|pluginB" importance="optional"/>
</plugin>
```

Version and support information

The following extension points are used by convention to define version and support info within a plugin.

- package.support.name
- package.support.email
- package.version



Note:

The toolkit does not currently do anything with these values, but may do so in the future.

The package.version value should follow the syntax rules:

```
version    ::= major ( '.' minor ( '.' micro ( '.' qualifier )? )? )?
major      ::= number
minor      ::= number
micro      ::= number
qualifier  ::= ( [0..9] | [a..zA..Z] | '_' | '-' )+
```

The default value is 0.0.0.

Example

```
<plugin id="com.example.WithSupportInfo">
  <feature extension="package.support.name" value="Joe the
  Author"/>
  <feature extension="package.support.email"
  value="joe@example.com"/>
  <feature extension="package.version" value="1.2.3"/>
</plugin>
```

Creating a new plug-in extension point

If your plug-in needs to define its own extension point in an XML file, add the string "_template" to the filename before the file suffix. During integration, this file will be processed like the built-in DITA-OT templates.

Template files are used to integrate most DITA-OT extensions. For example, the file dita2xhtml_template.xsl contains all of the default rules for converting DITA topics to XHTML, along with an integration point for plug-in extensions. When the integrator runs, the file dita2xhtml.xsl is recreated, and the integration point is replaced with references to all appropriate plug-ins.

To mark a new file as a template file, use the <template> element.

Example

The following plug-in defines myTemplateFile_template.xsl as a new template for extensions. When the integrator runs, this will be used to recreate myTemplateFile.xsl, replacing any anchor points with the appropriate XSLT imports.

```
<plugin id="com.example.new-extensions">
```

```
<template file="myTemplateFile_template.xml" />
</plugin>
```

Example plugin.xml file

The following is a sample of a `plugin.xml` file. This file adds support for a new set of specialized DTDs, and includes an override for the XHTML output processor.

This `plugin.xml` file would go into a directory such as `DITA-OT\plugins\music\` and referenced supporting files would also exist in that directory. A more extensive sample using these values is available in the actual music plug-in, available at the [DITA-OT download page](#) at SourceForge

```
<plugin id="org.metadita.specialization.music">
  <feature extension="dita.specialization.catalog.relative"
    file="catalog-dita.xml">
  <feature extension="dita.xml.xhtml" file="xml/
music2xhtml.xml"/>
</plugin>
```

Implementation dependent features

Chunking

Supported chunking methods:

- select-topic
- select-document
- select-branch
- by-topic
- by-document
- to-content
- to-navigation.

When no chunk attribute values are given, no chunking is performed.



Note: In effect, for HTML based transformation types this is equivalent to select-document and by-document defaults.

Error recovery:

- When two tokens from the same category are used, no error or warning is thrown.
- When an unrecognized chunking method is used, no error or warning is thrown.

Filtering

Error recovery:

- When there are multiple revprop elements with the same val attribute, no error or warning is thrown
- When multiple prop elements define a duplicate attribute and value combination, attribute default, or fall-back behaviour, DOTJ007E error is thrown.

Debug attributes

The debug attributes are populated as follows:

xtrf	absolute system path of the source document
xtrc	element counter that uses the format
<code>element-name ":" integer-counter</code>	

Image scaling

If both height and width attributes are given, image is scaled nonuniformly.

If scale attribute is not an unsigned integer, no error or warning is thrown during preprocessing.

Extended functionality

Code reference processing

DITA-OT supports defining the code reference target file encoding using the `format` attribute. The supported format is:

```
format ( ";" space* "charset=" charset )?
```

If charset is not defined system default charset will be used. If charset is not recognized or supported, DOTJ052E error is thrown and system default charset is used as a fall-back.

```
<coderef href="unicode.txt" format="txt; charset=UTF-8"/>
```

Topic merge

The topic merge feature improves the build speed of DITA files and reduces the possibility of meeting the out of memory exception in the build process. As illustrated in the following figure, when you run the build in previous releases of DITA Open Toolkit, the build speed is slow and you are likely to get out of memory exception.

```

C:\WINDOWS\system32\cmd.exe

he file extension name to 'dita' or 'xml'.
[xs1t] [DOTX006E][ERROR]: Unknown file extension in href: '%1'.
ink to a non-DITA resource, set the format attribute to match the
example, 'txt', 'pdf', or 'html'). If it's a link to a DITA resour
xtension must be 'dita' or 'xml'. Set the format attribute and spe
t of the file if href link doesn't point to dita topic file. Other
he file extension name to 'dita' or 'xml'.
[move] Moving 230 files to C:\ditaot\temp
Build XSL-FO output from ditamap...
Build PDF from FO using FOP...
Log file 'ditaref-book_pdf.log' was generated successfully in dire
ot\out'.
Processing ended.

BUILD FAILED
C:\ditaot\build.xml:55: The following error occurred while executi
C:\ditaot\build_dita2pdf.xml:36: The following error occurred whil
is line:
C:\ditaot\build_dita2pdf.xml:80: The following error occurred whil
is line:
java.lang.OutOfMemoryError

Total time: 3 minutes 20 seconds

C:\ditaot>

```

With this enhanced topic merge feature, you will be less likely to meet the out of memory exception error when you build output through DITA files. The intermediate merged file will keep the structure information in the DITA map, and the structured toc will be reflected in the output.

To know more about this topic feature, you can write a script file first. DITA OT 1.3 offers a module, `TopicMerge`, that helps you implement this feature. You can use this module to generate the merged files. A sample usage of this module is as follows.

sample.xml:

```

<project name="sample">
  <property name="dita.dir" value="${basedir}"/>
  <import file="${dita.dir}${file.separator}build.xml"/>

  <target name="premerge">
    <antcall target="preprocess">
      <param name="args.input" value="{input}"/>
      <param name="output.dir" value="${dita.dir}${file.separator}output"/>
    </antcall>
  </target>
  <target name="merge" description="Merge topics" depends="premerge">

```

```

    <basename property="temp.base" file="${input}" suffix=".ditamap" />
    <property name="temp.input"
value="${basedir}${file.separator}${dita.temp.dir}${file.separator}${temp.base}" /
>
    <dirname property="temp.dir" file="${temp.input}" />
    <pipeline message="topicmerge" module="TopicMerge"
      inputmap="${temp.dir}${file.separator}${temp.base}.ditamap"
      extparam="output=${dita.dir}${file.separator}output
${file.separator}${temp.base}_merged.xml;
      style=${dita.dir}${file.separator}xsl${file.separator}pretty.xsl" />
  </target>
</project>

```

Then, you need to type `ant -f sample.xml merge -Dinput="C:\DITA-OT1.5.4\test.ditamap"` in the command window.



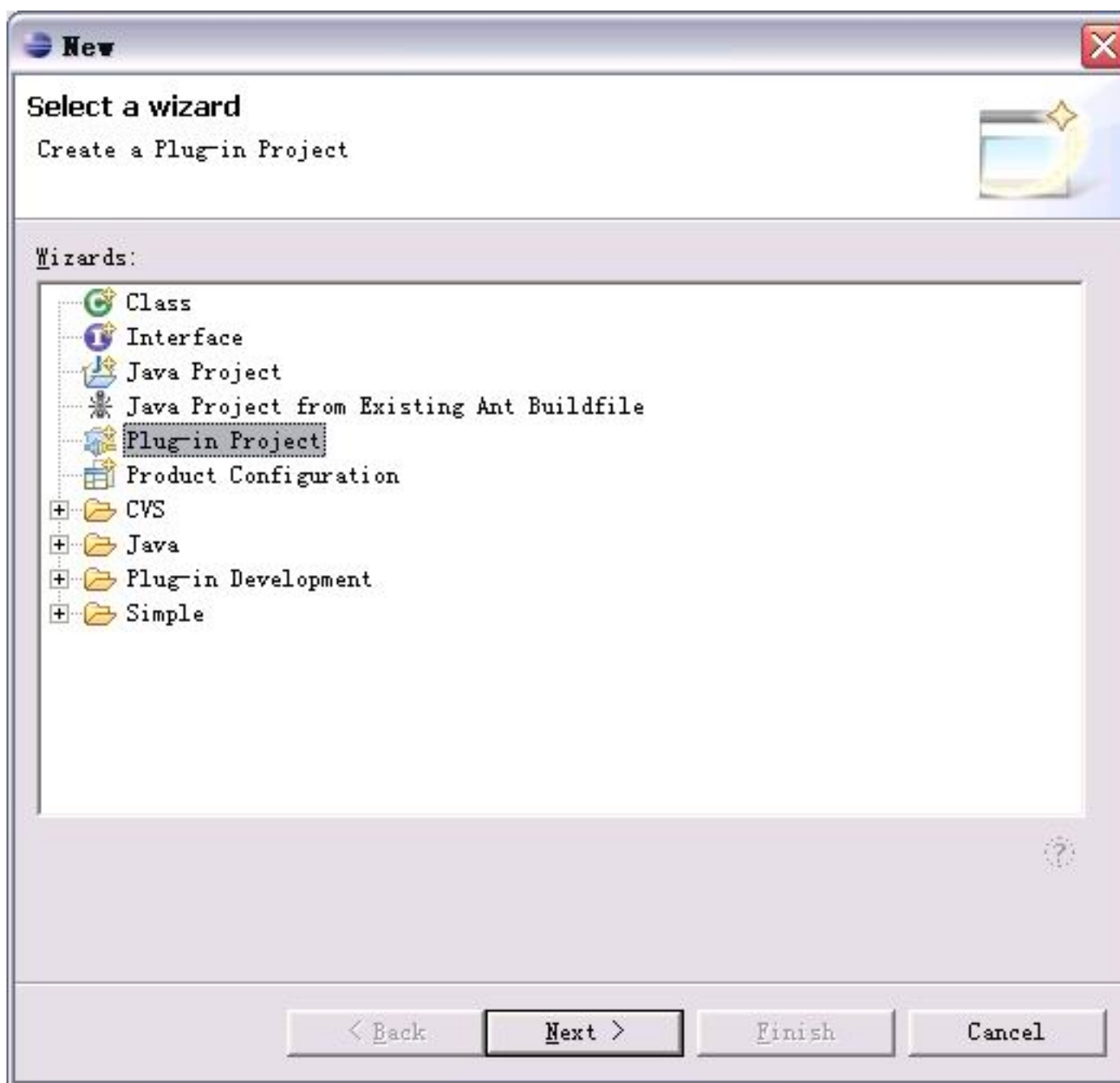
Note: The path for `-Dinput` must be an absolute path

Creating Eclipse help from within Eclipse

This topic explains how to create an Eclipse help documentation plug-in from within the Eclipse platform. This process allows you to set up repeatable builds directly within Eclipse, which may already be familiar to many developers working within Eclipse. The topic was originally written based on Eclipse 3.3, but much of the information still applies.


You can use a template to develop documentation plug-in with DITA in Eclipse PDE and use DITA-OT 1.5.4 to build and pack the final plug-in. When you want to develop a documentation plug-in with DITA in Eclipse, you cannot use the previous releases of DITA-OT in Eclipse to transform DITA to HTML. Though previous releases of DITA-OT support the feature to transform DITA files to Eclipse documentation plug-in, they are not integrated with Eclipse. With DITA-OT 1.5.4 integrated with WPT, you can develop document plug-ins with DITA in Eclipse PDE and build and pack the final plug-in by taking the following steps.

1. Create a new PDE project in Eclipse, and apply the DITA template to the project by following the wizard.




2. Set the source directory, the main ditamap file, the output directory (default value is root directory of project), css storage directory (used to contain `common.css`, `commonltr.css`, and `commonrtl.css`), user customized .css file name, and conditional processing ditaval file in the wizard. **Use root as output directory** is selected as the default.

You can also clear **Use root as output directory** and specify another output directory.



Create a DITA plug-in project

This wizard will help you to initiate the dita project



DITA OT Directory

Source Directory

Main ditamap File

☒ Use root as output directory

Output Directory (Default: root)

☐ Use customized css file

CSS Directory (Place which contains CSS files in the output directory)

CSS File: (Created in CSS Directory as User Customization to Default CSS Files)

☐ Conditional Processing

Condition File

< Back Next > Finish Cancel

DITA

Create a DITA plug-in project
This wizard will help you to initiate the dita project

DITA OT Directory:

Source Directory:

Main ditamap File:

☐ Use root as output directory

Output Directory (Default: root):

☐ Use customized css file

CSS Directory:
(Place which contains CSS files in the output directory)

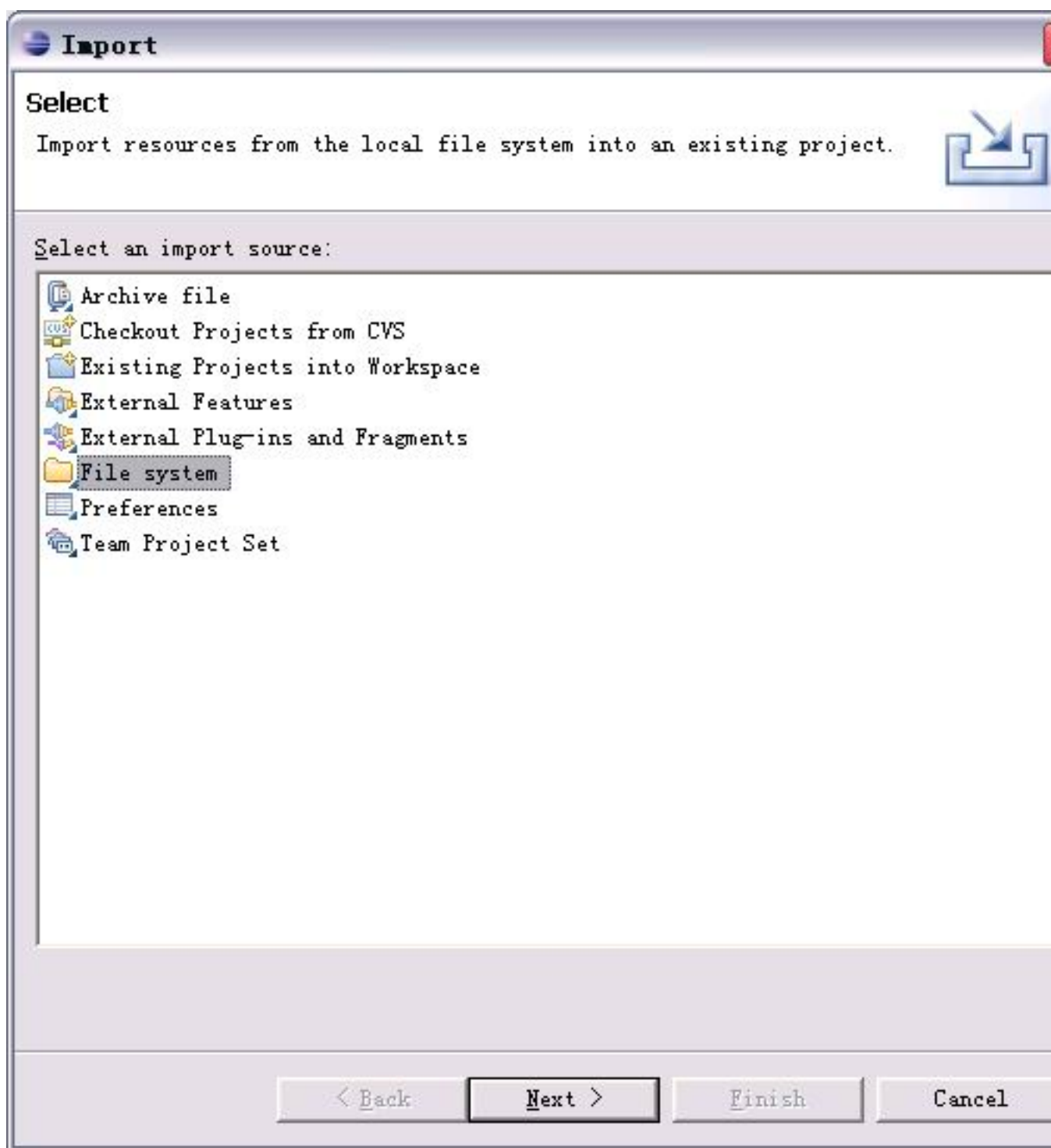
CSS File: (Created in CSS Directory as User Customization to Default CSS Files)

☐ Conditional Processing

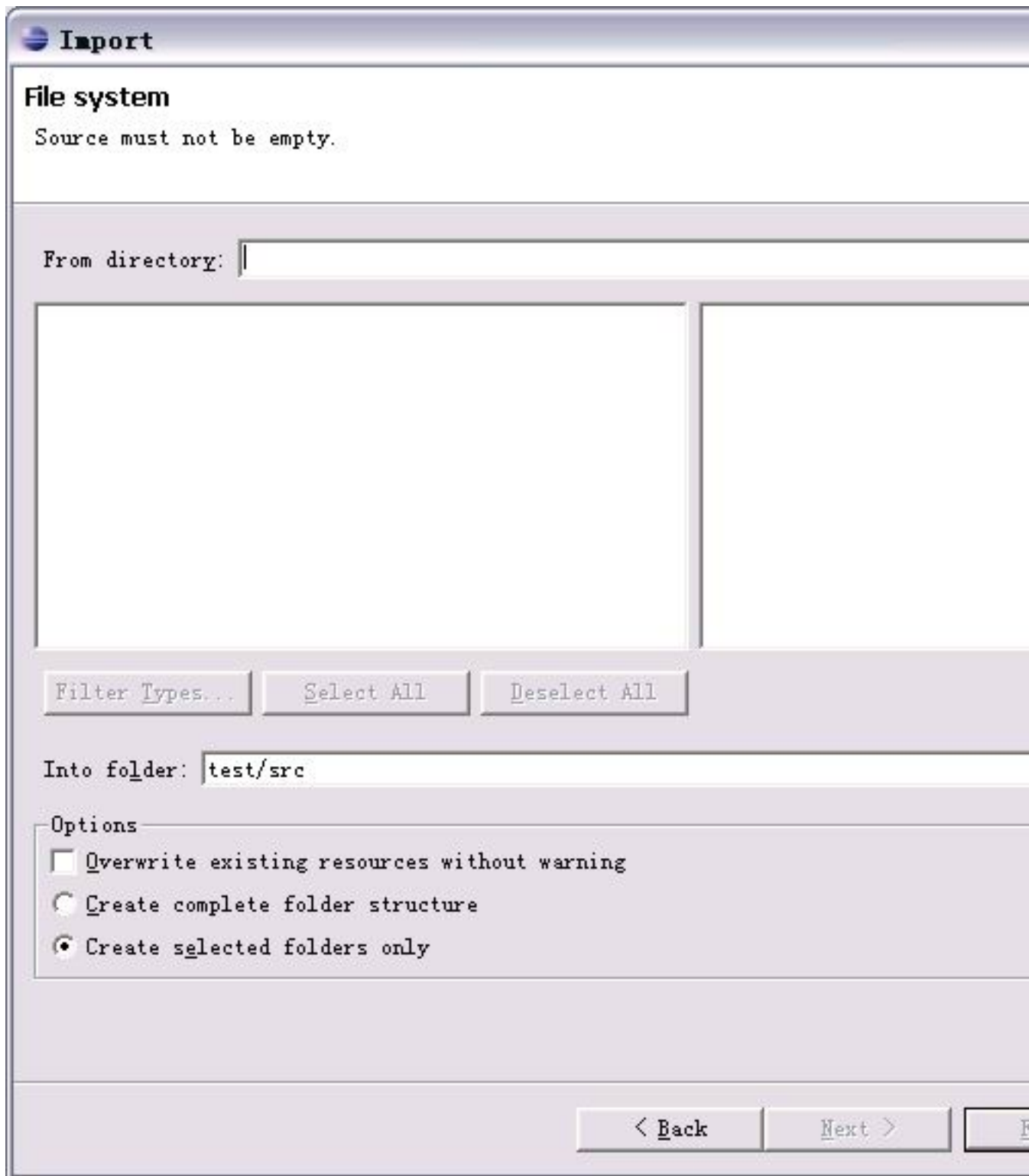
Condition File:

< Back Next > Finish Cancel

3. Create DITA files in the source directory and a ditamap to include the topic files that you created.
4. Optional: Import the DITA files into the `src` directory of the DITA plug-in project you just created.
 - a) Right-click a directory that you want to put the imported files and select **Import > File system**.



b) Select the directory under which you put the DITA files.



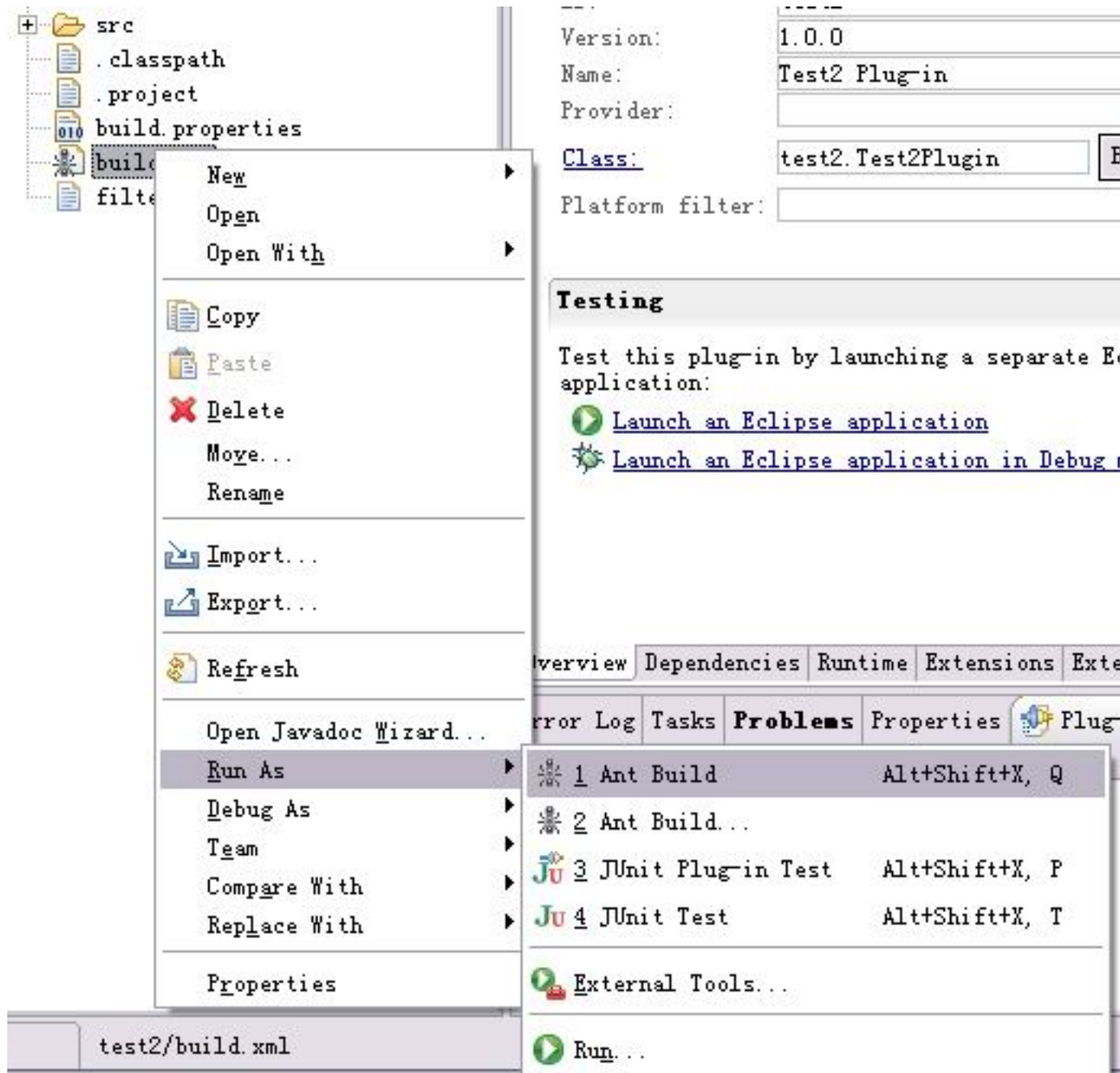
- c) Click Finish after you selected the DITA files under the specified directory. The DITA files are then imported to your DITA project.

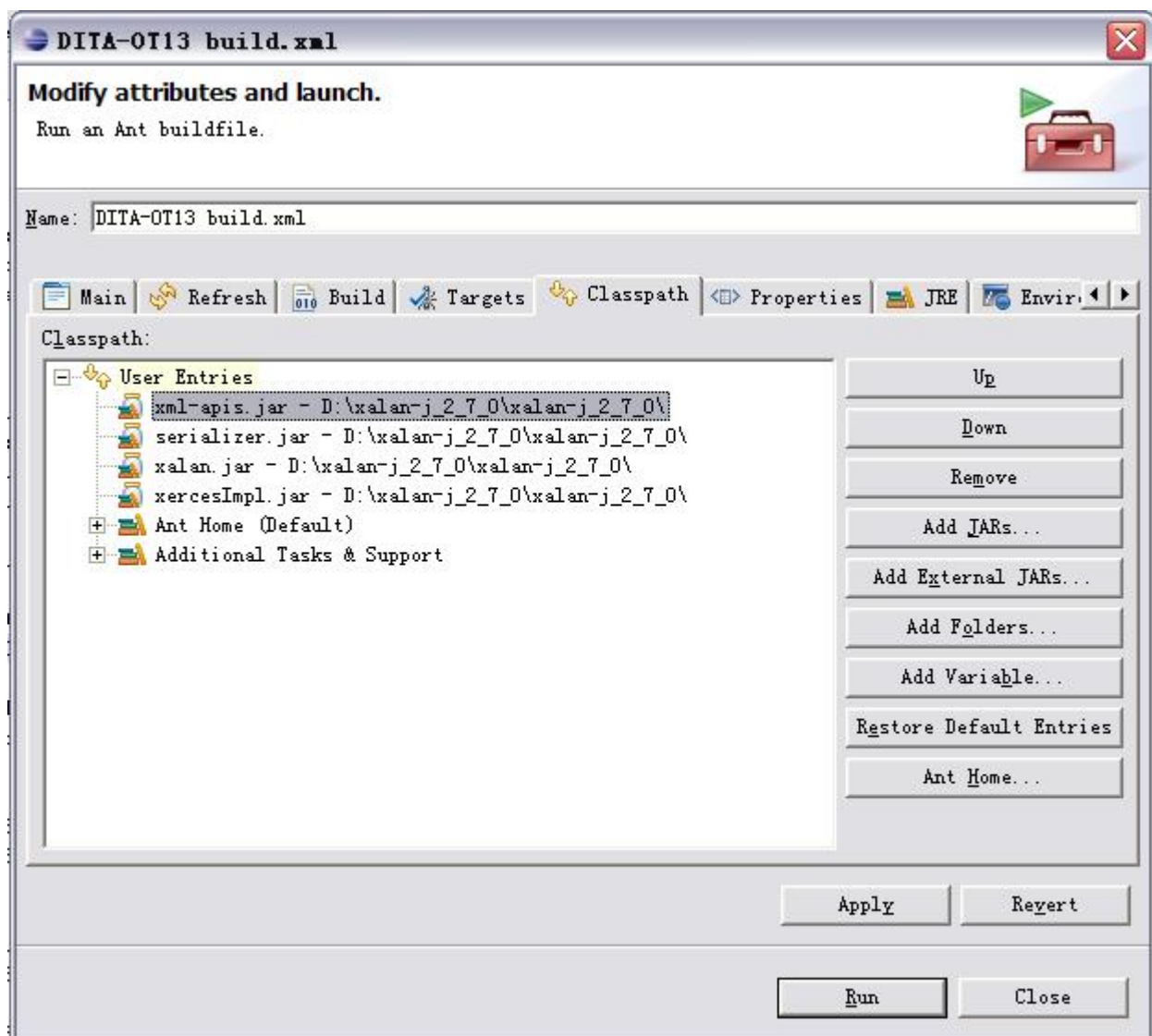
5. Right click `build.xml`, select **Run As > Ant Build**.



Note: If you're using Oracle JDK, please download and use the latest Xalan. The Xalan shipped with Oracle JDK has some issue that will cause the build failure. You can use the latest Xalan by selecting **ANT Build ...** and include the all of Xalan's jar files in Classpath.

After the transformation, the output is in the output directory set in `build.xml`. Refresh the project after the build is successful.





6. Edit the plug-in description of the property file `MANIFEST.MF` in the plug-in editor after you run the Ant build successfully.
 - a) Click `MANIFEST.MF` to go to the **Overview** page.

Overview

General Information

This section describes general information about this plug-in:

ID:	test2	
Version:	1.0.0	
Name:	Test2 Plug-in	
Provider:		
Class:	test2.Test2Plugin	Browse...
Platform filter:		





Testing

Test this plug-in by launching a separate Eclipse application:

-  [Launch an Eclipse application](#)
-  [Launch an Eclipse application in Debug mode](#)

Plug-in Content

The content of the plug-in:

-  [Dependencies](#): list of plug-in's classpath entries.
-  [Runtime](#): lists the plug-in's runtime.
-  [Extensions](#): declares extensions to the platform.
-  [Extension Points](#): lists extension points the plug-in adds to the platform.

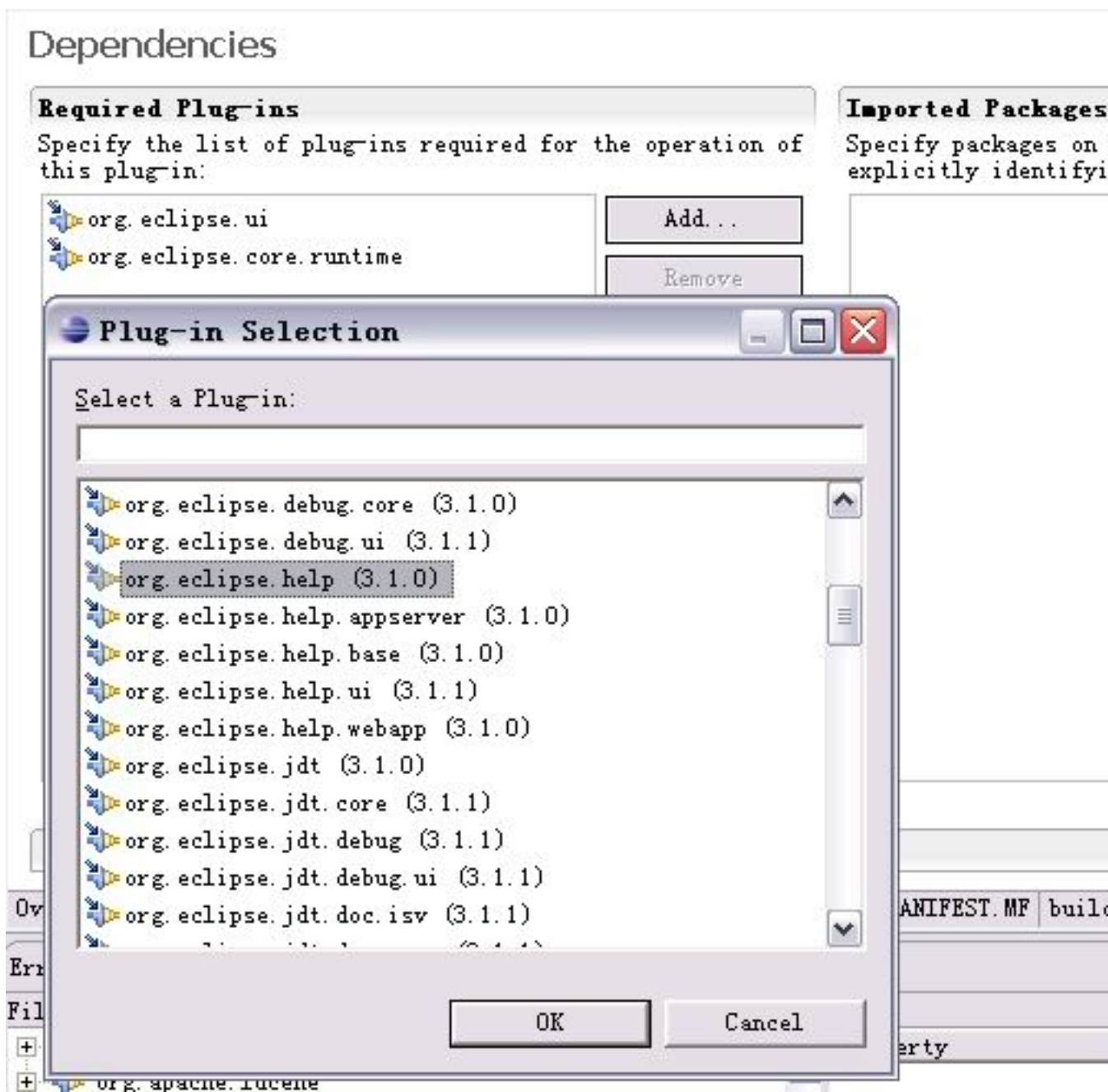
Exporting

To package and export the plug-in:

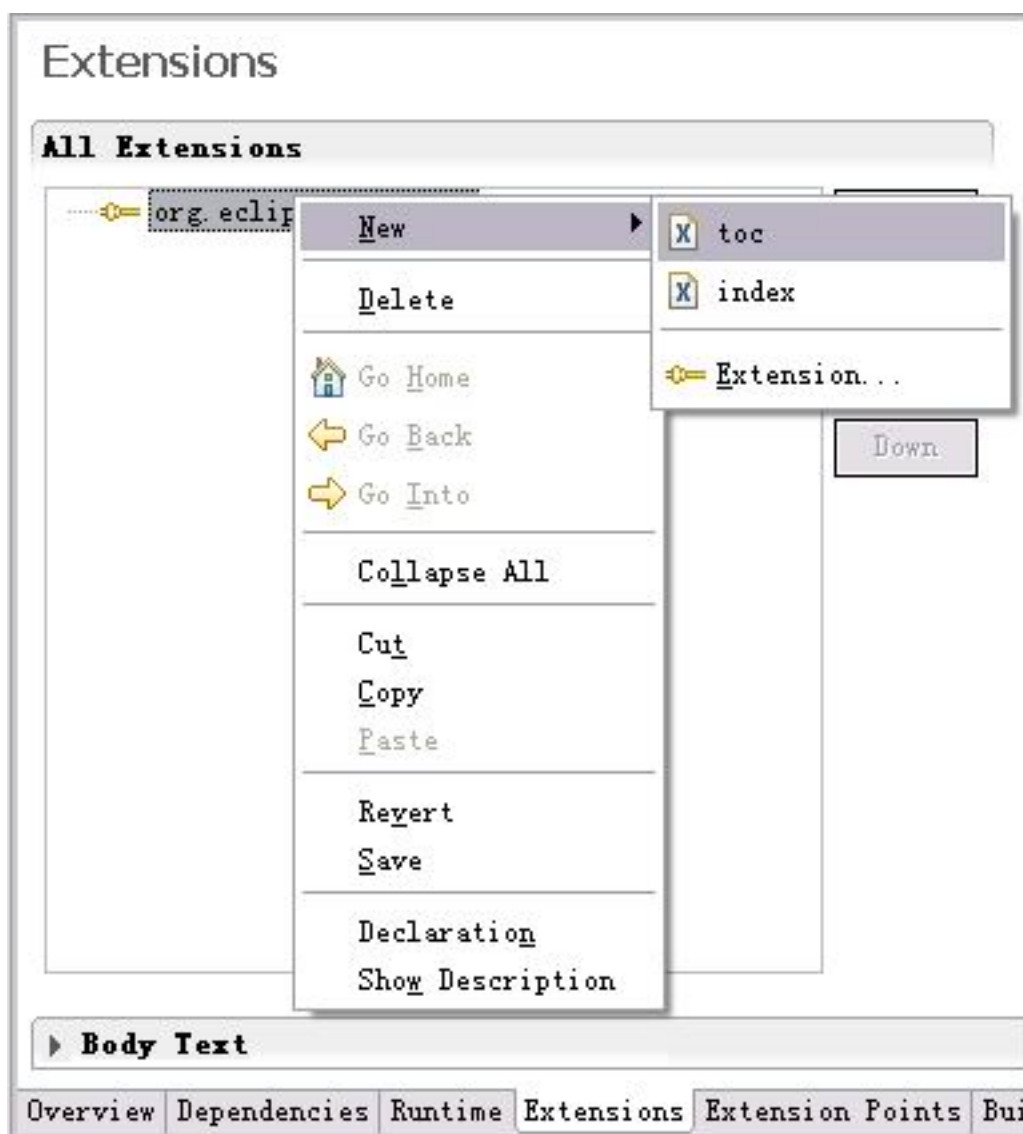
1. Specify what needs to be exported on the [Build](#) tab.
2. Export the plug-in using the [Export Wizard](#).

Overview | Dependencies | Runtime | Extensions | Extension Points | Build | MANIFEST.MF | build.properties

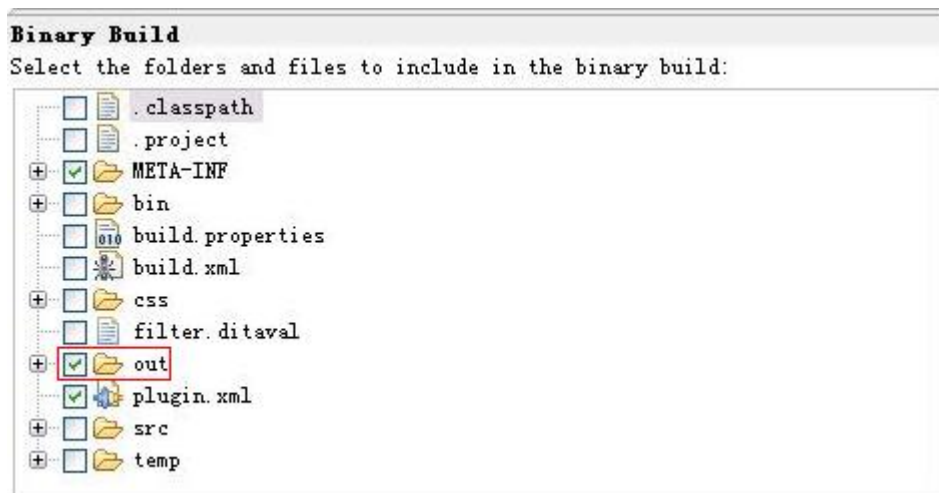
- b) Edit Dependencies to include org.eclipse.help.



- c) Edit Extensions to add `org.eclipse.help.toc`; right click the added `org.eclipse.help.toc`, and select **New > toc**.



- d) Edit the Build Configuration to include the out directory or the directory you specified in 2 on page 84.

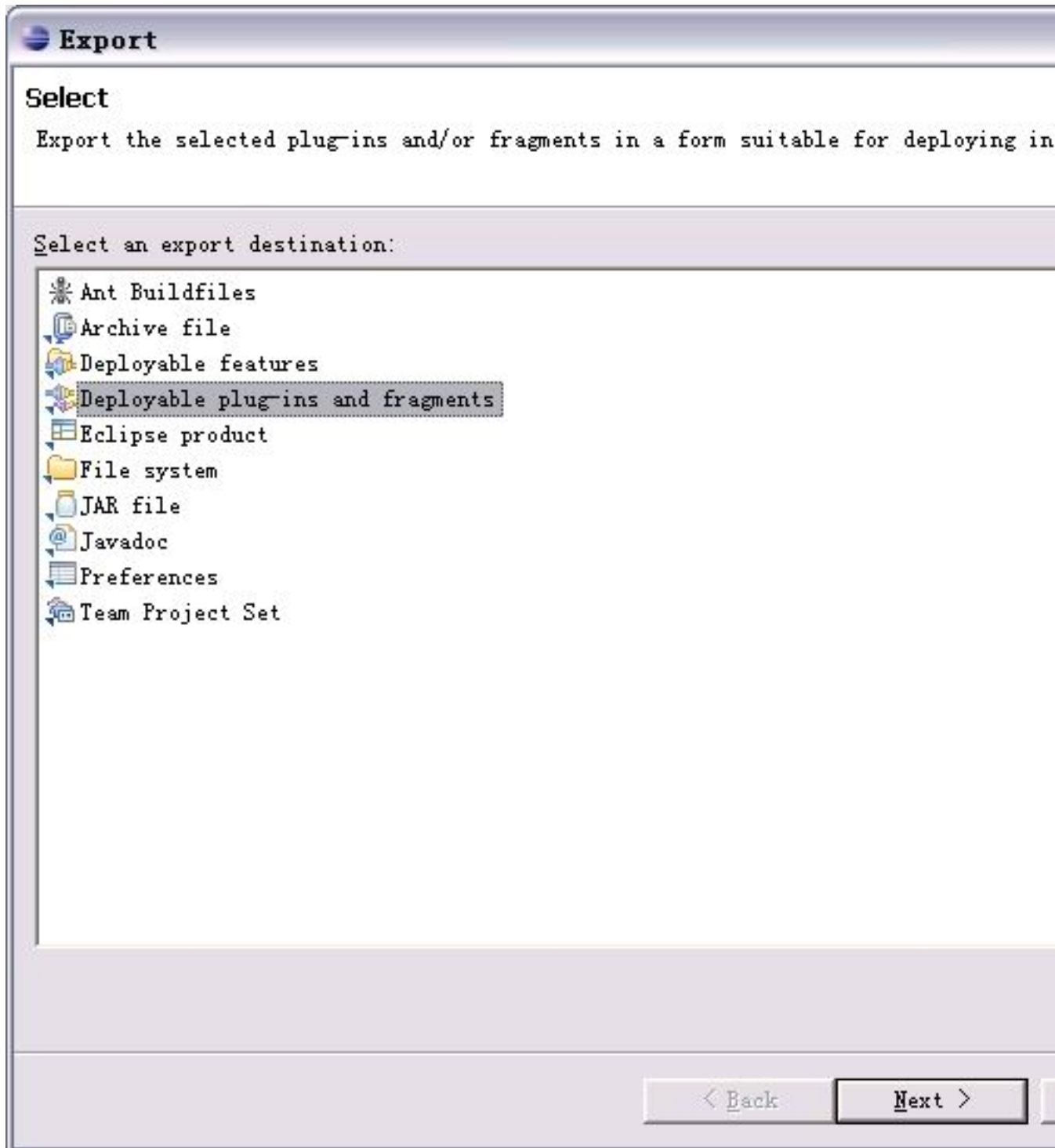


- e) Save the changes you made to the property file MANIFEST.MF.
7. Export the output to a documentation plug-in.

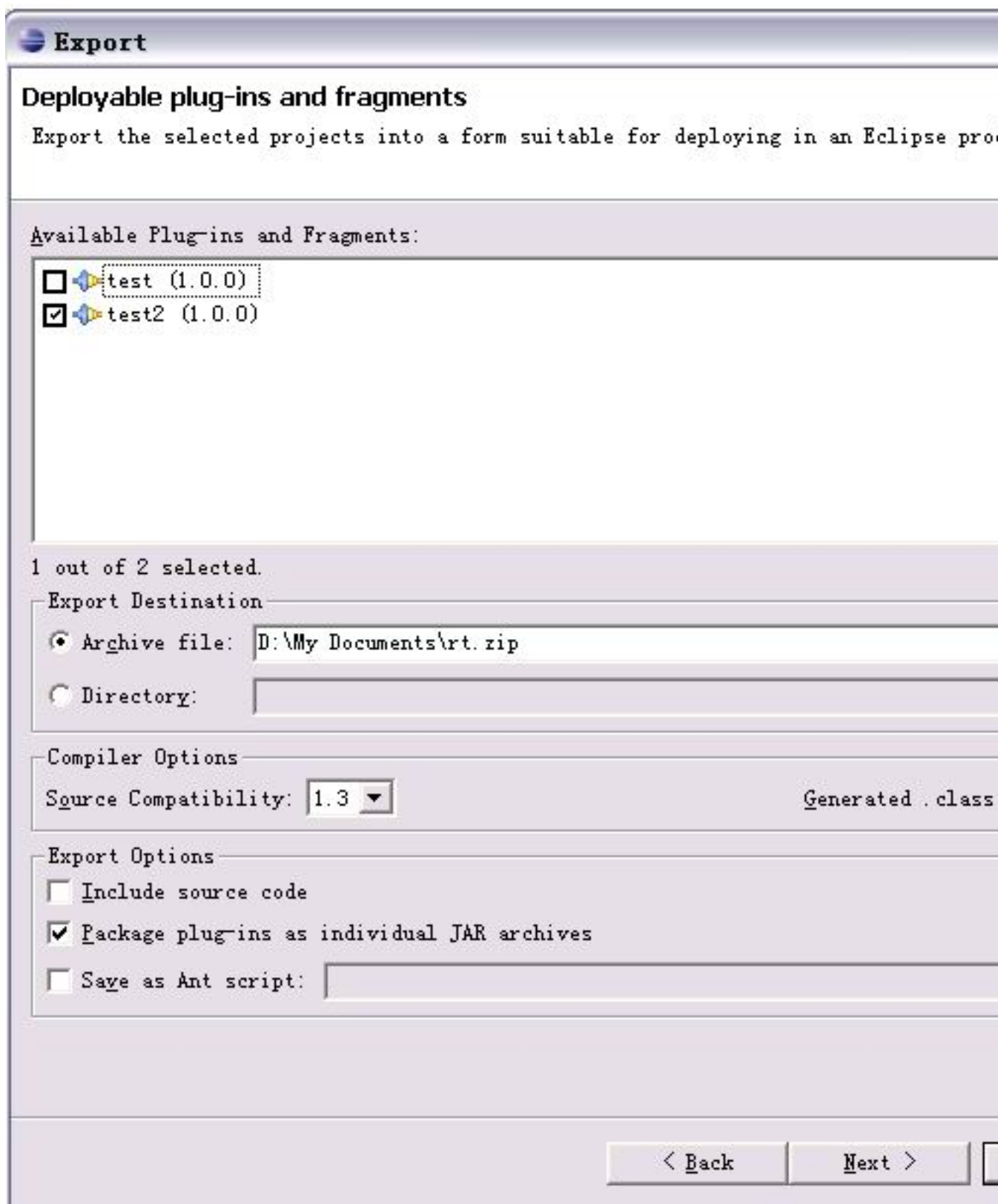


Note: build.xml can be customized to meet the requirement of headless build.

- a) Select **File > Export**; select **Deployable plug-ins and fragments** and click **Next**.



- b) Select the plug-in you want to export and specify a directory under which you want to put the plug-in package.



- c) Click **Finish** to export the plug-in package.

Appendix

A

DITA-OT release history

Topics:

- [DITA Open Toolkit Release 1.5.4](#)
- [DITA Open Toolkit Release 1.5.3](#)
- [DITA Open Toolkit Release 1.5.2](#)
- [DITA OT Release 1.5.1](#)
- [DITA OT release 1.5](#)
- [DITA OT release 1.4.3](#)
- [DITA OT release 1.4.2.1](#)
- [DITA OT release 1.4.2](#)
- [DITA OT release 1.4.1](#)
- [DITA OT release 1.4](#)
- [DITA OT release 1.3.1](#)
- [DITA OT release 1.3](#)
- [DITA OT release 1.2.2](#)
- [DITA OT release 1.2.1](#)
- [DITA OT release 1.2](#)
- [DITA OT release 1.1.2.1](#)
- [DITA OT release 1.1.2](#)
- [DITA OT release 1.1.1](#)
- [DITA OT release 1.1](#)
- [DITA OT release 1.0.2](#)
- [DITA OT release 1.0.1](#)
- [DITA OT release 1.0](#)
- [DITA history on developerWorks \(pre-Open Source\)](#)

DITA Open Toolkit Release 1.5.4

General Enhancements and Changes

Configuration file for defaults

In previous versions, `lib/configuration.properties` was generated by integration process. Integration has been changed to generate `lib/org.dita.dost.platform/plugin.properties` and the role of the old `lib/configuration.properties` has been changed to contain defaults and configuration options, such as default language.

Plug-in extension point for file extension configuration

New plug-in extension points have been added allow configuring DITA-OT behaviour based on file extensions.

Extension point	Description	Default values
<code>dita.topic.extension</code>	DITA topic	<code>.dita, .xml</code>
<code>dita.map.extensions</code>	DITA map	<code>.ditamap</code>
<code>dita.html.extensions</code>	HTML file	<code>.html, .htm</code>
<code>dita.resource.extensions</code>	Resource file	<code>.pdf, .swf</code>

Both HTML and resource file extensions are used to determine if a file in source is copied to output.

New plug-in extension point has been added to allow declaring transtypes as print types.

Extension point	Description
<code>dita.transtype.print</code>	Declare transtype as a print type.

Strict integration mode

Two modes have been added to integration process: lax and strict. In strict mode the processing will fail if any errors are encountered. In lax mode an error message may be thrown for an error and the integration process will try to run to the end, even if there are errors that were unrecoverable. The default mode is lax.



Note: In lax mode, even if the process runs to the end and reports a successful result, DITA-OT may not be able to function correctly because of e.g. corrupted plug-in files or unprocessed template files.

Code reference charset support

Encoding of the code reference target file can be set using the `format` attribute, for example

```
<coderef href="unicode.txt" format="txt; charset=UTF-8"/>
```

Plugin URI scheme

Support for plugin URI scheme has been added to XSLT stylesheets. Plug-ins can refer to files in other plug-ins without hard coding relative paths, for example

```
<xsl:import href="plugin:org.dita.pdf2:xsl/fo/topic2fo_1.0.xsl"/>
```

PDF

Support for mirrored page layout has been added. The default is the unmirrored layout.

The `args.bookmap-order` property has been added to control how front and back matter are processed in bookmarks. The default is to reorder the frontmatter content as in previous releases.

A new extension point has been added to add mappings to the PDF configuration catalog file.

Extension point	Description
<code>org.dita.pdf2.catalog.relative</code>	Configuration catalog includes.

Support for the following languages has been added:

- Finnish
- Hebrew
- Romanian
- Russian
- Swedish

PDF processing no longer copies images or generates XSL FO to output directory. Instead, the temporary directory is used for all temporary files and source images are read directly from source directory. The legacy processing model can be enabled by setting `org.dita.pdf2.use-out-temp` to `true` in configuration properties; support for the legacy processing model may be removed in future releases.

Support for FrameMaker index syntax has been disabled by default. To enable FrameMaker index syntax, set `org.dita.pdf2.index.frame-markup` to `true` in configuration properties.

A configuration option has been added to disable I18N font processing and use stylesheet defined fonts. To disable I18N font processing, set `org.dita.pdf2.i18n.enabled` to `false` in configuration properties

XHTML

Support for the following languages has been added:

- Indonesian
- Kazakh
- Malay

Migration from previous releases

The `print_transtypes` property in `integrator.properties` has been deprecated in favor of `dita.transtype.print` plug-in extension point.

The `dita.plugin.org.dita.*.dir` properties have been changed to point to DITA-OT base directory.

PDF

Support for mirrored page layout was added and the following XSLT configuration variables have been deprecated:

- `page-margin-left`
- `page-margin-right`

The following variables should be used instead to control page margins:

- `page-margin-outside`
- `page-margin-inside`

XSLT Parameters `customizationDir` and `fileProfilePrefix` have been removed in favor of `customizationDir.url` parameter.

A new shell stylesheet has been added for FOP and other shell stylesheets have also been revised. Plug-ins which have their own shell stylesheets for PDF processing should make sure all required stylesheets are imported.

Font family definitions in stylesheets have been changed from Sans, Serif, and Monospaced to sans-serif, serif, and monospace, respectively. The I18N font processing still uses the old logical names and aliases are used to map the new names to old ones.

SourceForge trackers

Feature requests

- 3333697 Add strict mode processing (Milestone 1)
- 3336630 Add resource file extension configuration (Milestone 1)
- 3323776 Base HTML stylesheets (Milestone 1)
- 3355860 Enable defining code ref target encoding (Milestone 1)
- 3393969 Make default TocJS output more usable (Milestone 3)
- 3394708 cfg/catalog.xml should be an extension point (Milestone 4)
- 3411030 Add args.fo.userconfig to PDF2 (Milestone 5)
- 3411961 Change margin-* to space-* property (Milestone 5)
- 3412144 Add FOP specific shell to PDF2 (Milestone 5)
- 3413215 Add schemas for PDF2 configuration files (Milestone 5)
- 3414416 Support bookmap order in PDF2 front and back matter (Milestone 5)
- 3413933 Fix inconsistencies in PDF2 page headers (Milestone 5)
- 3418877 Mechanism to refer to other plug-ins in XSLT (Milestone 5)
- 3411476 Add extension point for print type declaration (Milestone 6)
- 3392891 Copy the graphic files to the temporary folder (Milestone 6)
- 3429290 Remove unused Apache Commons Logging JAR (Milestone 6)
- 3434640 Add XHTML NLS support for Indonesian, Malay, Kazakh (Milestone 6)
- 3435528 Add base configuration file (Milestone 7)
- 3432219 Refactor dita.list read and write (Milestone 7)
- 3401849 PDF2: runtime switch for localization post-processing (Milestone 7)
- 3438361 Add "tocjs" transform to demo script (Milestone 7)
- 3341648 Clean HTML and XHTML stylesheets (Milestone 8)
- 3343562 Java clean-up (Milestone 8)
- 3346094 Improve test coverage (Milestone 8)
- 3372147 Improve logging (Milestone 8)
- 3373416 Refactor PDF attribute sets (Milestone 8)
- 3376114 Improve PDF page layout configuration (Milestone 8)
- 3415269 Support for more languages in the PDF transform (Milestone 8)
- 3412211 Refactor PDF index stylesheet for XSL 1.1 support (Milestone 8)
- 3425838 General PDF2 improvements (Milestone 8)
- 3428152 General I18N improvements (Milestone 8)
- 3429390 General XHTML improvements (Milestone 8)
- 3438790 Clean up build_demo script (Milestone 8)
- 3440826 Dutch patch for feature request 3415269 (Milestone 8)
- 1785391 Make Java code thread-safe (in progress)

Patches

- 2963037 PDF changes to fix index rendering of colon (bug 2879196) (Milestone 7)

Bugs

- 2714699 FO plug-in doesn't support specialized index elements (Milestone 1)
- 2848636 Duplicate key definitions should produce info messages (Milestone 1)
- 3353955 Frontmatter child order is not retained in PDF2 (Milestone 1)
- 3354301 XRef with conreffed phrases not properly generate HTML link (Milestone 1)
- 3281074 Bad attribute being applied to fo:bookmark-title element (Milestone 2)
- 3344142 Conref Push order of validation (Milestone 2)
- 3358377 Cryptic error message when DITA Map has "bookmap" extension (Milestone 3)
- 3384673 ODF transtype no longer embeds images in output (OT 1.5.3) (Milestone 3)
- 3394000 TocJS needs cleanup for several minor bugs (Milestone 3)
- 3392718 TOCJS sample should not require ant target (Milestone 3)
- 3389277 DocBook transform redundantly nests Related Links (Milestone 3)
- 3105339 '<' and '>' characters in a title cause tocjs trouble (Milestone 3)
- 3104497 tocjs JavaScripts don't work in Japanese environment (Milestone 3)
- 3394130 Remove outdated developer documentation (Milestone 3)
- 3397165 chunk on topichead not honored (Milestone 4)
- 3397501 Custom reltable column headers are reversed (Milestone 4)
- 3397495 Relcolspec with <title> does not generate link group headers (Milestone 4)
- 3399030 <ph> Elements not flagged with alt-text in HTML output (Milestone 4)
- 3396884 NPE in EclipseIndexWriter.java<Merges,setLogger for AbstractIndexWriters (Milestone 4)
- 3398004 -d64 flag to JVM not allowed for Windows JVMs (Milestone 4)
- 3401323 Fix PDF nested variable handling (Milestone 4)
- 3401721 Processing broken for <topicsetref> elements (Milestone 4)
- 3404049 Setting of clean_temp is backwards (Milestone 4)
- 3386590 Product name repeated hundreds of times in PDF (Milestone 4)
- 3405417 Shortdesc output twice when using abstract (Milestone 4)
- 3402165 wrong image output dir if using generate.copy.outer=2 (Milestone 4)
- 2837095 Positions of index and TOC in bookmaps are ignored (Milestone 5)
- 3414826 DITA OT not handling image path with chunking turned on (Milestone 5)
- 3411767 Not so meaningful messages given by ImgUtils (Milestone 5)
- 3405851 Incorrect entry@colname in merged XML with row and colspan (Milestone 5)
- 3406357 Custom profiling issue (Milestone 5)
- 3413203 Remove references to OpenTopic in PDF2 (Milestone 5)
- 3414270 @props specialization not used in map (Milestone 5)
- 3383618 Attribute 'link-back' cannot occur at element 'fo:index-key (Milestone 5)
- 3418953 Scale computation for XHTML uncorrectly looks up images (Milestone 6)
- 3413229 onlytopic.in.map & symlink (Milestone 6)
- 3423537 Additional line breaks in <menucascade> should be ignored (Milestone 6)
- 3423672 Problems with refs to images outside the DITA Map directory (Milestone 6)
- 2879663 indexterm/keyword causees NullPointerException (Milestone 7)
- 2879196 Colon character in <indexterm> causes nesting in output (Milestone 7)
- 3179018 Indexterm with only nested subelement results in NPE (Milestone 7)
- 3432267 Task example title processing incorrect for PDF (Milestone 7)
- 3430302 Unitless images sizes in throw errors (Milestone 7)
- 3429845 No variables for Warning (Milestone 7)
- 3428871 topicmerge gives incomplete topicref when reference or topic (Milestone 7)
- 3132976 Duplicate index text in index page (Milestone 7)
- 2795649 Java topicmerge ignores xml:lang (Milestone 7)
- 3431798 Relative CSS paths incorrectly computed for @copy-of (Milestone 7)
- 3438421 Remove transtype default (Milestone 7)

- 2866342 Nested see also is ignored (Milestone 8)
- 1844429 PDF2: Non-DITA link broken unless marked external (Milestone 8)
- 3270616 "lcTime" not displayed in PDF output (Milestone 8)
- 3388668 Data in figure captions not suppressed in xrefs (Milestone 8)
- 3429824 topicmerge gives wrong topicref with nested topics (Milestone 8)
- 3414332 PDF2 variable string translations missing (Milestone 8)
- 3323806 Improve Java logging and exception handling (Milestone 8)
- 3426920 Image files not copied or referenced correctly for eclipse (Milestone 8)
- 3445159 entry/@colname has been removed! (Milestone 8)
- 3447732 Bug in handling of longdesc ref (Milestone 8)
- 3452510 Ant parameter customization.dir not documented anywhere (Milestone 8)
- 3451621 Revisions on <plentry> use wrong image for nested <pd> (Milestone 8)

DITA Open Toolkit Release 1.5.3

Release 1.5.3 is a maintenance release based on the final version of the DITA 1.2 standard.

Version 1.5.3 contains many enhancements, user patches, bug fixes, and significant updates to the documentation.

Release 1.5.3 was developed using a series of test builds released to the community every three weeks. Each item in the list below indicates which test build first contained the update. The eighth public build was the final build, released as the DITA-OT 1.5.3 final stable build.

General Enhancements and Changes

Base plug-ins

In earlier releases of OT, configuration parameters were hardcoded into Ant files and Java code. Starting from version 1.5.3 OT has externalized base configurations into base plug-ins in plugins folder. Base plug-in identifiers and folder names start with `org.dita`:

- `org.dita.base`
- `org.dita.docbook`
- `org.dita.eclipsecontent`
- `org.dita.eclipsehelp`
- `org.dita.htmlhelp`
- `org.dita.javahelp`
- `org.dita.odt`
- `org.dita.pdf`
- `org.dita.troff`
- `org.dita.wordrtf`
- `org.dita.xhtml`

For backwards compatibility, only configuration files were moved to plug-in folders, the actual code and resource files were left in original locations.

Installations of OT may remove base plug-ins in order to remove functionality, but the `org.dita.base` plug-in must be retained as it contains configuration for base functionality such as catalog files and preprocessing.

Plug-in configuration changes

The plug-in configuration file `plugin.xml` has support for new syntax, where the old

```
<feature extension="foo"
  value="bar.xml" type="file"/>
```

can be written as

```
<feature extension="foo"
  file="bar.xml"/>
```

The new `file` attribute only supports a single file, not a comma separated list like the `value` attribute.

In previous releases multiple feature elements with the same extension ID were not supported. In release 1.5.3 multiple definitions are combined, thus

```
<feature extension="foo"
  value="bar,baz"/>
```

can also be written as

```
<feature extension="foo"
  value="bar"/>
<feature extension="foo"
  value="baz"/>
```

Plug-in extension points can be added with

```
<extension-point id="extension-id"
  name="human readable name"/>
```

Plug-ins **should** declare all extension points they support. In version 1.5.3 undeclared extension points are supported, but a warning is thrown when running integration in verbose mode. Support for undeclared extension points **may** be removed in future releases.

PDF2 changes

Support for the `format` attribute in PDF2 variable files has been removed as redundant. The same functionality as

```
<variable id="foo"
  format="bar">baz</variable>
```

can be implemented with e.g.

```
<variable id="foo.bar">baz</variable>
```

PDF2 no longer logs a warning about PDF2 plug-in replacing the legacy PDF transformation type.

Support for flagging has been added.

Version of FOP that comes with Full Easy Install has been updated from 0.95 to 1.0.

Filtering configuration

List of transtypes which are considered to be print types has been moved to `integrator.properties` with the property name `print_transtypes`. In previous releases this list was hardcoded into Java code. Configuring print transtypes is currently not possible in plug-in configuration files.

Java API changes

Multiple Java classes have been changed from public non-final into package-private final. This enables clearer distinction between public and internal API, and forbids subclassing classes which have not been designed and documented for extensibility.

SourceForge Enhancements Added

1. 3177971 Improve plugin configuration file (Milestone 2)
2. 3178275 Add `xsl:import` extension point to PDF2 topic merge XSL (Milestone 3)
3. 3182113 Add common attribute processing to PDF2 plugin (Milestone 3)
4. 3185914 Improve integration login (Milestone 3)
5. 3189073 Plugin location should be available as Ant property (Milestone 3)
6. 3126848 Repository cleanup (Milestone 3)
7. 3204188 Support for defining extension point (Milestone 4)
8. 3213163 Clean PDF2 build and integration scripts (Milestone 4)
9. 3227387 Need extension to pass user param to `dita.map.xhtml.toc` targ (Milestone 4)
10. 3231695 Use an XML serializer object for writing XML (Milestone 4)
11. 3256796 Remove legacy PDF code from `xsl` (Milestone 4)
12. 3283638 Remove format attribute support from PDF2 vars (Milestone 5)
13. 3285716 Clean up PDF2 build files (Milestone 5)
14. 3286085 Add output and temp dir params to PDF2 (Milestone 5)
15. 3293738 Use extensible pipeline task implementation (Milestone 6)
16. 3271552 `${args.xsl.pdf}` as an absolut path not supported (Milestone 6)
17. 3033000 update to Apache FOP 1.0 release (Milestone 6)
18. 3213324 Separate FOP/XEP/AXF stylesheets in PDF2 (Milestone 6)
19. 3302779 Dependency extension points for PDF2 formatting (Milestone 7)
20. 3304945 Allow setting local overrides with properties file (Milestone 7)
21. 3190356 Pluginize DITA-OT base configuration files (Milestone 8)
22. 3167087 Reduce static variable usage in Java code (Milestone 8)
23. 3158929 Java clean-up (Milestone 8)
24. 3194917 Change Java API to be more final and non-public (Milestone 8)
25. 3197328 Refactor writers for cleaner XML serialization (Milestone 8)
26. 3199755 Improve log integration (Milestone 8)
27. 3296040 Refactor PDF2 build files (Milestone 8)
28. 3306146 PDF2 stylesheet refactoring (Milestone 8)
29. 3304447 Add support for selecting output format in PDF2 (Milestone 8)
30. 3310476 Add plug-in ID and version syntax check (Milestone 8)
31. 3309275 Warning reported by Apache FOP on any topic (Milestone 8)
32. 3305843 Support list of tables/figures in PDF2 (Milestone 8)

SourceForge Patches Added

1. 3123507 String concat in `map2plugin` (Milestone 1)
2. 3110513 HTML XSLT uses complex casts (Milestone 1)

3. 3097677 Add property to reload XHTML stylesheets (Milestone 1)
4. 3106659 Added topicgroup elements to tocjs (Milestone 1)
5. 3109051 RestoreEntity duplicates functionality (Milestone 1)
6. 3107755 Configure templates with integrator properties (Milestone 1)
7. 3142967 IndexTermReader leaves tab characters in terms (Milestone 1)
8. 3140543 Add missing Commons Codec JAR into compile classpath for buildPackage.xml (Milestone 1)
9. 3087664 Clean plugin configuration parser (Milestone 1)
10. 3145258 Plug-in integrator code clean-up (Milestone 2)
11. 3147226 Use common directory layout for Junit (Milestone 2)
12. 3062765 Fix unit test file paths to be platform dependent (Milestone 2)
13. 3164523 Refactor platform Java code (Milestone 2)
14. 3160801 Improve unit test coverage (Milestone 8)
15. 3189026 Avoid strings where other types are more appropriate (Milestone 8)

SourceForge Bugs Fixed

1. 3114411 keyref links don't work for HTML Help (Milestone 1)
2. 3126578 Chunking Issues in DITA 1.5.1 (Milestone 1)
3. 3109616 More Antenna House Path Problems (Milestone 1)
4. 3155375 Incorrect way to specify recognized image extensions(Milestone 2)
5. 3157890 Navtitle Construction Does not Preserve Markup (Milestone 2)
6. 3155848 xml decl in ditaval file not closed properly (Milestone 2)
7. 3162808 Chunking remaps in-file <xref> to invalid value (Milestone 2)
8. 3164866 Upper letter estensions (Milestone 2)
9. 3165307 Add boilerplate to Java files (Milestone 2)
10. 2793836 CHM Index terms come out with extra spaces (Milestone 2)
11. 3165762 Initializer XMLReader without modifying system variables (Milestone 2)
12. 3175328 Imagemap alt text gets extra text (Milestone 2)
13. 3085106 FO: topicmerge drops id on map/topicref without href. (Milestone 2)
14. 3147328 Error in commons.xml: getTopicrefShortdesc (Milestone 2)
15. 3130724 Error in tables.xml: fix-relcolwidth (Milestone 2)
16. 3174906 Normalize Map and Bookmark titles for JavaHelp output (Milestone 3)
17. 3178361 Conkeyref push fails when equivalent conref push succeeds (Milestone 3)
18. 3180681 PDF2: Inconsistent template import / include. (Milestone 3)
19. 3191701 Conref Push to Same File Fails (Milestone 3)
20. 3191704 Push Replace Results in Pushed element Being removed (Milestone 3)
21. 3189883 MapLinksReader should not be namespace aware (Milestone 3)
22. 3164587 Warnings issued by Saxon 9.3.0.4 when publishing to PDF (Milestone 3)
23. 3159001 Clean unit tests (Milestone 4)
24. 3199985 @chunk : xrefs and links break (Milestone 4)
25. 3206158 Inconsistent message DOTJ038W (Milestone 4)
26. 3206373 Better handling of referenced SVG images (Milestone 4)
27. 3279539 Out of memory error from move-meta module (Milestone 5)
28. 3281108 Fallback to \$locale when xml:lang value is wrong format (Milestone 5)
29. 3286679 ODT output transform deletes too many files (Milestone 6)
30. 3287609 Chunking rewrites image based on map directory (Milestone 6)
31. 3288639 Conref code improperly generalizes map domain elements (Milestone 6)
32. 3294295 PDF2 indexing and I18N fails with missing languages (Milestone 6)
33. 3294864 tocjs-demo: tocjs.ditamap is referencing a missing file (Milestone 6)
34. 3297930 PDF2: axf specific templates can't be overridden. (Milestone 7)
35. 2001271 DITA-OT documentation wants Ant 1.6.5 (Milestone 8)

- 36. 3136773 Incorrect version reported in log file (Milestone 8)
- 37. 3260746 Topichead not processed the same as title-only topic (Milestone 8)
- 38. 3315029 Garbled character problem in Japanese HTMLHelp (Milestone 8)
- 39. 3308775 Keyref map in grandparent folder fails (Milestone 8)

DITA Open Toolkit Release 1.5.2

Release 1.5.2 is a maintenance release based on the final version of the DITA 1.2 standard.

- **Release date:** December 10, 2010
- **Supports:** DITA 1.0 through 1.2
- **Download at:** [DITA-OT Latest Stable Build](#)
- **Which package is for me?** See [DITA-OT Packages](#).

In addition to tweaks to match late changes in the standard, version 1.5.2 contains many enhancements, user patches, bug fixes, and significant updates to the documentation.

Release 1.5.2 was developed using a series of test builds released to the community every three weeks. Each item in the list below indicates which test build first contained the update. The eighth build was the final build, released as the DITA-OT 1.5.2 final stable build.

General Enhancements

1. Include final version of DITA 1.2 schemas and DTDs
2. Minor updates to DITA 1.2 support added in earlier releases, to ensure compliance with the final standard
3. Overhaul of documentation to remove outdated material
4. Reorganization of doc directory to highlight new and important info

11 SourceForge Enhancements Added

1. 2797337 Support for ODF output transform (Prototype added in version 1.5.1, updates in each 1.5.2 Milestone, transform complete in Milestone 7)
2. 3021544 Preserve DITA elements in XHTML class by default (Milestone 1)
3. 3019853 Create new "textonly" output method for use by any transform (Milestone 1)
4. 3012392 PDF transformation should allow args.xsl style override (Milestone 1)
5. 2882123 Add Ant Quick Start Guide to DITA-OT (Updated in each Milestone after 3)
6. 3086936 Add extension points for TOC output (Eclipse TOC, HTML Help TOC and Project, HTML TOC) (Milestone 6)
7. 3079610 Add current OT version to log (Milestone 6)
8. 1520909 HTML Help requires appropriate codepage (Milestone 6)
9. 3125994 Allow PDF index conf. to be overridden in Customization (Milestone 7)
10. 3125983 Create a basic glossary implementation for PDF (Milestone 7)
11. 3109395 Add parameter for Eclipse symbolic name (Milestone 8)

SourceForge Patches Added

1. 3058008 Refactor chunk module for cleaner code (Milestone 4)
2. 3067681 Add class to ordered child links (Milestone 5)
3. 3064412 Integrator fails to escape XML correctly (Milestone 5)
4. 3062765 Fix unit test file paths to be platform dependent (Milestone 5)
5. 2949860 PDF build.xml with args for JVM memory and architecture (Milestone 6)
6. 3077935 Plug-in ignore in Integrator (Milestone 6)
7. 3065050 Common logging interface (Milestone 6)
8. 3063318 ChunkModule refactoring (Milestone 6)

9. 3061100 Define AbstractPipelineInput's function (Milestone 6)
10. 3102905 Move supported image extensions to configuration file (Milestone 7)
11. 3097518 Show effective property values (Milestone 8)
12. 3101335 apiMap.mod missing from catalog (Milestone 8)

SourceForge Bugs Fixed

1. 2928582 commonTopicProcessing template prolog processing out of order (Milestone 1)
2. 2823221 version of Xalan-J inconsistent (Milestone 1)
3. 3023642 Invalid @colname generated in nested table (Milestone 1)
4. 3016739 Chunking mixes up <link> to topic in reltable (Milestone 1)
5. 3020314 Chunk output includes index terms in navtitles (Milestone 1)
6. 3020313 Chunk processor adds <topicref> before <topicmeta> (Milestone 1)
7. 3031513 Nested table processing in pdf2 (Milestone 2)
8. 3030317 Filtering doesn't work on @rev or @props attributes (Milestone 2)
9. 3028650 Replace xs:float with xs:double in Plus plugins (Milestone 2)
10. 3022847 PDF transform gives Java exceptions for spaces in dir name (Milestone 2)
11. 3032950 Scale is not correctly computed in XHTML transforms (Milestone 2)
12. 3033141 dita.xml.properties file not closed after generating (Milestone 2)
13. 3034445 "CURRENDDIR" typo in plus-plugins (Checked in to CVS during Milestone 3)
14. 3034489 Remove all occurrences of <xmlcatalog> from plus-plugins (Checked in to CVS during Milestone 3)
15. 3035816 When creating .chm, .hhp-file is missing a line-break (Milestone 3)
16. 3036222 RTF transform not editable with Word 2007 (Milestone 3)
17. 3036985 Infinite recursivity in replaceString template (Milestone 3)
18. 3038941 Link with & breaks in abstract (Milestone 3)
19. 3039017 Comments in PDF plugin files are confusing (Milestone 3)
20. 3058124 Toolkit Allows Unescaped URLs, doesn't handle escaped ones (Milestone 4)
21. 3056939 Conref of keyref-based xref results in xref with no href (Milestone 4)
22. 3052913 Multiple levels of keyref in map not resolved (Milestone 4)
23. 3052904 Keydef with no href causes hard failure (Milestone 4)
24. 3052156 Object with data that starts with slash breaks image copying (Milestone 4)
25. 3044861 Inappropriate warning for resource-only topic to graphic (Milestone 4)
26. 3042978 @copy-to and @chunk on topichead gives file not found (Milestone 4)
27. 3016994 The included-domains entity cannot be used in document (Milestone 4)
28. 2994593 Transformation breaks when DITA Topics contain entity refs (Milestone 4)
29. 3028894 no support for title in plugin.xml file (Milestone 4)
30. 3065853 Indent from <title> gets displayed in TOC (PDF) (Milestone 5)
31. 3065486 CURRENTFILE not aware of DITAEXT (Milestone 5)
32. 3065422 Wrong filename and filedir parameters for eclipse xsl (Milestone 5)
33. 3063533 Adjacent words get glued together using DITA to RTF (Milestone 5)
34. 3062912 Messages extension damages custom message formatting (Milestone 5)
35. 3059256 Peer links break with missing format or wrong extension (Milestone 5)
36. 2972393 Need to parameterize maxmemory and VM args for forked JVMs (Milestone 6)
37. 3060269 Problem displaying French content TOC in CHM output (Milestone 6)
38. 3038412 zh-CN file for PDF puts English strings in output (partial fix) (Milestone 6)
39. 3079676 <navtitle> contents included in PDF output (Updated so that <navtitle> in a topic will only appear when the draft parameter is set to 'yes') (Milestone 6)
40. 3004895 XHTML output for <draft-comment> should use class attribute (Milestone 6)
41. 2794487 No Easy Way to Override/Extend HTML TOC Navtitle Generation (Milestone 6)
42. 3088314 Need to clarify many error messages (Milestone 6)
43. 3095233 Shortdesc metadata missing when using abstract (Milestone 6)

- 44. 3081597 conkeyref accepts values in conref style (Milestone 6)
- 45. 3081459 fragment generation without plugin fails (Milestone 6)
- 46. 3073262 missing terminating quote in bundle version (Milestone 6)
- 47. 2832863 <group> elements in catalogs don't work for all editors (Milestone 6)
- 48. 3038933 Troff output drops prereq links (Milestone 7)
- 49. 3098975 Disable Output Escaping Should Not Be Used (Milestone 7)
- 50. 3102827 Allow a way to specify recognized image extensions (Milestone 7)
- 51. 3102219 Unexpected character code in Japanese string definition (Milestone 7)
- 52. 3101964 Unnecessary XML declaration in HHP and HHC (Milestone 7)
- 53. 3095233 Shortdesc metadata missing when using abstract (Milestone 7)
- 54. 3097409 PDF should skip empty columns in property tables (Milestone 7)
- 55. 3090803 PDF fails when chunk specified and topic appears twice (Milestone 7)
- 56. 3102845 Japanese character-set definition (Milestone 7)
- 57. 3103488 Update Saxon command line args for IDIOM PDF build.xml (Milestone 7)
- 58. 3026627 side-col-width variable has no effect (Milestone 7)
- 59. 3126007 TOC entries not properly indented in PDF (Milestone 7)
- 60. 3109616 Update PDF plug-in to check for latest Antenna House dirs (Milestone 8)
- 61. 3056040 problematic Bundle-Version test in eclipseMap (Milestone 8)
- 62. 3029074 Index file not generated by default for Eclipse Help (Milestone 8)
- 63. 3086552 XMLReader.parse does not correctly receive the XML system ID (Milestone 8)
- 64. 3110418 Duplicate @colname generated for entry (Milestone 8)
- 65. 3114353 Java sun.* packages should not be used (Milestone 8)

DITA OT Release 1.5.1

Release 1.5.1 is a maintenance release based on Committee Draft 01 of the DITA 1.2 standard.

- **Release date:** June 18, 2010
- **Supports:** DITA 1.0 through 1.2 (Committee Draft 02 level)
- **Download at:** [DITA-OT Latest Stable Build](#)
- **Which package is for me?** See [DITA-OT Packages](#).

This is the same version of the standard used for the DITA 1.2 Public Review. Release 1.5.1 contains many fixes and minor enhancements. It also includes a preview of a new output transform to the Open Document Format; this transform will be completed in a later release.

Release 1.5.1 was developed using a series of test builds released to the community every three weeks. Each item in the list below indicates which test build first contained the update. There were seven total test builds.

General Enhancements

1. Update to latest copy of DITA 1.2 Draft DTDs and Schemas (last update in Milestone 5)

14 SourceForge Enhancements Added

1. 2797337 Support for ODF output transform (first prototype available in Milestone 2, further updates in each milestone)
2. 2703335 Reduce duplicated code in XHTML <note> processing (Milestone 3)
3. 2976463 Provide finer grained control of links in PDF (include reltable and in-topic links, without parent/child links) (Milestone 4)
4. 2971536 New Java options for existing Ant parameters (Milestone 4)
5. 2979084 Obey the "args.draft" parameter (Milestone 5)
6. 2990783 allow caller-provided IndexTermCollection (Milestone 6 contains the core code updates; M7 contains the full enhancement)

7. 3001750 Shortdesc from map should override link description in PDF (Milestone 7)
8. 3004305 Notes with type="warning" need styling / localization in XHTML (Milestone 7)
9. 3004859 "eclipsecontent" transform should drop debug info (Milestone 7)
10. 2892706 Don't delete the FO file (new option to preserve topic.fo) (Milestone 7)
11. 2928584 Add general model for end-of-topic processing in PDF (Milestone 7)
12. 3006675 Support new DITA 1.2 <stepsection> element in PDF (Milestone 7)
13. 3006847 Add generated task headers to PDF (using the option that works for XHTML in DITA-OT 1.5) (Milestone 7)
14. 2987769 Add support for Eclipse Help index redirects (Milestone 7)

2 SourceForge Patches Added

1. 2914475 Use Xerces Grammar Pool to Improve Performance (Milestone 1)
2. 2991688 Refine package build Ant (Milestone 6)

46 SourceForge Bugs Fixed

1. 2916469 @locktitle not respected by preprocessor (Milestone 1)
2. 2917809 need empty lib/saxon directory for minimum and standard pkg. (Milestone 1)
3. 2925636 Build fails when excluded section contains a table (Milestone 1)
4. 2926417 Absolute file: URLs are not resolved. (Milestone 1)
5. 2930109 Move meta module pushes content into peer topic. (Milestone 1)
6. 1852808 args.css requires dummy file. (Milestone 1)
7. 2952956 Imagemap processing drops outputclass from image (Milestone 3)
8. 2953706 Minor improvements to "garage" samples (Milestone 3)
9. 2961909 /onlytopicinmap does not respect conref (Milestone 3)
10. 2957456 conkeyref breaks when topic is in subdir (Milestone 3)
11. 2962813 stepsection throws off numbering in links to steps (Milestone 3)
12. 2959588 Template Processor Doesn't handle XSLT atts that require ' (Milestone 3)
13. 2914574 plus-htmlhelp-alias-map: using same extension point twice (Milestone 3)
14. 2957938 coderef not working everytime (Milestone 3)
15. 2962781 html documentation out of date (Milestone 3)
16. 2952956 refactored XSL code in ut-d.xsl (Milestone 3)
17. 2954819 NullPointerException while processing simple BookMap (Milestone 3)
18. 2954154 Updated default version from 1.0 to 1.0.0 (Milestone 3)
19. 2970471 XSLFO test for @compact wrong (Milestone 4 for PDF, Milestone 5 for LegacyPDF)
20. 2972043 Setting onlytopicinmap causes a blank imagelist (Milestone 4)
21. 2974667 Integrator adds spaces into XML Catalog entries (Milestone 4)
22. 2986492 Duplicate parameter in XHTML code (Milestone 5)
23. 2982485 Cannot read a document that was written during the same transform (Milestone 5)
24. 2981216 <tm> @tmclass requires IBM-specific values (Milestone 5)
25. 2979361 Java stack traces in OT log (Milestone 5)
26. 2979328 Output parameters info at INFO level (Milestone 5)
27. 2978858 keyref processing doesn't respect basedir (Milestone 5)
28. 2990162 Conref to elements in same DITA file throw parsing errors (Milestone 6)
29. 3000677 msgph and systemoutput should use <samp> instead of <tt> (Milestone 7)
30. 3004220 <tm> elements dropped when keyref text resolved (Milestone 7)
31. 2987322 Navtitle attribute of glossarylist breaks PDF (Milestone 7)
32. 2916474 Inappropriate match on mapgroup/topichead in PDF code (Milestone 7)
33. 2916408 Topichead with <navtitle> not processed same as @navtitle in PDF (Milestone 7)
34. 3006443 CSS for prereq links indents prereq element (Milestone 7)

35. 2607892 (Plus Plug-in) plus-allhtml-encoding: map x-windows-950 to Big5 (Milestone 7)
36. 2385466 Handle @font-family="inherit" (Milestone 7)
37. 2928540 Shortdesc should align with body in PDF (use topic__shortdesc) (Milestone 7)
38. 1839827 PDF does not properly process <xref> to elements (Milestone 7)
39. 2521819 PDF topic title widowed due to fo:marker placed in separate block (Milestone 7)
40. 1385654 docbook/topic2db.xsl - better linking support (Milestone 7)
41. 3004550 Conkeyref does not work if key referenced is not in same folder (Milestone 7)
42. 3004060 keyref/id not producing link in PDF output (Milestone 7)
43. 3001705 conkeyref doesn't work across ditamaps (Milestone 7)
44. 3000604 Legacy PDF: empty @column-number causes errors (Milestone 7)
45. 3013079 Keyref handling does not respect scope="external" (Final build)
46. 3005748 XHTML: Topics w/ @print set to "printonly" are included (Final build)

DITA OT release 1.5

Release 1.5 is a major release based on the new DITA 1.2 draft standard.

It contains full support for DITA 1.2 draft as defined in the fall of 2009 (prior to public review).

In addition to DITA 1.2 support, release 1.5 contains the following updates, which are available in Milestone 21 or earlier of the DITA-OT test builds:

General Enhancements

1. New parameter to generate output for only the topics referenced in a map
2. Use fileparameter in Ant 1.7 to replace processing instruction in intermediate dita files
3. Determine the version of DITA-OT via the class org.dita.dost.util.Version
4. Remove duplicate ids in PDF topicmerge
5. Move Notices before the TOC, suppress in the TOC and suppress the second copy after the TOC
6. Include @scope="peer" condition when creating rel-links.

SourceForge Enhancements Added

1. 2859612 Add support for Serbian (Latin)
2. 2845278 Add Ant parameters for XSLT parameters
3. 2824371 Support Hindi and Urdu for XHTML output
4. 2790755 Process in unique temp directories (Designed to allow multiple builds to take place at once)
5. 2780998 startcmd.sh: Run the user's shell, not sh
6. 2698921 Add a way to set attributes on <body>
7. 2120219 Add PDF to the supported image
8. 2002857 homeID in javahelp .hs file is not set
9. 1725284 add support for headings for sections within task (XHTML only)
10. 1623246 Support RFC4646 language tags in Index modules
11. 1367897 Xref content generation enforces formatting choices
12. 2882109 Convert named PDF templates to mode template
13. 2883406 Add extension point for modifying PDF link text
14. 2882870 Add parameter to control PDF Chapter style
15. 2882103 Provide hook for specializations to add custom headers
16. 2882072 Add parameter to control PDF bookmarks (default collapsed)

93 SourceForge Bugs Fixed

1. 2860433 Keyref on <image> fails to resolve

2. 2860199 Chunk to-content in submap resulting in missing output
3. 2860168 Suppress data and data-about in PDF output
4. 2857167 conrefImpl.xsl 1.6.4.13 has duplicate variable decls
5. 2856742 Problem of keys attribute in nested topicref elements
6. 2854546 Peer xref leads to invalid destination error in PDF
7. 2849200 Style on ditaval prop or revprop is ignored
8. 2846111 Unnecessary text generated for external xref
9. 2845598 1.5-M18-demo/fo/Customization pdf.formatter
10. 2842753 catalog-dita.xml has invalid entries
11. 2839035 Chunk code cannot parse xref with &
12. 2832696 Move meta module pushes content into non-DITA file
13. 2829350 Build_demo.xml places files in incorrect directory
14. 2826143 Move meta module discards link text
15. 2824907 1.5 M17 Bug - FileNotFoundException export.xml eclipsehelp
16. 2819853 missing nested indexterm after 3rd
17. 2815492 PDF: keep-with-next on shortdesc
18. 2815485 Prolog indexterm not processed properly for PDF
19. 2813082 Eclipse help MANIFEST.MF filename wrong case, needs CRLF
20. 2811980 pdf2: japanese translate for generated page number of a ref
21. 2811358 Incorrect entry@colname in middle file at PDF generation
22. 2804442 Bad param name in prereq-fmt template's call to sect-heading
23. 2799543 Child links in HTML Help output are wrong
24. 2797030 Chunking fails with bookmap
25. 2796614 Leading slash breaks PLUGINS_ROOT usage for Eclipse help
26. 2791696 reltable DITA 1.2 (#12048)
27. 2791345 Topicmeta searchtitles in map not used in topic
28. 2791278 Keyref Resolution Fails for Non-Descendant Topics
29. 2790807 Demo code should use new PDF output
30. 2788069 Topicmerge does not handle copy-to properly
31. 2788069 Topicmerge does not handle copy-to properly
32. 2782503 Extra space before bullet list in xhtml output DITA-OT 1.4.3
33. 2774128 startcmd.sh is encoded with DOS line endings in v1.5 M13
34. 2759964 HTML outputs filters conditional topics poorly
35. 2759964 HTML outputs filters conditional topics poorly
36. 2748371 Revision + ID gives XSL error
37. 2739236 plus-allhtml-svgobject handles <alt> badly
38. 2724090 XHTML: conreffed by-reference footnotes break w/ chunking
39. 2723928 gen-toc template not matching properly
40. 2723715 Itemgroup sets @id when element is not open
41. 2712074 XHTML: chunk=to-content on map breaks by-reference footnotes
42. 2706725 Single quote inserted in empty table cells in FO
43. 2696229 FO plugin code incompatible with Saxon 9 (OT1.4.3 default)
44. 2696191 Java libraries missing from 1.4.3 distributions
45. 2647292 HTML stylesheets should style <wintitle> and <filepath>
46. 2629271 maplink: should not modify href if scope=external
47. 2629256 mapref: should not modify href if scope=external
48. 2573681 Move link module does extra processing
49. 2547437 zh_TW.properties and zh_CN.properties contents are mixed up
50. 2418932 toc attribute does not work for references to ditamaps
51. 2317681 Extra files generated when many topics are chunked to one

- 52. 2317581 inline formatting in shortdesc
- 53. 1931457 Need to identify transtype during XHTML output step
- 54. 1629094 docbook creates empty simplelists
- 55. 1628936 transtype=docbook does not handle publisher and copyright element
- 56. 2849078 Problems using keyrefs with DITA OT 1.5 M19 - ID: 2849078
- 57. 2875373 tm in linktext is dropped
- 58. 2870935 keyref within topics ignore @copy-to attributes
- 59. 2873560 SWF flash not defined as resources to be copied
- 60. 2872954 Conref push not working at map level
- 61. 2871009 Temp directory leaves behind single directory and file
- 62. 2866322 Generated links have bad URLs
- 63. 2873654 PDF missing rules for new DITA 1.2 content elements
- 64. 2872988 Bad relcolwidth crashes PDF output
- 65. 2866204 Topicref with keyref fails to produce output
- 66. 2878446 Issue with ampersand in xrefs in attributes
- 67. 1629096 docbook creates invalid varlistentry
- 68. 2871326 Cannot use different TOC titles for same topic in PDF
- 69. 1880097 PDF2 ignores contents of <xref> elements
- 70. 1815571 Invalid property in fo:table-body
- 71. 2879171 Shortdesc & Abstract formatting is incorrect for PDF
- 72. 2882085 Obsolete code in PDF plug-in should be removed
- 73. 2661418 make the TOC in pdf2 output link to topics in pdf
- 74. 2871017 eclipsehelp plugin.xml invalid in 1.5M20
- 75. 2887331 chunk="to-content" on a sub-map causes path error
- 76. 2891736 indexterm in topicref level are copied into topic/prolog
- 77. 2893316 This is a bug in the integrator.xml
- 78. 2893493 \${clean.temp} is not set to true per documentation
- 79. 2893664 ampersand entity with conref actions causes build to fail
- 80. 2893924 htmlhelp transtype requires video plugin
- 81. 1628937 Rename supportingboth.ditaand.xmlinaditamap.dita
- 82. 1771123 Inappropriate Message 018 On type= value
- 83. 1819660 Eclipse Help issue producing toc file.
- 84. 1848313 ditaval file sent to XSLT not URL
- 85. 1897542 Chunk to-content may need to rewrite topic IDs
- 86. 2875946 PDF: table @pgwide semantics not implemented
- 87. 2893745 missing fop's file/directory in standard package
- 88. 2900047 map2htmtoc.xsl doesn't allow for extension of root template
- 89. 2900417 html transform does not use image/@scale properly
- 90. 2906957 Update Eclipse plugin DTDs and packaging
- 91. 2796964 Use xml:lang for generated text in PDF
- 92. 2860596 PullPrologIndexTerms selection too wide, Removed spurious pullPrologIndexTerms in section/title
- 93. 1852733 Image "scale" attribute is ignored

4 SourceForge Patches Added

- 1. 2790337 Add extension points to related-links code
- 2. 2778178 Flagging code duplicates XSL
- 3. 2715750 plus-transtype-htmlhelp: support alternate compilers
- 4. 2804311 Feature value trim test bug

DITA OT release 1.4.3

Release 1.4.3 is a maintenance release and is intended to be the last released based fully on DITA 1.1.

It contains a significant amount of preview function for the upcoming DITA 1.2 standard, but the DITA 1.2 document types must be specifically referenced in order to use this capability.

Major Updates

1. The PDF codebase switched from the older deprecated code to the plugin previously known as PDF2. Running a build with the transform type PDF or PDF2 will now run the new code. To continue using the older code, set your transform type to "legacypdf".
2. The PDF plugin now runs with FOP, as well as with the latest version of Saxon.
3. The Full Easy Install package now ships with Saxon 9B instead of with Xalan, and code in the DITA-OT package has been updated to work with Saxon 9B.
4. The Full Easy Install package updates the versions of many open source libraries (including Ant and FOP)
5. RFE 1982567 Allow spaces in DITA file names
6. Preview function for the majority of the upcoming OASIS DITA 1.2 standard, including everything from milestone builds one through twelve of DITA-OT 1.5

10 SF Enhancements Added

1. 1982567 Allow spaces in DITA file names
2. 2631145 dita2htmlImpl.xsl should not define doctype
3. 2562718 Rename the ant directory to avoid naming collisions with ant
4. 2314086 Let maplink generate <linklist> elements
5. 2258659 Extend integrator to allow addition of messages.
6. 2117337 Customize Integrator to integrate custom directories
7. 2074933 Make index output for new transtypes more extensible
8. 1995667 Add plugin extension points to preprocess pipeline
9. 1972537 Let users specify reloadstylesheet on xslt tasks
10. 1803199 Allow ability to include class ancestry in XHTML class attribute

4 SourceForge Patches Added

1. 1996733 Structure Ant <pipeline> better
2. 1796007 Refactor of related links HTML for specialized processing
3. 1947566 pdf2: allow XEP to be installed elsewhere
4. 2477370 Refactor of conrefImpl.xsl

31 SF Bugs Fixed

1. 2008294 End flag does not work in ditaval
2. 2078563 Keyref not working for external links
3. 2027170 maprefImpl.xsl removes part of class attr from topichead
4. 2027058 Topichead element stops map processing in ChunkMapReader
5. 2001268 DITA-OT documentation wants JDK 1.4
6. 2008317 Chunking one topic from a large file hogs memory
7. 2219479 <proinfo> subelement associations are lost in HTML output
8. 2340727 No link in TOC to topics in ditabase
9. 2317627 Chunking map to create one file should use map file name
10. 2143078 Relcolwidth not respected in choicetable
11. 1995223 bookmap: bookmeta->bookrights information is not generated

12. 1990167 reltable doesn't generate external links correctly
13. 1953553 FO 1.4.2 plugin: Inconsistent display of "on page" in xref
14. 1951879 Link generation is disabled for bookmaps
15. 1997171 topic body suppressed in chapter topics
16. 2417980 Toolkit should support both versions of Serbian lang code
17. 1947817 Extra topics appear in PDF output when using reltables
18. 2004588 Image alt text needs space normalized
19. 1954463 Conditional processing multiple PDFs in the same JVM
20. 2061301 Useless import of xslt4j stops dost.jar being built
21. 2317659 Chunking "by-topic" does not work
22. 1955211 Footnotes-by-reference rendered naively
23. 2010062 Spaces dropped in indexterm that has keyword
24. 2607496 USEINDEX=no breaks HTML Help HHP
25. 2570118 <xref> without href drops content
26. 2414891 Map contains wrong reference after chunking
27. 2010092 example or section title can be orphaned in PDF2
28. 1999117 Ubuntu 8.04 | startcmd.sh doesn't work
29. 2020317 Many duplicated id in xxx_MERGED.xml in PDF2 output.
30. 2614006 Conref processing ignores -dita-use-conref-target
31. 1727863 <groupchoice> has an extra "|" character

DITA OT release 1.4.2.1

Release 1.4.2.1 is a full build to provide an urgent fix to fix the following critical problem which users found in release 1.4.2.

SF Bugs Fixed

1. SF Bug 1956231 OSGI Manifest for eclipsehelp transtype contains spaces.

For this fix, we have updated `xsl/map2pluginImpl.xsl` to prevent generating extra spaces at the head of every line in OSGI manifest file of eclipse help output.

DITA OT release 1.4.2

Release 1.4.2 is a maintenance release to fix defects and make patches based on release 1.4.1.

Release 1.4.2 comes in three versions:

1. Minimal package contains only the core processing code.
2. Standard package contains the core processing code, plus demos, documentation, and samples.
3. Full "easy install" package contains the standard package plus external libraries useful for many toolkit processes (such as Xalan and ANT), plus a batch file to setup a local environment.

39 SF Bugs Fixed

1. 1945824 Index-see works for "ru-ru" but not for "ru"
2. 1944245 Null pointer error with complex filename
3. 1923519 Conrefs in nested, conreffed topics don't work
4. 1911285 files.txt is not up-to-date
5. 1906954 Constants.ANT_INVOKER_EXT_PARAM_OUTPUTDIR resulted in null ...
6. 1903830 Error when collection-type=sequence in map
7. 1903626 Topicref to Glossentry With Topicmeta Throws Null Pointer

8. 1900907 Documentation of generateouter command-line param is incorrect
9. 1900427 TIFF file format not a supported type
10. 1898810 Problem running ant with DITA-OT in path with Latin char
11. 1897358 Compiling CHM's in sequence results in errant index entries
12. 1894561 wordrtf not correctly handling p inside li
13. 1893234 Java TopicMerge removes processing instruction
14. 1868423 Null pointer exception when a PI is at the end of the file
15. 1857405 chunk processing fails when no section element
16. 1855047 startup.sh fails under OS X
17. 1849346 FO file generated from DITA MAP not valid
18. 1843652 Image referenced in map is not found, topicmerge breaks
19. 1843583 Extra bullet in TOC for topicref with no href or navtitle
20. 1839765 index-sort-as not used, content appears in index
21. 1908306 Index entry for external resource is ignored
22. 1908293 HTML Help index contains extra anchor
23. 1900916 Pointer to CSS is Incorrect in index.html
24. 1898451 HTML titles should be space-normalized for CHM consumption
25. 1898228 Table desc not being processed
26. 1897551 maplink is unaware of chunk to-content
27. 1893461 map file href handling
28. 1889918 Index link goes with wrong entry
29. 1883907 IndexTermReader class doesn't handle specialized titles
30. 1873401 XHTML: colsep in last column when @frame=none
31. 1872434 _merge.xml missing metadata
32. 1868047 htmlhelp path in demo ant script is fixed
33. 1864247 PIs missing from ditamaps in temp dir
34. 1857282 path to css output is not correct
35. 1848355 gen-list wants class on <foreign> descendants
36. 1843693 Bad XREF syntax gives confusing message about unique_193
37. 1841175 Need to clean up doc/ directory (remove invalid items)
38. 1832800 Empty end-range indexterm causes eclipse error
39. 1606387 Shortdesc & Abstract formatting is incorrect for XHTML

3 SF Patches Added

1. 1930220 Simplify flag templates
2. 1876118 Add plug-in support for string resource-files
3. 1818318 Path to HTML Help compiler on x64 Windows

6 SF Enhancements Added

1. 1855523 Pass dost.class.path to XSLT tasks
2. 1827322 Let plugins add their own template files
3. 1825843 Let plugins add dependencies to Ant targets
4. 1824466 Subclass ImportAction
5. 1782256 Let plugins not have to choose to go in "demo" or "plugins"
6. 1859421 Add plug-in support for string resource-files

DITA OT release 1.4.1

Release 1.4.1 is a maintenance release to fix defects and make patches based on release 1.4.

23 SF Bugs Fixed

1. 1833801 Infinite loop in MapMetaReader
2. 1833796 move-meta-entries creates invalid XML
3. 1827055 Dita 1.4 move metadata method failing
4. 1819663 XHTML processing add in output files.
5. 1815155 Using xref moves output directory
6. 1807808 Java TopicMerge calling XSLT transformer with URL not file
7. 1806728 Merge doesn't normalize filenames
8. 1806130 chunk module wraps long lines
9. 1806081 <dita> without class attribute triggers warning
10. 1803190 XHTML: processing <xref> to
11. 1803183 XHTML: and <xref> within <pre>
12. 1796207 topicmeta in ditamap causes build failure
13. 1782109 Title input to Help Compiler invalid for taskbook example
14. 1779066 [DOTX031E] Errors
15. 1770571 Chunk "to-content" on map not implemented
16. 1732678 Map without DOCTYPE declaration produces odd error
17. 1675195 No Error Location for Titleless Topic
18. 1639672 The Toolkit does not properly support valid xml:lang values.
19. 1639344 Xref : topicpull : the spectitle not used as linktext
20. 1628937 Rename supportingboth.ditaand.xmlinaditamap.dita
21. 1584187 Bookmap 1.1: <title> element breaks topicmerge
22. 1563093 Difficult to find location of error
23. 1505172 foimgext Considered Harmful

5 SF Patches Added

1. 1741302 Prevent indexterm crash with two-letter language codes
2. 1630214 HTML Help HHP generator: Language tag
3. 1498936 Failure when moving links with embedded mathml
4. 1481586 CSS for ditamap-to-HTML TOC
5. 1457541 xref to elements fails within topics in PDF

5 SF RFE Added

1. 1764910 Allow greater control over the output directory
2. 1764905 Allow option to build only topics listed in the map
3. 1725280 Improve error reporting in general
4. 1686939 Make dita.list into an XML file
5. 1676947 Integration points for passing params to XSL

DITA OT release 1.4

Release 1.4 is a major release to add new functions, fulfill new requirements, make some function enhancements and fix bugs over release 1.3.1. Available since August 1, 2007

The DITA-OT Release 1.4 contains full support for the OASIS DITA 1.1 standard. This completes the preliminary support added in the 1.3 and 1.3.1 versions of the toolkit. New and improved items for 1.1 are listed under [Improvements] below. Support for the new bookmap standard is available in the latest version of the FO plug-in, which uses the "pdf2" transform type; it will be released together with or soon after the release of DITA-OT 1.4. The deprecated "pdf" transform type has not been updated for the new bookmap. Together with DITA 1.1 support, the toolkit development team has improved error reporting so that build failures are more accurately reported at the

end of the build. Error handling will continue to improve in future releases. Release 1.4 comes in two versions. The full version contains several external packages that are useful or critical to running the toolkit, such as Xalan and the XML Catalog resolver. The smaller package contains only core toolkit code. **NOTE ABOUT DEPRECATED CODE:** changes for the new DITaval standard required a change to code in dita2htmlImpl.xml. The "flagit" named template is deprecated and will not work with the new ditaval format. Overrides to this step should be updated to use "start-flagit" and "end-flagit". The flagit template will continue to work with the old ditaval but will generate a warning for each call.

Changes

1. Release 1.4 improves the processing of DITA documents using XML Schemas. One was able to process these type of documents in Release 1.3.1 but it meant that the schema location had to have the absolute location of the schema in order for the Toolkit properly.

DITA 1.1 introduces the use of URNs to normatively identify the schemas used for validation. The URNs have the following desing pattern "urn:oasis:names:tc:dita:xsd:<schemaDocument>:1.1". You should use these in as the value for the attribute xsi:noNamespaceSchemaLocation.

13 Improvements

1. Support <title> in map
2. Ignore Index-base in default processing
3. Retrieve the link text from abstract element.
4. Format shortdesc in abstract appropriately
5. Add standard code to allow overrides to easily process generalized version of unknown and foreign element
6. Support @dir on every element
7. Refactor mapref resolution
8. Support generalization and re-specialization of unknown/foreign elements
9. Replace Move Index module with new Move Metadata module
10. New DITaval standard support
11. New chunk attribute support
12. Support XML Schema validated instance document processing using XML Catalogs

17 SF Bugs Fixed

1. 1700561 Null Pointer Exception on Missing domain= Attribute
2. 1733264 pretty.xml is broken
3. 1619074 table in step screws up following steps for HTML generation
4. 1728700 GenMapAndTopicList keeps filtering when called a second time
5. 1732562 DitaWriter.java can duplicate @xtrf and @xtrc
6. 1733108 Update Bookmap sample files to DITA 1.1
7. 1706263 Conrefing from a map to topic is not working properly
8. 1677620 Non-DITA file is treated as DITA in pre-process
9. 1717471 Links show up more than once
10. 1712543 gen-list-without-flagging : NullPointerException
11. 1652892 Invalid hdr/fttr arg value causes build failure
12. 1647950 PIs in DITA source are dropped in the processing pipeline
13. 1644559 Force Toolkit to use private catalog to allow schemas to work properly
14. 1642138 Move javamerge target out of build_template.xml
15. 1643155 Map TOC is HTML even for transtype="xhtml"
16. 1637564 topicpull breaks specializations of xref
17. 1676968 Plugins adding to classpath break when basedir != dita.dir

DITA OT release 1.3.1

Release 1.3.1 is a maintenance release to fix defects and make patches based on release 1.3.

15 SF Bugs Fixed

1. SF Bug 1385642 docbook/topic2db.xml - shortdesc
2. SF Bug 1528638 wordrtf does not correctly number steps
3. SF Bug 1562518 Flag is confusing when a list is mixed with text
4. SF Bug 1563665 Should use CSS to honor rowsep and colsep in table entries
5. SF Bug 1567117 Xref to footnote is not resolved correctly
6. SF Bug 1569671 <reltable> in nested map creates bogus TOC entries
7. SF Bug 1573996 Plugins do not work in plugins directory
8. SF Bug 1574011 Spaces in a file name prevent XHTML output
9. SF Bug 1584186 Bookmap 1.1: <title> element duplicated in mappull
10. SF Bug 1588039 Conref domain checking is sub-par
11. SF Bug 1588624 OT v1.3 map2hhc.xml error
12. SF Bug 1597444 Java topicmerge breaks when text contains less-than
13. SF Bug 1597473 Nothing references common.css
14. SF Bug 1598109 Java topicmerge does not rewrite image/@href
15. SF Bug 1598230 jhindexer of JavaHelp breaks Search Index for DITAOT content

DITA OT release 1.3

OASIS DITA 1.1 support

Things to know about OASIS DITA 1.1 support in this release:

1. DITA-OT 1.3 provides preliminary processing support for the upcoming OASIS DITA 1.1 specification (see http://wiki.oasis-open.org/dita/Roadmap_for_DITA_development). Because the proposed OASIS DITA 1.1 DTDs and Schemas are fully backwards compatible with the latest DITA 1.0.1 DTDs and Schemas, the 1.3 Toolkit provides the proposed 1.1 materials as the default DTDs for processing. The XML Catalog resolution maps any references for DITA 1.0 doctypes to the 1.1 DTDs, for example. All processing ordinarily dependent on the 1.0 definition continues to work as usual, and any documents that make use of the newer 1.1-based elements or attributes will be supported with specific new processing function (such as base support for the new <data> element). *Documents created with the proposed OASIS DITA 1.1 DTDs are the only ones ever likely to have features that invoke the specific new 1.1-based processing support.*



Important: Because this support is based on a yet-to-be-approved version of the proposed OASIS DITA 1.1 specification, if you choose to investigate any 1.1-based function, be aware that the 1.1 implementation in this version of the Toolkit is preliminary and very much forward-looking. Upon final approval of the DITA 1.1 standard, Toolkit developers will, of course, review our implementation to make certain that it conforms to the defined level of reference implementation.

2. Related to the DITA 1.1 preliminary implementation, the much-discussed bookmap updates for DITA 1.1 will be provided as override capabilities for the FO plugin (Idiom's donation). Note that:
 - The FO demo transform code at the 1.2.2 level is still included in the DITA 1.3 package, but is now deprecated.
 - To get the FO updates for 1.3, grab the FO plug-in at its next update, which should be shortly after the 1.3 core Toolkit code is released.
 - The updated FO plug-in will be usable with FOP as well as with XEP.

Changes

The DITA Open Toolkit team understands the need for stability in essential APIs in the Toolkit. This version of the toolkit provides some strategic updates that correct some long-overdue faults in the original implementation. Necessarily, there are some changes to note:

1. **Change to build.xml:** To make the DITA processing environment more like other Ant-driven build environments, the original build.xml has been renamed as build_demo.xml. The current build.xml in this release is now the normal ANT script entrance for starting a transformation. If you have created Ant tasks that tried to work around the former build.xml architecture, those might need to be revised to take advantage of the separated function.
2. **Change to command line invocations:** The "Ant refactoring" exercise for this release has changed some previously documented Ant calls for running demos. This change enables better use of the Ant modules for power users who need to integrate the Toolkit into programming build environments such as Eclipse, but the change affects some documentation. This is a permanent change that should remain stable from now on. Wherever you see an older instruction like "c:\dita-ot>ant all", you now need to indicate the component that contains the demos, so you would type "c:\dita-ot>ant all -f build_demo.xml".
3. **Separation of demo targets from formal component targets:** Another effect of the Ant refactoring is that the internal programming targets will now be displayed when you type "ant -p". To see both those programmings targets and the demos that are part of this component, type "c:\dita-ot>ant -p -f build_demo.xml". To run just one of the demos that you see in the resulting list, dita.faq for example, type "c:\dita-ot>ant dita.faq -f build_demo.xml".
4. **Classpath update to enable catalog resolver:** This release now includes the Apache catalog resolver for improved lookup of DTDs by any of the Toolkit components. The fullpackage version of the Toolkit sets up these variables for each session. For the regular (smaller) version of the Toolkit, you need to include lib and lib\resource\resolver.jar into your classpath. For example if your CLASSPATH is like:

```
c:\dita-ot\lib\dost.jar
```

you need to change it to:

```
c:\dita-ot\lib;c:\dita-ot\lib\dost.jar;c:\dita-ot\lib\resolver.jar
```

At any time, the full version can be used like a normal installation as long as you update the system variables either in the environment settings or in a batch file that sets up the shell environment.

5. **License bundling:** To reduce the duplication of builds on Sourceforge in which the only difference was the license provided in each, both the Apache and CPL licenses are included in root directory of the Toolkit. Use the one that applies to your situation.
6. **Two install options:** Two download versions are now offered. The smaller one is for updating existing installations or for reuse in embedded applications that already provide the other processing components--business as usual. A new package with "fullpackage" in the name now incorporates the essential processing modules to create a processing environment for new users and evaluators that requires nothing more than to unzip the file into an appropriate directory and then click on a "start" batch file. A new document in its root directory (an output of doc/EvaluateOT.dita, "Evaluating the DITA Open Toolkit (fullpackage version)") informs new users how to install and use the Toolkit for the first time.
7. **Other enhancements:** The public design discussions that fed into the final selection and architectures for this release are documented at the DITA Focus Area in a topic called "DITA OT 1.3 Issues tracking" (<http://dita.xml.org/node/1282>).

7 Improvements

1. Preliminary support for OASIS DITA 1.1
2. Support ICU in index sorting
3. Integrate with Eclipse
4. Refactor Ant script for easy override
5. Topicmerge reimplement in JAVA
6. Enable XML Catalog Resolver
7. Full package distribution (was GUI/usability)

21 SourceForge Bugs Fixed

1. SF Bug 1582506 Docbook cannot handle <author>
2. SF Bug 1548189 Sections should not jump to <h4> for Accessibility reasons
3. SF Bug 1548180 Spaces dropped from index terms
4. SF Bug 1548154 XHTML index links should go to the topic
5. SF Bug 1545038 CommandLineInvoker is unfriendly towards spaces
6. SF Bug 1541055 topicref @id incorrectly uses NMTOKEN type
7. SF Bug 1530443 dost.jar relies on the incorrect behavior of Xerces
8. SF Bug 1473029 Syntax code makes overrides difficult
9. SF Bug 1470101 Metadata in topics is left out of XHTML headers
10. SF Bug 1470077 Choicetable headers create attribute inside attribute
11. SF Bug 1470057 Step template creates attributes after creating tags
12. SF Bug 1465947 <topichead> without children the whole branch to disappear
13. SF Bug 1465941 Keywords defined in map are ignored if <topicref> contains t
14. SF Bug 1465866 Problems in catalog-dita.txt
15. SF Bug 1460447 <morerows> not well supported in pdf transformation.
16. SF Bug 1457187 'copy-to' doesn't actually copy files
17. SF Bug 1454835 OT renders files referenced via conref only
18. SF Bug 1427808 Should be easier to modify link attributes in XHTML
19. SF Bug 1422182 @colname renaming needs to apply to @namestart and @nameend
20. SF Bug 1417820 fo and docbook outputs can't handle deep topic dirs
21. SF Bug 1368997 PDF Vertical list of author redundancy

1 SourceForge Patch Added

1. SF Patch 1503296 Refactor of HTMLHelp infiles creation

1 SourceForge RFE Added

1. SF RFE 1160960 Enh: Toolkit should work with both *.dita and *.xml

DITA OT release 1.2.2

Release 1.2.2 is a maintenance release to fix defects and make patches based on release 1.2.1.

Improvements

1. Chinese support in WORD RTF
2. Improve plug-in architecture in plug-in dependency handling

SourceForge Changes

1. SF Bug 1461642 Relative paths in toolkit.
2. SF Bug 1463756 TROFF output is not usable
3. SF Bug 1459527 Properties elements should generate default headings
4. SF Bug 1457552 FO gen-toc does not work right for ditamaps and bookmaps
5. SF Bug 1430983 Specialized indexterm does not generate entries in index
6. SF Bug 1363055 Shortdesc disappears when optional body is removed
7. SF Bug 1368403 The dita2docbook transformation lacks support for args.xml
8. SF Bug 1405184 Note template for XHTML should be easier to override
9. SF Bug 1407646 Map titles are not used in print outputs
10. SF Bug 1409960 No page numbers in PDF toc

11. SF Bug 1459790 Related Links omitted when map references file#topicid
12. SF Bug 1428015 Topicmerge.xsl should leave indentation alone
13. SF Bug 1429400 FO output should allow more external links
14. SF Bug 1405169 Space inside XHTML note title affects CSS presentation
15. SF Bug 1402377 Updated translations for Icelandic
16. SF Bug 1366845 XRefs do not generate page numbers
17. SF Patch 1326450 Make \${basedir} mine
18. SF Patch 1328264 FOP task userconfig file
19. SF Patch 1385636 Tweaks to docbook/topic2db.xsl
20. SF Patch 1435584 Recognize more image extensions
21. SF Patch 1444900 Add template for getting input file URI
22. SF Patch 1460419 Add a new parameter /cssroot:{args.cssroot}
23. SF Patch 1460441 map2hhp [FILES] include
24. SF RFE 1400140 Add a new parameter /cssroot:{args.cssroot}

DITA OT release 1.2.1

Release 1.2.1 is a maintenance release to fix defects and make patches based on release 1.2.

Improvements

1. Corrupt table generated in WORD RTF is fixed
2. Pictures are merged into the WORD RTF instead of creating links to them
3. lq element is supported in WORD RTF
4. Generated text can be translated to different languages in WORD RTF
5. In WORD RTF, if no <choptionhd> given, head will be generated in table

SourceForge Changes

1. SF Bug 1460451 Spaces preserving methods are different among tags.
2. SF Bug 1460449 Nested list can not be well supported.
3. SF Bug 1460445 h2d stylesheet cannot handle HTML files within namespace.
4. SF Bug 1431229 hardcoded path in MessageUtils.java
5. SF Bug 1408477 <desc> element is not handled inside xref for XHTML
6. SF Bug 1398867 ampersands in hrefs (on xref and link) cause build to fail
7. SF Bug 1326439 filtered-out indexterms leak into index through dita.list
8. SF Bug 1408487 Short description is not retrieved for <xref> element
9. SF Bug 1407454 XHTML processing for <alt> is incomplete
10. SF Bug 1405221 Some table frames ignored in dita->xhtml
11. SF Bug 1414398 Cannot set provider for Eclipse help transformation
12. SF RFE 1448712 add support for /plugins directory in plug-in architecture

DITA OT release 1.2

DITA open toolkit Release 1.2 is a major release to add new functions, fulfill new requirements, make some function enhancements and fix bugs over release 1.1.2.1.

Important Change

DITA-OT 1.2 offers new error handling and logging system. If you invoke your transformation by using java command line where new error handling and logging system is mandatory, you need to set the *CLASSPATH* Environment Variable for `dost.jar`. If you invoke your transformation by using an ant script, you need to do one

more step after the setting above. That is adding a parameter in your command to invoke an ant script. For example, use `ant -f ant\sample_xhtml.xml -logger org.dita.dost.log.DITAOTBuildLogger` instead of `ant -f ant\sample_xhtml.xml` to start a transformation defined in ant script file `ant\sample_xhtml.xml`.

New Functions

1. New plugin architecture

DITA Open Toolkit 1.2 provides a new function to help users to download, install and use plug-ins and help developers create new plug-ins for DITA Open Toolkit.

2. Transformation to wordrtf

DITA Open Toolkit 1.2 provides DITA to Word transforming function to transform DITA source files to output in Microsoft® Word RTF file.

3. HTML to DITA migration tool

DITA Open Toolkit 1.2 provides a HTML to DITA migration tool, which migrates HTML files to DITA files. This migration tool originally comes from the developerWorks publication of Robert D. Anderson's how-to articles with the original h2d code.

4. Problem determination and log analysis

In DITA Open Toolkit 1.2, a new logging method is supported to log messages both on the screen and into the log file. The messages on the screen present user with the status information, warning, error, and fatal error messages. The messages in the log file present user with more detailed information about the transformation process. By analyzing these messages, user can know what cause the problem and how to solve it.

5. Open DITA User Guide for conditional processing

In DITA Open Toolkit 1.2, a new user guide which can help users to use conditional processing is added to toolkit document.

6. Include the OASIS version langref

In DITA Open Toolkit 1.2, a new OASIS version of language reference for DITA standard is added to toolkit document.

7. Document adapt to OASIS DITA 1.0.1 DTDs

DITA DTD files are updated to 1.0.1 version in DITA Open Toolkit 1.2.

Other Changes

1. SF Bug 1304545 Some folders were copied to DITA-OT's root directory
2. SF Bug 1328689 Stylesheet links in HTML emitted with local filesystem paths
3. SF Bug 1333481 Mapref function does not work for maps in another directory
4. SF Bug 1343963 Blank index.html generated for ditamap contains only reltabe
5. SF Bug 1344486 java.io.EOFException thrown out when reading ditaval file
6. SF Bug 1347669 Path Spec. in nested DITA maps
7. SF Bug 1357139 filtering behavior doesn't conform to spec
8. SF Bug 1358619 The property.temp file gets cleaned out by default
9. SF Bug 1366843 XRefs do not generate proper links in FO/PDF
10. SF Bug 1367636 dita2fo-elems.xsl has strange line breaks
11. SF RFE 1296133 Enable related-links in PDF output
12. SF RFE 1326377 Add a /dbg or /debug flag for diagnostic info
13. SF RFE 1331727 Toolkit need to run on JDK 1.5.x(only support to run under Sun JDK 1.5 with saxon in normal case)
14. SF RFE 1357054 Be more friendly towards relative directories
15. SF RFE 1357906 Provide a default output directory
16. SF RFE 1368073 Enable plugins for DITA open toolkit
17. SF RFE 1379518 Clearer error messages and improved exception handling

18. SF RFE 1379523 DITA to Rich Text Format (.rtf) file

19. SF RFE 1382482 plugin architecture of DITA-OT

DITA OT release 1.1.2.1

Release 1.1.2.1 is a full build to provide an urgent fix to fix the following critical problem which users found in release 1.1.2.

- SF Bug 1345600 The build process failed when run "Ant all" in release 1.1.2

For this fix, we have restored all the source DITA files in 'doc' and directories in the binary packages.

Note that the original parameter "args.eclipse.toc" in "Ant tasks and script" was separated to "args.eclipsehelp.toc" for DITA-to-Eclipse help transformation, and "args.eclipsecontent.toc" for DITA-to-dynamic Eclipse content transformation.

Another issue is that we found there is a mismatch in the document and the toolkit behavior when you are trying to use the following command

```
ant -f conductor.xml -propertyfile ${dita.temp.dir}/property.temp.
```

Now we have updated the documentation. Please refer to the topic 'Building DITA output with Java command line' on our website for more details.

These updates do not affect standard operation of the toolkit. The main goal of this minor release to enable new users of the toolkit to run the installation verification tests without failure.

DITA OT release 1.1.2

Release 1.1.2 is a maintenance release to fix defects and make patches based on release 1.1.1.

But there are certain limitations and unfixed bugs in this release, such as,

- Bug 1343963 Blank index.html generated for ditamap contains only reltabe
- Bug 1344486 java.io.EOFException thrown out when reading ditaval file

Please check the current 'open' bugs on the SourceForge bugs tracker.

Changes

1. SF Bug 1297355: Multilevel HTML Help popup shows filenames
2. SF Bug 1297657: Update for Supported Parameters page
3. SF Bug 1304859: Toolkit disallows repetition of topic ID within map
4. SF Bug 1306361: Fatal error in published ditamap example
5. SF Bug 1306363: common.css not compiled with htmlhelp
6. SF Bug 1311788: DTD references not resolved
7. SF Bug 1314081: Fix catalog entries in catalag-ant.xml for OASIS DTDs
8. SF Bug 1323435: wrong system id for html output used in validation
9. SF Bug 1323486: HTML Help subterm indexes not sorted
10. SF Bug 1325290: JavaHelp output does not work for Russian
11. SF Bug 1325277: File missing from the map causes abend
12. SF Patch 1253783: dita2fo-links relative hrefs
13. SF Patch 1324387: In xslfo, groupchoice var prints extra | delimiter
14. SF RFE 1324990: Installation Guide

Parameter Changes

1. The original parameter "args.eclipse.toc" in "Ant tasks and script" was separated to "args.eclipsehelp.toc" for dita2eclipsehelp transformation, and "args.eclipsecontent.toc" for dita2eclipsecontent transformation.
2. Several parameters were added to the java command line interface, including "/javahelptoc", "/javahelpmap", "/eclipsehelptoc", "/eclipsecontenttoc", "/xhtmltoc".

Other Changes

Change to the "doc" directory, except "doc\langref" directory:

1. The source dita files and the generated HTML, CHM, and PDF files were separated into separate downloads.
2. The source package contains the source dita files.
3. The binary package contains the generated HTML, CHM, and PDF files.

DITA OT release 1.1.1

Release 1.1.1 is a maintenance release to fix defects and make patches based on release 1.1.

For patch 1284023, we are changing the name of the jar lib file from dost1.0.jar back to dost.jar because we believe we need to keep the jar file name consistent through various releases.

Changes

1. SF Bug 1196409: HTMLHelp output does not reference CSS
2. SF Bug 1272687: extra "../" link generated by topicgroup
3. SF Bug 1273751: revision flag using unavailable pictures
4. SF Bug 1273816: Index generation doesn't cope with multilevel well
5. SF Bug 1281900: Unnecessary comment and href typo
6. SF Bug 1283600: unnecessary space in document cause invalid parameter of Ant
7. SF Bug 1283644: multipul document(\$FILTERFILE,/) doesn't work (XALAN)
8. SF Patch 1251609: pretargets xsl directory needs to use \${ dita.script.dir }
9. SF Patch 1252441: Files in temp directory not deleted before build
10. SF Patch 1253785: Inline images in dita2fo-elems
11. SF Patch 1284023: change the name of jar file and remove the version name

DITA OT release 1.1

Release 1.1 is a major release to add new functions, fulfill new requirements, make some function enhancements and fix bugs over release 1.0.2.

1. Adaptation to the new OASIS DITA standard

Release 1.1 implements the new OASIS DITA 1.0 standard for DITA DTDs and Schemas.

DTDs of the previous release locate in the directory **dtd/dita132** and schemas of the previous release locate in the directory **schema/dita132**.

2. Transformation to troff

Release 1.1 supports new troff output. Troff output looks like Linux man page output.

3. XML catalog support

An XML catalog, which can consist of several catalog entry files, is a logical structure that describes mapping information between public IDs and URLs of DTD files. A catalog entry file is an XML file that includes a group of catalog entries. If you want to know more about XML catalog, please refer [XML Catalog](#).

A catalog entry can be used to locate a unified resource identifier (URI) reference for a certain resource such as a DTD file. An external entity's public identifier is used for mapping to the URI reference. The URI of any system identifier can be ignored.

4. Topicref referring to a nested topic

The href attribute of the topicref is extended to quote a nested topic in a dita file.

For example, in previous releases, href attribute is set like: href = "xxx.dita"; in release 1.1, href attribute can be set like: href = "xxx.dita#abc.dita".

5. Globalization support

Release 1.1 supports over 20 popular languages within the content of dita files. And it also provides translation function for DITA keywords to over 20 languages. Currently this globalization support fully applies to Eclipse Help and XHTML transformations, and partially applies to other transformations.

6. Accessibility support

Accessibility support is now partially applies to PDF and XHTML transformations.

7. Eclipse Content Provider Support

Please refer to [Eclipse Content Provider](#) for detail information.

8. Index information in output

The output of HTML Help and Java Help transformations contain index information now.

9. Mapref function

Mapref refers to a special usage of the <topicref> element as a reference to another ditamap file. This allows you to manage the overall ditamap file more easily. A large ditamap file can thus be broken down into several ditamap files, making it easier for the user to manage the overall logical structure. On the other hand, this mechanism also increases the reusability of those ditamap files. If you want to know more about mapref, please refer [Mapref](#).

10. TOC generation for Eclipse Help transformation

TOC generation now supported in transformation to Eclipse Help. Eclipse.

11. Helpset generation for Java Help transformation

Helpset generation now supported in transformation to Java Help.

12. New parameters supported in Java commands

In Java commands: /indexshow, /outext, /copycss, /xsl, /tempdir.

13. New parameters supported in Ant scripts

In Ant scripts: args.indexshow, args.outext, args.copycss, args.xsl, dita.temp.dir

Other Changes

1. SF bug 1220569: Add XML Schema processing to DITA-OT
2. SF bug 1220644: Prompted ant--image does not link for single topic to PDF
3. SF bug 1229058: Add schema validation loading file for processing
4. SF RFE 1176855: Ant must be run from toolkit directory
5. SF RFE 1183482: Copy pre-existing html to output dir
6. SF RFE 1183490: Provide argument to specify the location of temp dir
7. SF RFE 1201242: override capability

DITA OT release 1.0.2

Release 1.0.2 is a maintenance release to fix defects and adds some minor enhancements in release 1.0.1.

Changes

1. SF Bug 1181950: format attribute should be set to 'dita' for dita topic
2. SF RFE 1183487: Document the usage of footer property
3. SF RFE 1198847: command line interface support
4. SF RFE 1198850: architecture document update
5. SF RFE 1200410: need explanation for dita.list
6. SF RFE 1201175: XML catalog support
7. SF Patch 1176909: Add template for getting image URI

DITA OT release 1.0.1

Release 1.0.1 is a maintenance release to fix defects and adds some minor enhancements in release 1.0.

Changes

1. Committer: maplink.xsl doesn't generate related links for second level referred topic
2. Committer: avoid infinite loop of conref
3. SF Bug 1160964: Can't point above the directory which contains the map file
4. SF Bug 1163523: Broken XPath expression in mappull.xsl
5. SF Bug 1168974: useless DRAFT param in FO transformation
6. SF Bug 1173162: generate null internal link destination in fo transformation
7. SF Bug 1173164: Not correctly use document() in dita2fo-links.xsl
8. SF Bug 1173663: All base directories are DITA-OT 1.0
9. SF Patch 1163561: XLST match patterns test for element names
10. SF Patch 1165068: FO hyperlinks and FOP-generated PDF bookmarks
11. SF Patch 1174012: Modification to sequence.ditamap

DITA OT release 1.0

The initial release of the Open Sourced DITA Toolkit introduces major architectural changes from the previous, developerWorks version of the Toolkit.

New features

1. A new, Java-based processing architecture that supports single-threaded execution throughout.
2. Ant-based orchestration of the processing environment, from preprocessing to transformation to any required post-processing.
3. A pre-processor core that supports conditional processing and conref resolution.
4. Map-driven processing that generates links for transformed topics.
5. A new DITA to HTML transform that replaces the previous topic2html_Impl.xsl core transform. This new core is based on requirements for high-volume usage within IBM for the past several years.

Ant-driven processing means that you can integrate the DITA processing tools into a seamless pipeline within supportive environments such as Eclipse.

The DTDs and Schemas in this version are based on those in the previous dita132 package with bug fixes. The DITA OS Toolkit will later support the OASIS 1.0 specification in its public review form.

DITA history on developerWorks (pre-Open Source)

Versions of the toolkit prior to Open Source are in the developerWorks XML Zone at this address: [DITA Downloads](#)
Change logs for those versions are within the Readme files in each distribution.

Appendix

B

Project Management

Topics:

- *Goals and objectives of the DITA Open Toolkit*
- *DITA Open Toolkit Development Process*
- *DITA Open Toolkit Contribution Policy*
- *DITA-OT Contribution Questionnaire Form 1.2*
- *Due Diligence for Submission of Bug Fixes and Patches from Non-Committers*

Goals and objectives of the DITA Open Toolkit

The long term goal of the DITA Open Toolkit is to provide a high quality implementation for production level output of DITA XML content, built in a professionally-managed project environment by vetted contributors, and tested thoroughly for each release.

The DITA-OT may be used in a variety of ways:

- It may be integrated as one component in a larger end-to-end production pipeline. Either the minimal or standard package may be the most suitable for this use.
- It may be used on its own as a complete DITA production system. Users looking for this will most likely use the Full Easy Install package.
- It may be integrated into vendor tools such as editors or content management systems, to provide an easy DITA production system. The minimal package is designed for this type of use.

Project objectives

The project will be managed similarly to any commercial software development project, with requirements gathering, plan validation with stakeholders, scheduled activities, tests, reviews, and builds, with an emphasis on quality.

The project will emplace a process for tracking all contributions. Any individual or any organization can contribute, but the spirit of the community is personal contribution. IBM as Committer is desirous that all major contributors get appropriate recognition in press releases, etc..

The DITA Open Toolkit project will track and fix bugs reported against its products, and issue patches based on urgency.

The DITA Open Toolkit project will track to major version updates of the DTDs/Schemas by the OASIS DITA TC.

The DITA Open Toolkit project will track to bug fixes of the DTDs/Schemas by the OASIS DITA TC as necessary (such fixes tend to have little impact to tools).

The project agrees with the Open Source motto of "Release early and often" to get wide consensus on issues. The project works with an Agile development process, releasing test builds approximately every three weeks, and encourages feedback on test builds while function is being development.

DITA Open Toolkit Development Process

The DITA Open Toolkit development process is modelled after other popular and successful Open Source projects, notably the Eclipse development process (for definitions and process statements).

Revision 1.0

Last updated on February 27, 2005

Purpose and scope

This document describes the development process of DITA Open Toolkit (DITA-OT) project.

Roles and Responsibilities

There are several roles in the DITA Open Toolkit open source project. Each has different rights and obligations.

Project Manager (PM)

An individual responsible for managing this open source project. The PM is expected to ensure that

- All Projects operate effectively by providing leadership to guide this project's overall direction

and by removing obstacles, solving problems, and resolving conflicts.

- All Project plans, technical documents and reports are publicly available.
- All Projects operate using open source rules of engagement: meritocracy, transparency, and open participation. These principles work together. Anyone can participate in this project. This open interaction, from answering questions to reporting bugs to making code contributions to creating designs, enables everyone to recognize and utilize the contributions.

It is anticipated that the PM scope may be enlarged to head of a leadership committee as the project grows.

Committer

A developer who is expected to influence the Project's development and has the write access to the source code repository. This position reflects a track record of high quality contributions. At the initiation of this project, the PM nominates one Committer who will oversee the quality and originality of all contributions.

Contributor

An individual who contributes code, fixes, tests, documentation, or other work that is part of this project. A Contributor does not have write access to the source code repository.

Process life cycle

To ensure transparency and predictability, this project sequences through a series of Phases, with well-defined Phase Transition Reviews:

1. Plan phase

Plan phase is mainly to determine what to implement in each release. The Project Manager will collect requirements and feature requests (enhancements) from all the members at large. Then all the requirements will be prioritized. After discussion with the Committer(s) and other developers, top requirements will be put into. The schedule of this release and planned release dates will also be determined. Exit criteria for this release will be set to ensure high quality of all the releases.

2. Design phase

For all the requirements determined to be implemented in the current release, developers will perform high level design and create a prototype if necessary.

3. Implementation phase

Developers will implement the requirements for this release and perform testing before formal release. The Committer(s) will cooperate with contributors to ensure high quality of the code before implementation phase ends.

4. Release

Source and binary code of this release will be put into SourceForge and the next release planning phase begins.

Procedures

Manage requirements/features

Throughout the project lifecycle, requirements or features will be collected using the SourceForge requirements tracking tool. The PM will have periodic reviews with the Committer(s), Contributors, and interested parties to

assess the existing requirements, prioritize them and make decisions. Usually some requirements will be marked as candidates for next release and will be reviewed in the planning phase for next release (or in some cases for interim updates).

Fix bugs

Anyone can submit bug reports in SourceForge bug tracking system. The Committer will determine the owner of the relevant components and assign the bug for validation and disposition. Bugs have different severities. Higher severity bugs will be responded to first, usually worked to closure within one week.

Contribute code/bug fixes

Contributors can submit new code and bug fixes using the SourceForge facilities. The Committer(s) who owns the relevant components will first do due diligence to check code originality and licensing according to the [Contribution Policy](#) document. After due diligence, the Committer(s) will use his/her own judgment on whether to accept the code or bug fixes and merge them back to the original code base. Usually within a week after submitting the contribution, contributors will receive a response as to whether the contribution will be accepted or if more time is needed to check the contribution.

DITA Open Toolkit Contribution Policy

The purpose of the DITA Open Toolkit Contribution Policy is to set forth the general principles under which the DITA Open Toolkit project shall accept contributions, license contributions, license materials owned by this project, and manage other intellectual property matters.

Overview

Currently, IBM is the sole Committer for the DITA Open Toolkit project. The [Common Public License](#) (CPL) and [Apache License 2.0](#), which are incorporated herein by reference, will serve as the primary licenses under which the Committer will accept contributions of software, documentation, information (including, but not limited to, ideas, concepts, know-how and techniques) and/or other materials (collectively "Content") to the project from Contributors. A copy of the CPL and Apache License 2.0 can be found at the root directory of the DITA Open Toolkit deliverable package.

This Contribution Policy should at all times be interpreted in a manner that is consistent with the Purposes of the this project as set forth in the [DITA Open Toolkit Development Process](#) goals and objectives. This Contribution Policy shall serve as the basis for how non-Committers interact with this project through participation in this project, web-sites owned, controlled, published and/or managed under the auspices of the this project, or otherwise.

The Common Public License and Apache License 2.0 shall serve as the primary licenses under which the Committer(s) shall accept software, documentation, information (including, but not limited to, ideas, concepts, know-how and techniques) and/or other materials (collectively "Content") from contributors including, but not limited to, Contributors and Committers.

The DITA Open Toolkit project provides a process for accepting bug fixes and contributions from parties who have not accepted the license to be Contributors. See [Due Diligence for Submission of Bug Fixes and Patches from Non-Committers](#)

DUE DILIGENCE AND RECORD KEEPING

The Committer(s), shall be responsible for scrutinizing all Content contributed to the DITA Open Toolkit project and help ensure that the Contribution Policy licensing requirements set forth above are met. Except as set forth below, the applicable Committer shall conduct the following activities prior to uploading any Content into the repository or otherwise making the Content available for distribution:

1. Contact the potential contributor of the Content through an appropriate channel of communication and collect/confirm the following:
 - Contributor's name, current address, phone number and e-mail address;

- Name and contact information of the contributor's current employer, if any;
 - If the contributor is not self-employed, the Committer must request and receive a signed [consent form](#) (to be provided by the Committer) from the contributor's employer confirming that the employer does not object to the employee contributing the Content.
 - Determine if the Content can be contributed under the terms of the CPL and Apache License 2.0 or the alternative terms and conditions supplied by the Contributor. This can be done by asking the contributor questions such as;
 1. Did you develop all of the Content from scratch;
 2. If not, what materials did you use to develop the Content?
 3. Did you reference any confidential information of any third party?
 4. If you referenced third party materials, under what terms did you receive such materials?
 - If it is determined by the Committer that the Content is not the original work of the Contributor, collect the contact information of the copyright holder of the original or underlying work. The copyright holder of the Content or the underlying work may then need to be contacted to collect additional information.
2. The Contributor(s) shall document all information requested in (1) above and fill in Contribution Questionnaire (to be provided by the Committer) and provide the completed Contribution Questionnaire to the Committer.
 3. The Committer shall also be responsible for running a scan tool to help ensure that the Content does not include any code not identified by the contributor.
 4. Based on the information collected, the Committer shall use his/her reasonable judgment to determine if the Content can be contributed under terms and conditions that are consistent with the licensing requirements of this IP Policy.

If the applicable Committer has any doubts about the ability to distribute the Content under terms and conditions that are consistent with the CPL and Apache License 2.0 or the proposed alternative terms and conditions, the Committer may not upload the code to the repository or otherwise distribute the Content. The Committer(s) shall be responsible for filing/maintaining the information collected for future reference as needed.

The above record keeping requirements shall not apply to

- Minor modifications to Content previously contributed to and accepted by the Committer(s).
- Articles and White Papers
- Information or minor Content modifications provided through bug reports, mailing lists and news groups

While the record keeping requirements do not apply to the items listed above, Committers must conduct reasonable due diligence to satisfy themselves that proposed Contributions can be licensed under the terms of the CPL and Apache License 2.0.

DITA-OT Contribution Questionnaire Form 1.2

The Contribution Questionnaire is the first step in initiating the due diligence and approval process by the Project Manager (PM) for any significant contribution of content to be committed to the project. Prior to completing this Questionnaire, the Committer should have technical agreement from the PM that the new code is required. Once the PM has approved, the Committer, with the assistance of one or more of the contributors, may begin the due diligence and approval process by completing and submitting this questionnaire.

What is meant by a significant contribution?

Any initial code contribution used to kick off a new project. By definition, this code has been written elsewhere and it needs to be reviewed.

or

Any contribution authored by someone other than a committer which is adding new functionality to the codebase. In most cases, bug fixes do not add new functionality although it's not impossible.

or

Any contribution containing third-party code maintained by another open source project, individual, group, or organization. In addition to reviewing the contribution, if the license is not the Common Public License (CPL) or Apache License 2.0, the PM will need to review and approve the third-party license for compatibility with the CPL or Apache License 2.0.

How to send PM this form?

Please fill in this form and sign your name and get your employer's authorized signature, such as your manager's. then fax to DITA open toolkit Project Manager Lian, Li. The fax number is +8621-53060504. If you have problem sending international fax, please send a scanned copy to leelix@users.sourceforge.net

NOTE: A questionnaire and approval is not required for bug fixes or minor enhancements. If you have any questions, please send an email to the PM.

Your Info

Please provide your contact details:

Name:	
Organization:	
Address:	
Phone Number:	
E-mail:	

Committer

Please provide contact details for the committer who will be incorporating this contribution into the code base. If this is the same as above, just put "same" in the Name field.

Name:	
Organization:	
Address:	
Phone Number:	
E-mail:	

PM Approval

PM Approval is required for all significant contributions. Please provide the contact info of the PM who has given approval for this contribution:

Name:	
Phone Number:	
E-mail:	

Contribution

Please provide details about the contribution:

Component/Module (if known):	
Contribution Name:	
Contribution Version:	

Contribution Size (in lines of code):	
Contribution Description:	
Does this contribution require any packages maintained by a 3rd party?	
Please list all pkgs required by the contribution which are maintained by a 3rd party: (Please list one package per line e.g 3rd party package name v1.0)	
Supporting Information:	
Do you agree to distribute the Contribution under Common Public License 1.0?	
Do you agree to distribute the Contribution under Apache License 2.0?	
Provide any additional information you may have regarding intellectual property rights (patents, trademarks, etc.) related to the Contribution. If there is more than one committer who worked on this contribution, please list their name and email addresses.	

Contributor

Note: All of the contributors should ensure that they possess the necessary rights to make the contribution under the terms and conditions set out in the [Contribution Policy](#).

Please provide contact details for the contributor or the primary contributor if there is more than one:

Name:	
Organization:	
Phone Number:	
E-mail:	
% of content authored in the contribution:	

If there are other contributors, please provide names, organizations, e-mail, and percentage of content authored in the contribution:

Other Contributors:	
---------------------	--

Cryptography

If the contribution deals in any way with cryptography, please provide details:

Details:	
Identify the Cryptography algorithm used:	

Contributor's signature

Name (Type or Print)	
Title	
Signature	
Date	

Contributor employer's signature

Name (Type or Print)	
Title	
Signature	
Date	

Due Diligence for Submission of Bug Fixes and Patches from Non-Committers

Before committing code from a bug fix or patch provided by a third party who has not signed a current Common Public License to become a Contributor to the project, the IBM committer should ask the following questions and follow up as appropriate in order to ensure that the code can be contributed to the project:

- What is your name and who is your employer?
- Did you write the code that you wish to contribute to the DITA Open Toolkit project? (If the contributor says no, do not submit it as a “Contribution” to the project. You may ask the contributor to identify the complete details of the code’s source and of any licenses or restrictions applicable to the code, but you should conspicuously mark the work as “Submitted on behalf of a third-party: [name of contributor]” when you submit it to the list.)
- Do you have the right to grant the copyright and patent licenses for the contribution that are set forth in the CPL version 1.0 license and Apache License version 2.0?
- Does your employer have any rights to code that you have written, for example, through your contract for employment? If so, has your employer given you permission to contribute the code on its behalf or waived its rights in the code?
- Are you aware of any third-party licenses or other restrictions (such as related patents or trademarks) that could apply to your contribution? If so, what are they?

Appendix

C

developerWorks articles

Topics:

- [*Introduction to the Darwin Information Typing Architecture*](#)
- [*Specializing topic types in DITA*](#)
- [*Specializing domains in DITA*](#)
- [*How to define a formal information architecture with DITA map domains*](#)

This document contains articles about DITA that were originally published on developerWorks. The online articles have been updated more recently than the versions in this document.

Introduction to the Darwin Information Typing Architecture

This document is a roadmap for the Darwin Information Typing Architecture: what it is and how it applies to technical documentation. It is also a product of the architecture, having been written entirely in XML and produced using the principles described here...

Executive summary

The Darwin Information Typing Architecture (DITA) is an XML-based, end-to-end architecture for authoring, producing, and delivering technical information. This architecture consists of a set of design principles for creating "information-typed" modules at a topic level and for using that content in delivery modes such as online help and product support portals on the Web.

At the heart of DITA (Darwin Information Typing Architecture), representing the generic building block of a topic-oriented information architecture, is an XML document type definition (DTD) called "the topic DTD." The extensible architecture, however, is the defining part of this design for technical information; the topic DTD, or any schema based on it, is just an instantiation of the design principles of the architecture.

Background

This architecture and DTD were designed by a cross-company workgroup representing user assistance teams from across IBM. After an initial investigation in late 1999, the workgroup developed the architecture collaboratively during 2000 through postings to a database and weekly teleconferences. The architecture has been placed on IBM's developerWorks Web site as an alternative XML-based documentation system, designed to exploit XML as its encoding format. With the delivery of these significant updates contains enhancements for consistency and flexibility, we consider the DITA design to be past its prototype stage.

Information interchange, tools management, and extensibility

IBM, with millions of pages of documentation for its products, has its own very complex SGML DTD, IBMIDDoc, which has supported this documentation since the early 1990s. The workgroup had to consider from the outset, "Why not just convert IBMIDDoc or use an existing XML DTD such as DocBook, or TEI, or XHTML?" The answer requires some reflection about the nature of technical information.

First, both SGML and XML are recognized as meta languages that allow communities of data owners to describe their information assets in ways that reflect how they develop, store, and process that information. Because knowledge representation is so strongly related to corporate cultures and community jargon, most attempts to define a *universal DTD* have ended up either unused or unfinished. The *ideal for information interchange* is to share the semantics and the transformational rules for this information with other data-owning communities.

Second, most companies rely on many delivery systems, or process their information in ways that differ widely from company to company. Therefore any attempt at a *universal tool set* also proves futile. The *ideal for tools management* is to base a processing architecture on standards, to leverage the contributed experience of many others, and to solve common problems in a broad community.

Third, most attempts to formalize a document description vocabulary (DTD or schema) have been done as information modelling exercises to capture the *current business practices* of data owners. This approach tends to encode *legacy* practices into the resulting DTDs or vocabularies. The *ideal for future extensibility* in DTDs for technical information (or any information that is continually exploited at the leading edge of technology) is to build the fewest presumptions about the "top-down" processing system into the design of the DTD.

In the beginning, the workgroup tried to understand the role of XML in this leading edge of information technology. As the work progressed, the team became aware that any DTD design effort would have to account for a plurality of vocabularies, a tools-agnostic processing paradigm, and a legacy-free view of information structures. Many current DTDs incorporate ways to deal with some of these issues, but the breadth of the issues lead to more than just a DTD. To support many products, brands, companies, styles, and delivery methods, the entire authoring-to-delivery process had to be considered. What resulted was a range of recommendations that required us to represent our design, not just as a DTD, but as an information architecture.

Main features of the DITA architecture

As the "Architecture" part of DITA's name suggests, DITA has unifying features that serve to organize and integrate information:

- *Topic orientation.* The highest standard structure in DITA is the topic. Any higher structure than a topic is usually part of the processing context for a topic, such as a print-organizing structure or the helpset-like navigation for a set of topics. Also, topics have no internal hierarchical nesting; for internal organization, they rely on sections that define or directly support the topic.
- *Reuse.* A principal goal for DITA has been to reduce the practice of copying content from one place to another as a way of reusing content. Reuse within DITA occurs on two levels:
 - *Topic reuse.* Because of the non-nesting structure of topics, a topic can be reused in any topic-like context. Information designers know that when they reuse a topic in a new information model, the architecture will process it consistently in its new context.
 - *Content reuse.* The SGML method of declaring reusable external entities is available for XML users, but this has several practical limitations in XML. DITA instead leans toward a different SGML reuse technique and provides each element with a `conref` attribute that can point to any other equivalent element in the same or any other topic. This referencing mechanism starts with a base element, thus assuring that a fail-safe structure is always part of the calling topic (the topic that contains the element with the `conref` attribute). The new content is always functionally equivalent to the element that it replaces.
- *Specialization.* The class mechanism in CSS indicates a common formatting semantic for any element that has a matching class value. In the same way, any DITA element can be extended into a new element whose identifier gets added to the class attribute through its DTD. Therefore, a new element is always associated to its base, or to any element in its specialization sequence.
 - *Topic specialization.* Applied to topic structures, specialization is a natural way to extend the generic topic into new information types (or infotypes), which in turn can be extended into more specific instantiations of information structures. For example, a recipe, a material safety data sheet, and an encyclopedia article are all potential derivations from a common reference topic.
 - *Domain specialization.* Using the same specialization principle, the element vocabulary within a generic topic (or set of infotyped topics) can be extended by introducing elements that reflect a particular information domain served by those topics. For example, a keyword can be extended as a unit of weight in a recipe, as a part name in a hardware reference or as a variable in a programming reference. A specialized domain, such as programming phrases, can be introduced by substitution anywhere that the root elements are allowed. This makes the entire vocabulary available throughout all the infotyped topics used within a discipline. Also, a domain can be replaced within existing infotyped topics, in effect hiding the jargon of one discipline from writers dealing with the content of another. Yet both sets of topics can be appropriate for the same user roles of performing tasks or getting reference information.
- *Property-based processing.* The DITA model provides metadata and attributes that can be used to associate or filter the content of DITA topics with applications such as content management systems, search engines, processing filters, and so on.
 - *Extensive metadata to make topics easier to find.* The DITA model for metadata supports the standard categories for the Dublin Core Metadata Initiative. In addition, the DITA metadata enables many different content management approaches to be applied to its content.
 - *Universal properties.* Most elements in the topic DTD contain a set of universal attributes that enable the elements to be used as selectors, filters, content referencing infrastructure, and multi-language support. In addition, some elements, whose attributes can serve a range of specialized roles, have been analyzed to make sure that their enumerated values provide a rich basis for specialization (which usually constrains values and never adds to them).
- *Taking advantage of existing tags and tools.* Rather than being a radical departure from the familiar, DITA builds on well-accepted sets of tags and can be used with standard XML tools.
 - *Leveraging popular language subsets.* The core elements in DITA's topic DTD borrow from HTML and XHTML, using familiar element names like `p`, `ol`, `ul`, `dl` within an HTML-like topic structure. In fact, DITA topics can be written, like HTML, for rendering directly in a browser. In more ambitious designs, DITA topics can be written, like SGML, to be normalized through processing into a deliverable, say XHTML or a well-

formed XML format targeted for a particular browser's ability to handle XML. Also, DITA makes use of the popular OASIS (formerly CALS) table model.

- *Leveraging popular and well-supported tools.* The XML processing model is widely supported by a number of vendors. The class-based extension mechanism in DITA translates well to the design features of the XSLT and CSS stylesheet languages defined by the World Wide Web Consortium and supported in many transformation tools, editors and browsers. DITA topics can be processed by a spectrum of tools ranging from shareware to custom tailored products, on almost any operating platform.

Topic as the basic architectural unit

The various information architectures for online deliverables all tend to focus on the idea of topics as the main design point for such information. A topic is a unit of information that describes a single task, concept, or reference item. The information category (concept, task, or reference) is its information type (or infotype). A new information type can be introduced by **specialization** from the structures in the base topic DTD. Typed topics are easily managed within content management systems as reusable, stand-alone units of information. For example, selected topics can be gathered, arranged, and processed within a **delivery context** to provide a variety of deliverables. These deliverables might be groups of recently updated topics for review, helpsets for building into a user assistance application, or even chapters or sections in a booklet that are printed from user-selected search results or "shopping lists."

Benefits of the DITA architecture

Through topic granularity and topic type specialization, DITA brings the following benefits of the object-oriented model to information sets:

- *Encapsulation.* The designer of the topic type only needs to address a specific, manageable problem domain. The author only needs to learn the elements that are specific to the topic type. The implementer of the processing for the topic type only needs to process elements that are special.
- *Polymorphism.* Special topic types can be treated as more generic topic types for common processing.
- *Message passing.* The class attribute preserves at all times the derivation hierarchy of an element. At any time, a topic may be generalized back to any earlier form, and if the class attributes are preserved, these topics may be re-specialized. One use of this capability would be to allow two separate disciplines to merge data at an earlier common part of the specialization hierarchy, after which they can be transformed into one, the other, or a brand new domain and set of infotyped topics.

DITA can be considered object-oriented in that:

- Data and processors are separated from their environment and can be chunked to provide behaviors similar to object-orientation (such as override transforms that modify or redefine earlier behaviors).
- Classification of elements through a sequence of derivations that are progressively more specific, possibly more constrained, and always rigidly tied to a consistent processing or rendering model.
- Inheritance of behaviors, to the extent that new elements either fall through to behaviors for ancestors in their derivation hierarchy, or can be mapped to modified processors that extend previous behaviors.

With discipline and ingenuity, some of the benefits of topic information sets can be provided through a book DTD. In particular, techniques for chunking can generate topics out of a book DTD. In DITA, the converse approach is possible: a book can be assembled from a set of DITA topics. In both cases, however, the adaptation is secondary to the primary purpose of the DTD. That is, if you are primarily authoring books, it makes the most sense to use a DTD that is designed for books. If you are primarily authoring topics, it makes sense to use a DTD that is designed for topics and can scale to large, processable collections of topics.

DITA overview

The Darwin Information Typing Architecture defines a set of relationships between the document parts, processors, and communities of users of the information.

The Darwin Information Typing Architecture has the following layers that relate to specific design points expressed in its core DTD, *topic*.

Figure 1: Layers in the Darwin Information Typing Architecture

Delivery contexts			
helpset	aggregate printing	Web site; information portal	

Typed topic structures			
topic	concept	task	reference

Specialized vocabularies (domains) across information types			
Typed topic:	concept	task	reference
Included domains:	highlighting software programming user interface		

Common structures	
metadata	OASIS (CALs) table

A typed topic, whether concept, task, or reference, is a stand-alone unit of ready-to-be-published information. Above it are any processing applications that may be driven by a superset DTD; below it are the two types of content models that form the basis of all specialized DTDs within the architecture. We will look at each of these layers in more detail.

DITA delivery contexts

This domain represents the processing layer for topical information. Topics can be processed singly or within a delivery context that relates multiple topics to a defined deliverable. Delivery contexts also include document management systems, authoring units, packages for translation, and more.

delivery contexts		
helpset	aggregate printing	Web site; information portal

DITA typed topic specializations (infotyped topics)

The typed topics represent the fundamental structuring layer for DITA topic-oriented content. The basis of the architecture is the *topic* structure, from which the *concept*, *task*, and *reference* structures are specialized. Extensibility to other typed topics is possible by further specialization.

typed topic structures			
topic	concept	task	reference

The four information types (topic, concept, task, and reference) represent the primary content categories used in the technical documentation community. Moreover, specialized, information types, based on the original four, can be defined as required.

As a notable feature of this architecture, communities can define or extend additional information types that represent their own data. Examples of such content include product support information, programming message descriptions, and GUI definitions. Besides the ability to type topics and define specific content models therein, DITA also provides the ability to extend tag vocabularies that pertain to a domain. Domain specialization takes the place of what had been called "shared structures" in DITA's original design.

DITA vocabulary specialization (domains)

Commonly, when a set of infotyped topics are used within a domain of knowledge, such as computer software or hardware, a common vocabulary is shared across the infotyped topics. However, the same infotyped topic can be used across domains that have different vocabularies and semantics. For example, a hardware reference topic might refer to diagnostic codes while a software reference topic might refer to error message numbers, with neither domain necessarily needing to expose the other domain's unique vocabulary to its own writers.

Using the same technique as specialization for topics, DITA allows the definition of domains of special vocabulary that can be shared among infotyped topics. Domains can even be elided entirely, to produce typed topics that have only the core elements¹. The vocabulary of a domain can take the form of phrases, special paragraphs, and lists--basically anything allowed within a section, the smallest organizing part of a topic.

specialized vocabularies (domains) across information types			
Typed topic:	concept	task	reference
Included domains:		highlighting software programming user interface	

The basic domains defined as examples for DITA include:

Domain	Elements
highlighting	b, u, i, tt, sup, sub
software	msgph, msgblock, msgnum, cmdname, varname, filepath, userinput, systemoutput
programming	codeph, codeblock, option, var, parmname, synph, oper, delim, sep, apiname, parml, plentry, pt, pd, syntaxdiagram, synblk, groupseq, groupchoice, groupcomp, fragment, fragref, synnote, synnoteref, repsep, kwd
user interface	uicontrol, wintitle, menucascade, shortcut

By following the rules for specializing a new domain of content, you can extend, replace, or remove these domains. Moreover, content specialization enables you to name and extend *any* content element in the scope of DITA infotyped topics for a more semantically significant role in a new domain.

To enable specialized vocabulary, you declare a parameter entity equivalent for every element used in a DTD (such as topic or one of its specializations), and then use the parameter entities instead of literal element tokens within the content models of that DTD. Later, after entity substitution, because an element's parameter entity is redefined to include both the original element and the domain elements derived from that element, anywhere the original element is allowed, the other derived domain elements are also allowed. In effect, a domain-agnostic topic can be easily extended for different domains by simply changing the scope of entity set inclusions in a front-end DTD "shell" that formalizes the vocabulary extensions within that typed topic or family of typed topics

¹ In the original design of DITA, all of the shared vocabulary had been made global to all information types by being defined in the topic DTD, which had two undesirable effects:

- new vocabulary could not be added without increasing the size of the core DTD
- certain domain-specific vocabulary could not be prohibited for DTDs specialized for a different domain.

DITA common structures

One of the design points of DITA has been to exploit the reuse of common substructures within the world of XML. Accordingly, the topic DTD incorporates the OASIS table model (known originally as the CALS table model). It also has a defined set of metadata that might be shared directly with the metadata models of quite different DTDs or schemas.

common structures	
metadata	OASIS (CALS) table

The metadata structure defines document control information for individual topics, higher-level processing DTDs, or HTML documents that are associated to the metadata as side files or as database records.

The table structure provides presentational semantics for body-level content. The OASIS/CALS table display model is supported in many popular XML editors.

Elements designed for specialization

DITA provides a rich base for specialization because of the general design of elements used in its archetype-like topic DTD.

For example, a section in the base topic DTD can contain both text and element data. However, a section can be specialized to eliminate PCDATA, yielding an element-only content model similar to the body level of most DTDs. Specialized another way, a section can eliminate most block-like elements and thus be characterized as a description for definitions, field labels, parts, and so forth.

In DITA, an effort has been made to select element names that are popular or that are common with HTML. Some semantic names have been borrowed from industry DTDs that support large SGML libraries, such as IBMIDDoc and DocBook.

The attribute lists within the topic DTD reflect this design philosophy. For example, one of the "universal attributes" (they appear on most elements) is `importance`, which defines values for weightings or appraisals that are often used as properties in specialized elements. This attribute shows up in several elements of the task topic specialization with only two allowed values out of the original set, "optional" and "required." In other domains, the elements are more appropriately ranked as "high" or "low," again values that are provided at the topic level.

The values of specialization

A company that has specific information needs can define specialized topic types. For example, a product group might identify three main types of reference topic: messages, utilities, and APIs. By creating a specialized topic type for each type of content, the product architect can be assured that each type of topic has the appropriate content. In addition, the specialized topics make XML-aware search more useful because users can make fine-grained distinctions. For example, a user could search for "xyz" only in messages or only in APIs, as well as search for "xyz" across reference topics in general.

There are rules for how to specialize safely: each new information type must map to an existing one and must be more restrictive in the content that it allows. With such specialization, new information types can use generic processing streams for translation, print, and Web publishing. Although a product group can override or extend these processes, they get the full range of existing processes by default without any extra work or maintenance.

A corporation can have a series of DTDs that represent a consistent set of information descriptions, each of which emphasizes the value of specialization for those new information types.

Role of content communities in the Darwin Information Typing Architecture

The technical documentation community that designed this architecture defined the basic architecture and shared resources. The content owned by specified communities (within or outside of the defining community) can reuse processors, styles, and other features already defined. But, those communities are responsible for their unique business processes based on the data that they manage. They can manage data by creating a further specialization from one of the base types.

The following figure represents how communities, as "content owners at the topic level," can specialize their content based on the core architecture.

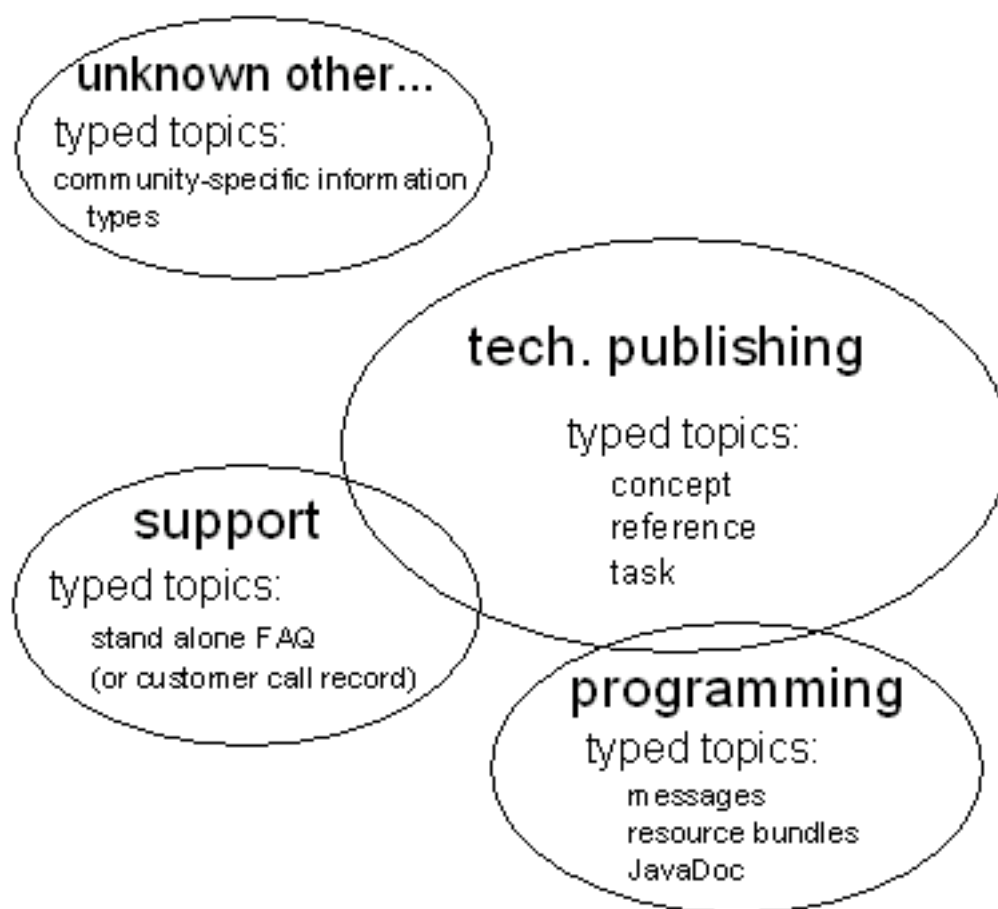


Figure 2: Relationship of specialized communities to the base architecture

In this figure, the overlap represents the common architecture and tools shared between content-owning communities that use this information architecture. New communities that define typed documents according to the architecture can then use the same tools at the outset, and refine their content-specific tools as needed.

Notices

© Copyright International Business Machines Corp., 2002, 2003. All rights reserved.

The information provided in this document has not been submitted to any formal IBM test and is distributed "AS IS," without warranty of any kind, either express or implied. The use of this information or the implementation of any of these techniques described in this document is the reader's responsibility and depends on the reader's ability to evaluate and integrate them into their operating environment. Readers attempting to adapt these techniques to their own environments do so at their own risk.

Specializing topic types in DITA

The Darwin Information Typing Architecture (DITA) provides a way for documentation authors and architects to create collections of typed topics that can be easily assembled into various delivery contexts. Topic specialization is

the process by which authors and architects can define topic types, while maintaining compatibility with existing style sheets, transforms, and processes. The new topic types are defined as an extension, or delta, relative to an existing topic type, thereby reducing the work necessary to define and maintain the new type.

The point of the XML-based Darwin Information Typing Architecture (DITA) is to create modular technical documents that are easy to reuse with varied display and delivery mechanisms, such as helpsets, manuals, hierarchical summaries for small-screen devices, and so on. This article explains how to put the DITA principles into practice with regards to the creation of a DTD and transforms that will support your particular information types, rather than just using the base DITA set of concept, task, and reference.

Topic specialization is the process by which authors and architects define new topic types, while maintaining compatibility with existing style sheets, transforms, and processes. The new topic types are defined as an extension, or delta, relative to an existing topic type, thereby reducing the work necessary to define and maintain the new type.

The examples used in this paper use XML DTD syntax and XSLT; if you need background on these subjects, see [Resources](#).

Architectural context

In SGML, architectural forms are a classic way to provide mappings from one document type to another. Specialization is an architectural-forms-like solution to a more constrained problem: providing mappings from a more specific topic type to a more general topic type. Because the specific topic type is developed with the general topic type in mind, specialization can ignore many of the thornier problems that architectural forms address. This constrained domain makes specialization processes relatively easy to implement and maintain. Specialization also provides support for multi-level or hierarchical specializations, which allow more general topic types to serve as the common denominator for different specialized types.

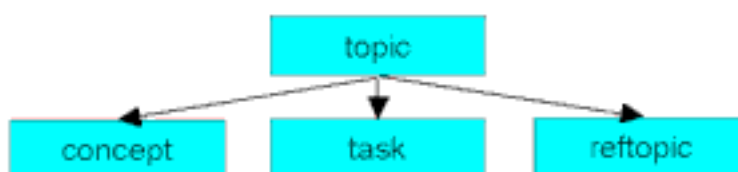
The specialization process was created to work with DITA, although its principles and processes apply to other domains as well. This will make more sense if you consider an example: Given specialization and a generic DTD such as HTML, you can create a new document type (call it MyHTML). In MyHTML you could enforce site standards for your company, including specific rules about forms layout, heading levels, and use of font and blink tags. In addition, you could provide more specific structures for product and ordering information, to enable search engines and other applications to use the data more effectively.

Specialization lets MyHTML be defined as an extension of the HTML DTD, declaring new element types only as necessary and referencing HTML's DTD for shared elements. Wherever MyHTML declares a new element, it includes a mapping back to an existing HTML element. This mapping allows the creation of style sheets and transforms for HTML that operate equally well on MyHTML documents. When you want to handle a structure differently (for example, to format product information in a particular way), you can define a new style sheet or transform that holds the extending behavior, and then import the standard style sheet or transform to handle the rest. In other words, new behavior is added as extensions to the original style sheet, in the same way that new constraints were added as extensions to the original DTD or schema.

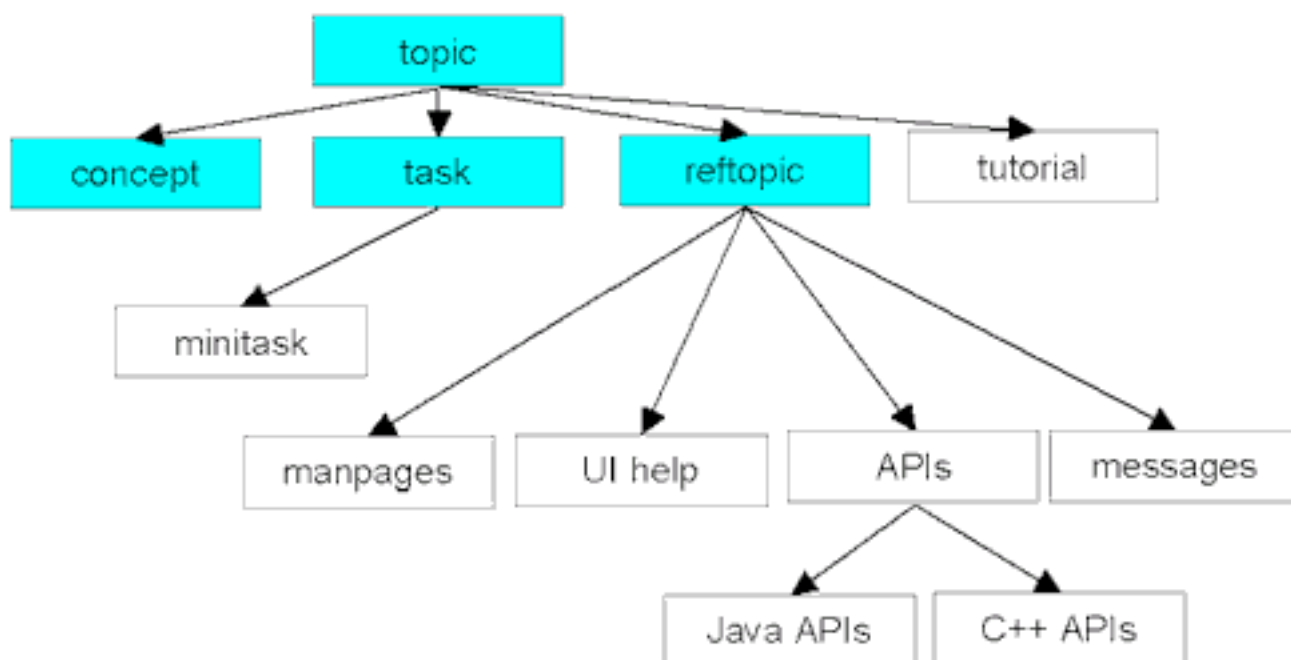
Specializing information types

The Darwin Information Typing Architecture is less about document types than information types. A document is considered to be made up of a number of topics, each with its own information type. A topic is, simply, a chunk of information consisting of a heading and some text, optionally divided into sections. The information type describes the content of the topic: for example, the type of a given topic might be "concept" or "task."

DITA has three types of topic: a generic topic, or information-typed concept, task, and reference topics. Concept, task, and reference topics can all be considered specializations of topic:



Additional information types can be added to the architecture as specializations of any of these three basic types, or as a peer specialization directly off of topic; and any of these additional specializations can in turn be specialized:



Each new information type is defined as an extension of an existing information type: the specializing type inherits, without duplication, any common structures; and the specializing type provides a mapping between its new elements and the general type's existing elements. Each information type is defined in its own DTD module, which defines only the new elements for that type. A document that consists of exactly one information type (for example, a task document in a help web) has a document type defined by all the modules in the information type's specialization hierarchy (for example, task.mod and topic.mod). A document type with multiple information types (for example, a book consisting of concepts, tasks, and reference topics) includes the modules for each of the information types used, as well as the modules for their ancestors (concept.mod, task.mod, reference.mod, plus their ancestor topic.mod).

Because of the separation of information types into modules, you can define new information types without affecting ancestor types. This separation gives you the following benefits:

- Reduces maintenance costs: each authoring group maintains only the elements that it uniquely requires
- Increases compatibility: the core information types can be centrally maintained, and changes to the core types are reflected in all specializing types
- Distributes control: reusability is controlled by the reuser, instead of by the author; adding a new type does not affect the maintenance of the core type, and does not affect other users of different types

Any information-typed topic belongs to multiple types. For example, an API description is, in more general terms, a reference topic.

Specialization example: Reference topic

Consider the specialization hierarchy for a reference topic:



Table 1 expresses the relationship between the general elements in `topic` and the specific elements in `reference`. Within the table, the columns, rows, and cells indicate information types, element mappings, and elements. Table 2 explains the relationships in detail to help you interpret Table 1.

Table 15: Relationships between `topic` and a specialization based on it

Topic	Reference
(<code>topic.mod</code>)	(<code>reference.mod</code>)
<code>topic</code>	<code>reference</code>
<code>title</code>	
<code>body</code>	<code>refbody</code>
<code>simpletable</code>	<code>properties</code>
<code>section</code>	<code>refsyn</code>

Structure

Columns

Associations

The **Topic** column shows basic `topic` structure, which comprises a title and body with optional sections, as declared in a DTD module called `topic.mod`. The **Reference** column shows a more specialized structure, with `reference` replacing `topic`, `refbody` replacing `body`, and `refsyn` replacing `section`; these new elements are declared in a DTD module called `reference.mod`.

Rows

Each row represents a mapping between the elements in that row. The elements in the **Reference** column specialize the elements in the **Topic** column. Each general element also serves as a category for more specialized elements in the same row. For example, `reference`'s `refsyn` is a kind of `section`.

Cells

Each cell in a column represents the following possibilities in relation to the cell to its left:

- A blank cell: The element in the cell to the left is reused as-is. For example, a `referencetitle` is the same as a `topictitle`, and `topic`'s declaration of the `title` element can be used by `reference`.
- A full cell: An element that is specific to the current type replaces the more general element to the left. For example, in `reference`, `refbody` replaces the more general `body`.
- A split row with a blank cell: The new specializations are in addition to the more general element, which remains available in the specialized type. For example, `reference` adds `properties` as a special type of `simpletable` (`dl`), but the general kind of `simpletable` remains available in `reference`.

The reference type module

Listing 1 illustrates not the actual `reference.mod` content, but a simplified version based on Table 1. The use of entities in the content models support domain specialization, as described in the domain specialization article.

Listing 1. reference.mod

```
<!ELEMENT reference ((%title;), (%prolog;)?, (%refbody;),(%info-types;)* )>
<!ELEMENT refbody (%section; | refsyn | %simpletable; | properties)*>
<!ELEMENT properties ((%sthead;)?, (%strow;)+) >
<!ELEMENT refsyn (%section;)* >
```

Most of the content models declared here depend on elements or entities declared in `topic.mod`. Therefore, if `topic`'s structure is enhanced or changed, most of the changes will be picked up by `reference` automatically. Also the definition of `reference` remains simple: it doesn't have to redeclare any of the content that it shares with `topic`.

Adding specialization attributes

To expose the element mappings, we add an attribute to each element that shows its mappings to more general types.

Listing 2. reference.mod (part 2)

```
<!ATTLIST reference class CDATA "- topic/topic reference/reference ">
<!ATTLIST refbody class CDATA "- topic/body reference/refbody ">
<!ATTLIST properties class CDATA "- topic/simpletable reference/properties ">
<!ATTLIST refsyn class CDATA "- topic/section reference/refsyn ">
```

Later on, we'll talk about how to take advantage of these attributes when you write an XSL transform. See the appendix for a more in-depth description of the class attribute.

Creating an authoring DTD

Now that we've defined the type module (which declares the newly typed elements and their attributes) and added specialization attributes (which map the new type to its ancestors), we can assemble an authoring DTD.

Listing 3. reference.dtd

```
<!--Redefine the infotype entity to exclude other topic types-->
<!ENTITY % info-types "reftopic">
<!--Embed topic to get generic elements -->
<!ENTITY % topic-type SYSTEM "topic.mod">
%topic-type;
<!--Embed reference to get specific elements -->
<!ENTITY % reference-type SYSTEM "reference.mod">
%reference-type;
```

Specialization example: API description

Now let's create a more specialized information type: API descriptions, which are a kind of (and therefore specialization of) reference topic:

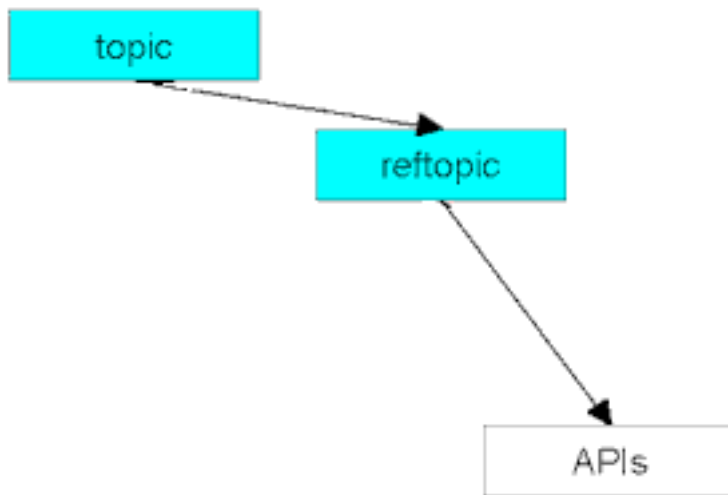


Figure 3: A more specialized information type, API description

Table 3 shows part of the specialization for an information type called `APIdesc`, for API description. As before, each column represents an information type, with specialization occurring from left to right. That is, each information type is a specialization of its neighbor to the left. Each row represents a set of mapped elements, with more specific elements to the right mapping to more general equivalents to the left.

As before, each cell specializes the contents of the cell to its left:

- A blank cell: The element to the left is picked up by the new type unchanged. For example, `simpletable` and `refsyn` are available in an API description.
- A full cell: The element to the left is replaced by a more specific one. For example, `APIname` replaces `title`.
- A split row with a blank cell: New elements are added to the elements on the left. For example, the API description adds a `usage` section as a peer of the `refsyn` and `section` elements.

Table 16: Summary of `APIdesc` specialization

Topic	Reference	APIdesc
(topic.mod)	(reference.mod)	(APIdesc.mod)
topic	reference	APIdesc
title		APIname
body	refbody	APIbody
simpletable	properties	parameters
section	refsyn	
		usage

The `APIdesc` module

Here you can see that the content for an API description is actually much more restricted than the content of a general reference topic. The sequence of `syntax`, then `usage`, then `parameters` is now imposed, followed by optional additional sections. This sequence is a subset of the allowable structures in a reference topic, which allows any sequence of `syntax`, `properties`, and `sections`. In addition, the label for the `usage` section is now fixed as `Usage`, taking advantage of the `spectitle` attribute of `section` (which is there for exactly this kind of `usage`): with the `spectitle` attribute providing the section title, we can also get rid of the `title` element in `usage`'s content model, making use of the predefined `section.notitle.cnt` entity.

APIdesc.mod

```
<!ELEMENT APIdesc (APIname, (%prolog;)?, APIbody, (%info-types;)* )>
<!ELEMENT APIname (%title.cnt;)*>
<!ELEMENT APIbody (refsyn, usage, parameters, (%section;)*)>
<!ELEMENT usage (%section.notitle.cnt;)*>
<!ATTLIST usage spectitle CDATA #FIXED "Usage">
<!ELEMENT parameters ((%sthead;)?, (%strow;)+)>
```

Adding specialization attributes

Every new element now has a mapping to all its ancestor elements.

APIdesc.mod (part 2)

```
<!ATTLIST APIdesc class CDATA "- topic/topic reference/reference APIdesc/
APIdesc " >
<!ATTLIST APIname spec CDATA "- topic/title reference/title APIdesc/APIname "
>
<!ATTLIST APIbody spec CDATA "- topic/body reference/refbody APIdesc/APIbody"
>
<!ATTLIST parameters spec CDATA "- topic/simpletable reference/properties
APIdesc/parameters ">
<!ATTLIST usage spec CDATA "- topic/section reference/section APIdesc/usage ">
```

Note that APIname explicitly identifies its equivalent in both reference and topic, even though they are the same (title) in both cases. In the same way, usage explicitly maps to section in both reference and topic. This explicit identification makes it easier for processes to keep track of complex mappings. Even if you had a specialization hierarchy 10 levels deep or more, the attributes would still allow unambiguous mappings to each ancestor information type.

Authoring DTDs

Now that we've defined the type module (which declares the newly typed elements and their attributes) and added specialization attributes (which map the new type to its ancestors), we can assemble an authoring DTD.

APIdesc.dtd

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

Working with specialization

After a specialized type has been defined the necessary attributes have been declared, they can provide the basis for the following operations:

- Applying a general style sheet or transform to a specialized topic type
- Generalizing a topic of a specialized type (transforming it into a more generic topic type)
- Specializing a topic of a general type (transforming it into a more specific topic type - to be used only when a topic was originally authored in specialized form, and has gone through a general stage without breaking the constraints of its original form)

Applying general style sheets or transforms

Because content written in a new information type (such as `APIdesc`) has mappings to equivalent or less restrictive structures in preexisting information types (such as `reference` and `topic`), the preexisting transforms and processes can be safely applied to the new content. By default, each specialized element in the new information type will be treated as an instance of its general equivalent. For example, in `APIdesc` the `<usage>` element will be treated as a topic `<section>` element that happens to have the fixed label "Usage".

To override this default behavior, an author can simply create a new, more specific rule for that element type, and then import the default style sheet or transform, thus extending the behavior without directly editing the original style sheet or transform. This reuse by reference reduces maintenance costs (each site maintains only the rules it uniquely requires) and increases consistency (because the core transform rules can be centrally maintained, and changes to the core rules will be reflected in all other transforms that import them). Control over reuse has moved from the author of the transform to the reuser of the transform.

The rest of this section assumes knowledge of XSLT, the XSL Transformations language.

Requirements

This process works only if the general transforms have been enabled to handle specialized elements, and if the specialized elements include enough information for the general transform to handle them.

Requirement 1: mapping attributes

To provide the specialization information, you need to add specialization attributes, as outlined previously. After you include the attributes in your documents, they are ready to be processed by specialization-aware transforms.

Requirement 2: specialization-aware transforms

For the transform, you need template rules that check for a match against both the element name and the attribute value.

The specialization-aware interface

```
<xsl:template match="*[contains(@class," topic/simpletable ")]">
<!--matches any element that has a class attribute that mentions
      topic/simpletable-->
<!--do something-->
</xsl:template>
```

Example: overriding a transform

To override the general transform for a specific element, the author of a new information type can create a transform that declares the new behavior for the specific element and imports the general transform to provide default behavior for the other elements.

For example, an `APIdesc` specialized transform could allow default handling for all specialized elements except `parameters`:

A specialized transformation for `APIdesc`

```
<xsl:import href="general-transform.xsl"/>
<xsl:template match="*[contains(@class," APIdesc/parameters ")]">
  <!--do something-->
<xsl:apply-templates/>
</xsl:template>
```

Both the preexisting `referenceproperties` template rule and the new `parameters` template rule match when they encounter a `parameters` element (because the `parameters` element is a specialized type of `referenceproperties` element), and its class attribute contains both values). However, because the `parameters` template is in the *importing* style sheet, the new template takes precedence.

Generalizing a topic

Because a specialized information type is also an instance of its ancestor types (an `APIdesc` is a reference topic is a topic), you can safely transform a specialized topic to one of its more generic ancestors. This upward compatibility is useful when you want to combine sets of documentation from two sources, each of which has specialized differently. The ancestor type provides a common denominator that both can be safely transformed to. This compatibility may also be useful when you have to feed topics through processes that are not specialization-aware. For example, a publication center that charges per document type or uses non-DTD-aware processes could be sent a generalized set of documents, so that they only support one document type or set of markup. However, wherever possible, you should use specialization-aware processes and transforms, so that you can avoid generalizing and process your documents in their more descriptive, specialized form.

To safely generalize a topic, you need a way to map from your information type to the target information type. You also need a way to preserve the original type in case you need round-tripping later.

The `class` attribute that was introduced previously serves two purposes. It provides:

- The information needed to map.
- A way to preserve the information to allow round-tripping.

Each level of specialization has its own set of class attributes, which in the end provide the full specialization hierarchy for all specialized elements.

Consider the `APIdesc` topic in Listing 11:

A sample topic from `APIdesc`

```
<APIdesc>
  <APIname>AnAPI</APIname>
  <APIbody>
    <refsyn>AnAPI (parm1, parm2)</refsyn>
    <usage spectitle="Usage">Use AnAPI to pass parameters to your process.
    </usage>
    <parameters >
      ...
    </parameters>
  </APIbody>
</APIdesc>
```

With the class attributes exposed (all values are provided as defaults by the DTD):

The same sample topic from `APIdesc`, including the class attributes

```
<APIdesc class="- topic/topic reference/reference APIdesc/APIdesc ">
  <APIname class="- topic/title reference/title APIdesc/APIname ">AnAPI
  </APIname>
  <APIbody class="- topic/body reference/refbody APIdesc/APIbody ">
    <refsyn class="- topic/section reference/refsyn ">AnAPI(parm1,
    parm2)</refsyn>
    <usage class="- topic/section reference/section APIdesc/usage "
    spectitle="Usage">
      <p class="- topic/p ">Use AnAPI to pass parameters to your process.</p>
    </usage>
    <parameters class="topic/simpletable reference/properties APIdesc/parameters
    ">
      ...
    </parameters>
  </APIbody>
</APIdesc>
```

From here, a single template rule can transform the entire `APIdesc` topic to either a reference or a generic topic. The template rule simply looks in the `class` attribute for the ancestor element name, and renames the current element to match.

After a transform to topic, it should look something like Listing 13:

A transformed topic from APIdesc

```
<topic class="- topic/topic reference/reference APIdesc/APIdesc ">
  <title class="- topic/title reference/title APIdesc/APIname ">AnAPI
  </title>
  <body class="- topic/body reference/refbody APIdesc/APIbody ">
    <section class="- topic/section reference/refsyn ">AnAPI(parm1,
    parm2)</section>
    <section class="- topic/section reference/section APIdesc/usage "
    spectitle="Usage">
      <p class="- topic/p ">Use AnAPI to pass parameters to your process.</p>
    </section>
    <simplatable class="topic/simplatable reference/properties APIdesc/
parameters ">
      ...
    </simplatable>
  </body>
</topic>
```

Even after generalization, specialization-aware transforms can continue to treat the topic as an `APIdesc`, because the transforms can look in the `class` attribute for information about the element type hierarchy.

From here, it is possible to round-trip by reversing the transformation (looking in the `class` attribute for the specializing element name, and renaming the current element to match). Whenever the `class` attribute doesn't list the target (the first section has no `APIdesc` value), the element is changed to the last value listed (so the first section becomes, accurately, a `refsyn`).

However, if anyone changes the structure of the content while it is a generic `topic` (as by changing the order of sections), the result might not be valid anymore under the specialized information type (which in the `APIdesc` case enforces a particular sequence of information in the `APIbody`). So although mapping to a more general type is always safe, mapping back to a specialized type can be problematic: The specialized type has more rules, which make the content specialized. But those rules aren't enforced while the content is encoded more generally.

Specializing a topic

It is relatively trivial to specialize a general topic if the content was originally authored as a specialized type.

However, a more complex case can result if you have authored content at a general level that you now want to type more precisely.

For example, suppose that you create a set of reference topics. Then, having analyzed your content, you realize that you have a consistent pattern. Now you want to enforce this pattern and describe it with a specialized information type (for example, API descriptions). In order to specialize, you need to first create the target DTD and then add enough information to your content to allow it to be migrated.

You can put the specializing information in either of two places:

- Add it to the `class` attribute. You need to be careful to get the order correct, and include all ancestor type values.
- Or give the name of the target element in an `outputclass` attribute, migrate based on that value, and add the `class` attribute values afterward.

In either case, before migration you can run a validation transform that looks for the appropriate attribute, then checks that the content of the element will be valid under the specialized content model. You can use a tool like Schematron to generate both the validating transform and the migrating transform, or you can migrate first and use the specialized DTD to validate that the migration was successful.

Specializing with schemas

Like the XML DTD syntax, the XML Schema language is a way of defining a vocabulary (elements and attributes) and a set of constraints on that vocabulary (such as content models, or fixed vs. implied attributes). It has a built-in specialization mechanism, which includes the capability to restrict allowable specializations. Using the XML Schema

language instead of DTDs would make it much easier to validate that specialized information types represent valid subsets of generic types, which ensures smooth processing by generic translation and publishing transforms.

Unlike DTDs, XML schemas are expressed as XML documents. As a result, they can be processed in ways that DTDs cannot. For example, we can maintain a single XML schema and then use XSL to generate two versions:

- An authoring version of it that eliminates any fixed attributes and any overridden elements
- A processor-ready version of it that includes the class attributes that drive the translation and publishing transforms

However, XML schemas are not yet popular enough to adopt wholeheartedly. The main problems are a lack of authoring tools, and incompatibilities between the implementations of an evolving standard. These problems should be remedied by the industry over the next year or so, as the standard is finalized and schemas become more widely adopted and supported.

Summary

You can create a specialized information type by using this general procedure:

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 type module file that holds your specialized element and attribute declarations (including the `class` attribute).
5. Create an authoring DTD file that imports the appropriate type modules.

You can create specialized XSL transforms by using this general procedure:

1. Create a new transform for your information type.
2. Import the existing transform that you want to extend.
3. Identify the elements that you need to treat specially.
4. Add template rules that match those elements, based on their `class` attribute content.

Appendix: Rules for specialization

Although you could create a new element equivalent for any tag in a general DTD, this work is useless to you as an author unless the content models that would include the tag are also specialized. In the `APIdesc` example, the `parameters` element is not valid content anywhere in `topic` or `reference`. For it to be used, you need to create valid contexts for `parameters`, all the way up to the topic-level container. To expose the `parameters` element to your authors, you need to specialize the following parts:

- A `body` element, to allow `parameters` as valid content (giving us `APIbody`)
- A `topic` element, to allow the specialized `body` (giving us `APIdesc`)

This domino effect can be avoided by using domain specialization. If you truly just want to add some new variant structures to an existing information type, use domain specialization instead of topic specialization (see [Specializing domains in DITA](#)).

To ensure that the specialized elements are more constrained than their general equivalents (that is, that they allow a proper subset of the structures that the general equivalent allows), you need to look at the content model of the general element. You can safely change the content model of your specialized element as shown in Table A:

Table 17: Summary of specialization rules

Content type	Allowed specialization	Example (Special specializing General)
Required	Rename only	<code><!ELEMENT General(a)></code>
		<code><!ELEMENT Special(a.1)></code>

Content type	Allowed specialization	Example (Special specializing General)
Optional (?)	Rename, make required, or delete	<pre><!ELEMENT General(a?)></pre> <pre><!ELEMENT Special(a.1?)></pre> <pre><!ELEMENT Special(a.1)></pre> <pre><!ELEMENT Special EMPTY></pre>
One or more (+)	Rename, make required, split into a required element plus others, split into one or more elements plus others.	<pre><!ELEMENT General(a+)></pre> <pre><!ELEMENT Special(a.1+)></pre> <pre><!ELEMENT Special(a.1)></pre> <pre><!ELEMENT Special(a.1,a.2,a.3+,a.4*)></pre> <pre><!ELEMENT Special(a.1+,a.2,a.3*)></pre>
Zero or more (*)	Rename, make required, make optional, split into a required element plus others, split into an optional element plus others, split into one-or-more plus others, split into zero-or-more plus others, or delete	<pre><!ELEMENT General(a*)></pre> <pre><!ELEMENT Special(a.1*)></pre> <pre><!ELEMENT Special(a.1)></pre> <pre><!ELEMENT Special(a.1?)></pre> <pre><!ELEMENT Special(a.1,a.2,a.3+,a.4*)></pre> <pre><!ELEMENT Special(a.1?,a.2,a.3+,a.4*)></pre> <pre><!ELEMENT Special(a.1+,a.2,a.3*)></pre> <pre><!ELEMENT Special(a.1*,a.2?,a.3*)></pre> <pre><!ELEMENT Special EMPTY></pre>
Either-or	Rename, or choose one	<pre><!ELEMENT General (a b)></pre> <pre><!ELEMENT Special (a.1 b.1)></pre> <pre><!ELEMENT Special (a.1)></pre>

Extended example

You have a general element `General`, with the content model `(a,b?,(c|d+))`. This definition means that a `General` always contains element `a`, optionally followed by element `b`, and always ends with either `c` or one or more `d`'s.

The content model for the general element `General`

```
<!ELEMENT General (a,b?,(c|d+))>
```

When you specialize `General` to create `Special`, its content model must be the same or more restrictive: It cannot allow more things than `General` did, or you will not be able to map upward, or guarantee the correct behavior of general processes, transforms, or style sheets.

Leaving aside renaming (which is always allowed, and simply means that you are also specializing some of the elements that `Special` can contain), here are some valid changes that you could make to the content model of `Special`, resulting in the same or more restrictive content rules:

A valid change to the model `Special`, making `b` mandatory

```
<!ELEMENT Special (a,b,(c|d))>
```

`Special` now requires `b` to be present, instead of optional, and allows only one `d`. It safely maps to `General`.

A valid change to the model `Special`, making `c` mandatory and disallowing `d`

```
<!ELEMENT Special (a,b?,c)>
```

`Special` now requires `c` to be present, and no longer allows `d`. It safely maps to `General`.

A valid change to the model `Special`, making three specializations of `d` mandatory

```
<!ELEMENT Special (a,b?,d1,d2,d3)>
```

`Special` now requires three specializations of `d` to be present, and does not allow `c`. It safely maps to `General`.

Details of the class attribute

Every element must have a class attribute. The class attribute starts and ends with white space, and contains a list of blank-delimited values. Each value has two parts: the first part identifies a topic type, and the second part (after a /) identifies an element type. The class attribute value should be declared as a default attribute value in the DTD. Generally, it should not be modified by the author.

Example:

```
<appstep class="- topic/li task:step bctask/appstep ">A specialized step</appstep>
```

When a specialized type declares new elements, it must provide a class attribute for the new element. The class attribute must include a mapping for every topic type in the specialized type's ancestry, even those in which no element renaming occurred. The mapping should start with topic, and finish with the current element type.

Example:

```
<appname class="- topic/kwd task/kwd bctask/appname ">
```

This is necessary so that generalizing and specializing transforms can map values simply and accurately. For example, if task/kwd was missing as a value, and I decided to map this bctask up to a task topic, then the transform would have to guess whether to map to kwd (appropriate if task is more general, which it is) or leave as appname (appropriate if task were more specialized, which it isn't). By always providing mappings for more general values, we can then apply the simple rule that missing mappings must by default be to more specialized values, which means the last value in the list is appropriate. While this example is trivial, more complicated hierarchies (say, five levels deep, with renaming occurring at two and four only) make this kind of mapping essential.

A specialized type does not need to change the class attribute for elements that it does not specialize, but simply reuses by reference from more generic levels. For example, since task and bctask use the p element without specializing it, they don't need to declare mappings for it.

A specialized type only declares class attributes for the elements that it uniquely declares. It does not need to declare class attributes for elements that it reuses or inherits.

Using the class attribute

Applying an XSLT template based on class attribute values allows a transform to be applied to whole branches of element types, instead of just a single element type.

Wherever you would check for element name (any XPath statement that contains an element name value), you need to enhance this to instead check the contents of the element's class attribute. Even if the element is unrecognized, the class attribute can let the transform know that the element belongs to a class of known elements, and can be safely treated according to their rules.

Example:

```
<xsl:template match="*[contains(@class,' topic/li ')]">
This match statement will work on any li element it encounters. It will also
work on step and appstep elements, even though it doesn't know what they are
specifically, because the class attribute tells the template what they are
generally.
<xsl:template match="*[contains(@class,' task/step ')]">
```

This match statement won't work on generic li elements, but it will work on both step elements and appstep elements; even though it doesn't know what an appstep is, it knows to treat it like a step.

Be sure to include a leading and trailing blank in your class attribute string check. Otherwise you could get false matches (without the blanks, 'task/step' would match on 'notatask/stepaway', when it shouldn't).

The class attribute in domains specialization

When you create a domains specialization, the new elements still need a class attribute, but should start with a "+" instead of a "-". This signals any generalization transforms to treat the element differently: a domains-aware generalization transform may have different logic for handling domains than for handling topic specializations.

Domain specializations should be derived either from topic (the root topic type), or from another domain specialization. Do not create a domain by specializing an already specialized topic type: this can result in unpredictable generalization behavior, and is not currently supported by the architecture.

Notices

© Copyright International Business Machines Corp., 2002, 2003. All rights reserved.

The information provided in this document has not been submitted to any formal IBM test and is distributed "AS IS," without warranty of any kind, either express or implied. The use of this information or the implementation of any of these techniques described in this document is the reader's responsibility and depends on the reader's ability to evaluate and integrate them into their operating environment. Readers attempting to adapt these techniques to their own environments do so at their own risk.

Specializing domains in DITA

In current approaches, DTDs are static. As a result, DTD designers try to cover every contingency and, when this effort fails, users have to force their information to fit existing types. DITA changes this situation by giving information architects and developers the power to extend a base DTD to cover their domains.

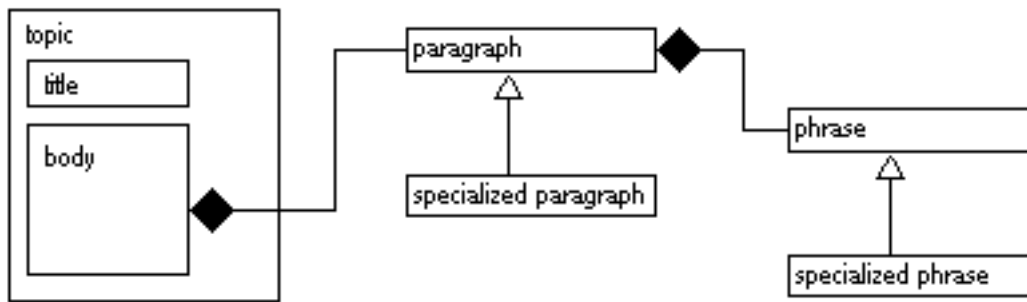
The Darwin Information Typing Architecture (DITA) is an XML architecture for extensible technical information. A domain extends DITA with a set of elements whose names and content models are unique to an organization or field of knowledge. Architects and authors can combine elements from any number of domains, leading to great flexibility and precision in capturing the semantics and structure of their information. In this overview, you learn how to define your own domains.

Introducing domain specialization

In DITA, the topic is the basic unit of processable content. The topic provides the title, metadata, and structure for the content. Some topic types provide very simple content structures. For example, the `concept` topic has a single concept body for all of the concept content. By contrast, a `task` topic articulates a structure that distinguishes pieces of the task content, such as the prerequisites, steps, and results.

In most cases, these topic structures contain content elements that are not specific to the topic type. For example, both the concept body and the task prerequisites permit common block elements such as `p` paragraphs and `ul` unordered lists.

Domain specialization lets you define new types of content elements independently of topic type. That is, you can derive new phrase or block elements from the existing phrase and block elements. You can use a specialized content element within any topic structure where its base element is allowed. For instance, because a `p` paragraph can appear within a concept body or task prerequisite, a specialized paragraph could appear there, too.



Here's an analogy from the kitchen. You might think of topics as types of containers for preparing food in different ways, such as a basic frying pan, blender, and baking dish. The content elements are like the ingredients that go into these containers, such as spices, flour, and eggs. The domain resembles a specialty grocer who provides ingredients for a particular cuisine. Your pot might contain chorizo from the *carnicería* when you're cooking TexMex or risotto when you're cooking Italian. Similarly, your topics can contain elements from the programming domain when you're writing about a programming language or elements from the UI domain when you're writing about a GUI application.

DITA has broad tastes, so you can mix domains as needed. If you're describing how to program GUI applications, your topics can draw on elements from both the programming and UI domains. You can also create new domains for *your* content. For instance, a new domain could provide elements for describing hardware devices. You can also reuse new domains created by others, expanding the variety of what you can cook up.

In a more formal definition, topic specialization starts with the containing element and works from the top down. Domain specialization, on the other hand, starts with the contained element and works from the bottom up.

Understanding the base domains

A DITA *domain* collects a set of specialized content elements for some purpose. In effect, a domain provides a specialized vocabulary. With the base DITA package, you receive the following domains:

Domain	Purpose
highlight	To highlight text with styles such as bold, italic, and monospace
programming	To define the syntax and give examples of programming languages
software	To describe the operation of a software program
UI	To describe the user interface of a software program

In most domains, a specialized element adds semantics to the base element. For example, the `apiname` element of the programming domain extends the basic `keyword` element with the semantic of a name within an API.

The highlight domain is a special case. The elements in this domain provide styled presentation instead of semantic or structural markup. The highlight styles give authors a practical way to mark up phrases for which a semantic has not been defined.

Providing such highlight styles through a domain resolves a long-standing dispute for publication DTDs. Purists can omit the highlight domain to enforce documents that should be strictly semantic. Pragmatists can include the highlight domain to provide expressive flexibility for real-world authoring. A semipragmatist could even include the highlight domain in conceptual documents to support expressive authoring but omit the highlight domain from reference documents to enforce strict semantic tagging.

More generally, you can define documents with any combination of domains and topics. As we'll see in [Generalizing a domain](#) on page 164, the resulting documents can still be exchanged.

Combining an existing topic and domain

The DITA package provides a DTD for each topic type and an omnibus DTD (`database.dtd`) that defines all of the topic types. Each of these DTDs includes all of the predefined DITA domains. Thus, topics written against one of the supplied DTDs can use all of the predefined domain specializations.

Behind the scenes, a DITA DTD is just a shell. Elements are actually defined in other modules, which are included in the DTD. Through these modules, DITA provides you with the building blocks to create new combinations of topic types and domains.

When you add a domain to your DITA installation, the new domain provides you with additional modules. You can use the additional modules to incorporate the domain into the existing DTDs or to create new DTDs.

In particular, each domain is implemented with two files:

- A file that declares the entities for the domain. This file has the `.ent` extension.
- A file that declares the elements for the domain. This file has the `.mod` extension.

As an example, let's say we're authoring the reference topics for a programming language. We're purists about presentation, so we want to exclude the highlight domain. We also have no need for the software or UI domains in this reference. We could address this scenario by defining a new shell DTD that combines the reference topic with the programming domain, excluding the other domains.

A shell DTD has a consistent design pattern with a few well-defined sections. The instructions in these sections perform the following actions:

1. Declare the entities for the domains.

In the scenario, this section would include the programming domain entities:

```
<!ENTITY % pr-d-dec PUBLIC "-//IBM//ENTITIES DITA Programming Domain//EN"
"programming-domain.ent">
%pr-d-dec;
```

2. Redefine the entities for the base content elements to add the specialized content elements from the domains.

This section is crucial for domain specialization. Here, the design pattern makes use of two kinds of entities. Each base content element has an *element entity* to identify itself and its specializations. Each domain provides a separate *domain specialization entity* to list the specializations that it provides for a base element. By combining the two kinds of entities, the shell DTD allows the specialized content elements to be used in the same contexts as the base element.

In the scenario, the `pre` element entity identifies the `pre` element (which, as in HTML, contains preformatted text) and its specializations. The programming domain provides the `pr-d-pre` domain specialization entity to list the specializations for the `pre` base element. The same pattern is used for the other base elements specialized by the programming domain:

```
<!ENTITY % pre      "pre"      | %pr-d-pre;">
<!ENTITY % keyword  "keyword"  | %pr-d-keyword;">
<!ENTITY % ph       "ph"       | %pr-d-ph;">
<!ENTITY % fig      "fig"      | %pr-d-fig;">
<!ENTITY % dl       "dl"       | %pr-d-dl;">
```

To learn which content elements are specialized by a domain, you can look at the entity declaration file for the domain.

3. Define the domains attribute of the topic elements to declare the domains represented in the document.

Like the `class` attribute, the `domains` attribute identifies dependencies. Where the `class` attribute identifies base elements, the `domains` attribute identifies the domains available within a topic. Each domain provides a *domain identification entity* to identify itself in the `domains` attribute.

In the scenario, the only topic is the `reference` topic. The only domain is the programming domain, which is identified by the `pr-d-att` domain identification entity:

```
<!ATTLIST reference domains CDATA "&pr-d-att;">
```

4. Redefine the infotypes entity to specify the topic types that can be nested within a topic.

In the scenario, this section would declare the reference topic:

```
<!ENTITY % info-types "reference">
```

5. Define the elements for the topic type, including the base topics.

In the scenario, this section would include the base topic and reference topic modules:

```
<!ENTITY % topic-type PUBLIC "-//IBM//ELEMENTS DITA Topic//EN"
"topic.mod">
%topic-type;
<!ENTITY % reference-typemod PUBLIC "-//IBM//ELEMENTS DITA Reference//EN"
"reference.mod">
%reference-typemod;
```

6. Define the elements for the domains.

In the scenario, this section would include the programming domain definition module:

```
<!ENTITY % pr-d-def PUBLIC "-//IBM//ELEMENTS DITA Programming Domain//EN"
"programming-domain.mod">
%pr-d-def;
```

Often, it would be easiest to work by copying an existing DTD and adding or removing topics or domains. In the scenario, it would be easiest to start with `reference.dtd` and remove the `highlight`, `software`, and `UI` domains as shown with the underlined text below.

```
<!--vocabulary declarations-->
<!ENTITY % ui-d-dec PUBLIC "-//IBM//ENTITIES DITA User Interface Domain//EN"
"ui-domain.ent">
%ui-d-dec;
<!ENTITY % hi-d-dec PUBLIC "-//IBM//ENTITIES DITA Highlight Domain//EN"
"highlight-domain.ent">
%hi-d-dec;
<!ENTITY % pr-d-dec PUBLIC "-//IBM//ENTITIES DITA Programming Domain//EN"
"programming-domain.ent">
%pr-d-dec;
<!ENTITY % sw-d-dec PUBLIC "-//IBM//ENTITIES DITA Software Domain//EN"
"software-domain.ent">
%sw-d-dec;

<!--vocabulary substitution-->
<!ENTITY % pre "pre | %pr-d-pre; | %sw-d-pre;">
<!ENTITY % keyword "keyword | %pr-d-keyword; | %sw-d-keyword; | %ui-d-
keyword;">
<!ENTITY % ph "ph | %pr-d-ph; | %sw-d-ph; | %hi-d-ph; |
%ui-d-ph;">
<!ENTITY % fig "fig | %pr-d-fig;">
<!ENTITY % dl "dl | %pr-d-dl;">

<!--vocabulary attributes-->
<!ATTLIST reference domains CDATA "&ui-d-att; &hi-d-att; &pr-d-att; &sw-d-
att;">

<!--Redefine the infotype entity to exclude other topic types-->
<!ENTITY % info-types "reference">
```



```

<!--Embed topic to get generic elements -->
<!ENTITY % topic-type PUBLIC "-//IBM//ELEMENTS DITA Topic//EN" "topic.mod">
%topic-type;

<!--Embed reference to get specific elements -->
<!ENTITY % reference-typemod PUBLIC "-//IBM//ELEMENTS DITA Reference//EN"
"reference.mod">
%reference-typemod;

<!--vocabulary definitions-->
<!ENTITY % ui-d-def PUBLIC "-//IBM//ELEMENTS DITA User Interface Domain//EN"
"ui-domain.mod">
%ui-d-def;
<!ENTITY % hi-d-def PUBLIC "-//IBM//ELEMENTS DITA Highlight Domain//EN"
"highlight-domain.mod">
%hi-d-def;
<!ENTITY % pr-d-def PUBLIC "-//IBM//ELEMENTS DITA Programming Domain//EN"
"programming-domain.mod">
%pr-d-def;
<!ENTITY % sw-d-def PUBLIC "-//IBM//ELEMENTS DITA Software Domain//EN"
"software-domain.mod">
%sw-d-def;

```

Creating a domain specialization

For some documents, you may need new types of content elements. In a common scenario, you need to mark up phrases that have special semantics. You can handle such requirements by creating new specializations of existing content elements and providing a domain to reuse the new content elements within topic structures.

As an example, let's say we're writing the documentation for a class library. We intend to write processes that will index the documentation by class, field, and method. To support this processing, we need to mark up the names of classes, fields, and methods within the topic content, as in the following sample:

```

<p>The <classname>String</classname> class provides
the <fieldname>length</fieldname> field and
the <methodname>concatenate()</methodname> method.
</p>

```

We must define new content elements for these names. Because the names are special types of names within an API, we can specialize the new elements from the `apiname` element provided by the programming domain.

The design pattern for a domain requires an abbreviation to represent the domain. A sensible abbreviation for the class library domain might be `cl`. The identifier for a domain consists of the abbreviation followed by `-d` (for domain).

As noted in [Combining an existing topic and domain](#) on page 159, the domain requires an entity declaration file and an element definition file.

Writing the entity declaration file

The entity declaration file has sections that perform the following actions:

1. Define the domain specialization entities.

A domain specialization entity lists the specialized elements provided by the domain for a base element. For clarity, the entity name is composed of the domain identifier and the base element name. The domain provides domain specialization entities for ancestor elements as well as base elements.

In the scenario, the domain defines a domain specialization entity for the `apiname` base element as well as the keyword ancestor element (which is the base element for `apiname`):

```
<!ENTITY % cl-d-apiname "classname | fieldname | methodname">
<!ENTITY % cl-d-keyword "classname | fieldname | methodname">
```

2. Define the domain identification entity.

The domain identification entity lists the topic type as well as the domain and other domains for which the current domain has dependencies. Each domain is identified by its domain identifier. The list is enclosed in parentheses. For clarity, the entity name is composed of the domain identifier and `-att`.

In the scenario, the class library domain has a dependency on the programming domain, which provides the `apiname` element:

```
<!ENTITY cl-d-att "(topic pr-d cl-d)">
```

The complete entity declaration file would look as follows:

```
<!ENTITY % cl-d-apiname "classname | fieldname | methodname">
<!ENTITY % cl-d-keyword "classname | fieldname | methodname">

<!ENTITY cl-d-att "(topic pr-d cl-d)">
```

Writing the element definition file

The element definition file has sections that perform the following actions:

1. Define the content element entities for the elements introduced by the domain.

These entities permit other domains to specialize from the elements of the current domain.

In the scenario, the class library domain follows this practice so that additional domains can be added in the future. The domain defines entities for the three new elements:

```
<!ENTITY % classname "classname">
<!ENTITY % fieldname "fieldname">
<!ENTITY % methodname "methodname">
```

2. Define the elements.

The specialized content model must be consistent with the content model for the base element. That is, any possible contents of the specialized element must be generalizable to valid contents for the base element. Within that limitation, considerable variation is possible. Specialized elements can be substituted for elements in the base content model. Optional elements can be omitted or required. An element with multiple occurrences can be replaced with a list of specializations of that element, and so on.

The specialized content model should always identify elements through the element entity rather than directly by name. This practice lets other domains merge their specializations into the current domain.

In the scenario, the elements have simple character content:

```
<!ELEMENT classname      (#PCDATA)>
<!ELEMENT fieldname      (#PCDATA)>
<!ELEMENT methodname     (#PCDATA)>
```

3. Define the specialization hierarchy for the element with `class` attribute.

For a domain element, the value of the attribute must start with a plus sign. Elements provided by domains should be qualified by the domain identifier.

In the scenario, specialization hierarchies include the keyword ancestor element provided by the base topic and the apiname element provided by the programming domain:

```
<!ATTLIST classname      class CDATA "+ topic/keyword pr-d/apiname cl-d/
classname ">
<!ATTLIST fieldname      class CDATA "+ topic/keyword pr-d/apiname cl-d/
fieldname ">
<!ATTLIST methodname     class CDATA "+ topic/keyword pr-d/apiname cl-d/
methodname ">
```

The complete element definition file would look as follows:

```
<!ENTITY % classname      "classname">
<!ENTITY % fieldname      "fieldname">
<!ENTITY % methodname     "methodname">

<!ELEMENT classname       (#PCDATA)>
<!ELEMENT fieldname       (#PCDATA)>
<!ELEMENT methodname      (#PCDATA)>

<!ATTLIST classname      class CDATA "+ topic/keyword pr-d/apiname cl-d/
classname ">
<!ATTLIST fieldname      class CDATA "+ topic/keyword pr-d/apiname cl-d/
fieldname ">
<!ATTLIST methodname     class CDATA "+ topic/keyword pr-d/apiname cl-d/
methodname ">
```

Writing the shell DTD

After creating the domain files, you can write shell DTDs to combine the domain with topics and other domains. The shell DTD must include all domain dependencies.

In the scenario, the shell DTD combines the class library domain with the concept, reference, and task topics and the programming domain. The portions specific to the class library domain are highlighted below in bold:

```
<!--vocabulary declarations-->
<!ENTITY % pr-d-dec PUBLIC "-//IBM//ENTITIES DITA Programming Domain//EN"
"programming-domain.ent">
%pr-d-dec;
<!ENTITY % cl-d-dec SYSTEM "classlib-domain.ent" %cl-d-dec;

<!--vocabulary substitution-->
<!ENTITY % pre           "pre           | %pr-d-pre;">
<!ENTITY % keyword       "keyword       | %pr-d-keyword; | %cl-d-apiname;">
<!ENTITY % ph            "ph            | %pr-d-ph;">
<!ENTITY % fig           "fig           | %pr-d-fig;">
<!ENTITY % dl            "dl            | %pr-d-dl;">
<!ENTITY % apiname "apiname | %cl-d-apiname;">

<!--vocabulary attributes-->
<ATTLIST concept         domains CDATA "&pr-d-att; &cl-d-att;">
<ATTLIST reference       domains CDATA "&pr-d-att; &cl-d-att;">
<ATTLIST task            domains CDATA "&pr-d-att; &cl-d-att;">

<!--Redefine the infotype entity to exclude other topic types-->
<!ENTITY % info-types    "concept | reference | task">

<!--Embed topic to get generic elements -->
<!ENTITY % topic-type PUBLIC "-//IBM//ELEMENTS DITA Topic//EN" "topic.mod">
%topic-type;
```

```
<!--Embed topic types to get specific topic structures-->
<!ENTITY % concept-typemod PUBLIC "-//IBM//ELEMENTS DITA Concept//EN"
"concept.mod">
%concept-typemod;
<!ENTITY % reference-typemod PUBLIC "-//IBM//ELEMENTS DITA Reference//EN"
"reference.mod">
%reference-typemod;
<!ENTITY % task-typemod PUBLIC "-//IBM//ELEMENTS DITA Task//EN" "task.mod">
%task-typemod;

<!--vocabulary definitions-->
<!ENTITY % pr-d-def PUBLIC "-//IBM//ELEMENTS DITA Programming Domain//EN"
"programming-domain.mod">
%pr-d-def;
<!ENTITY % cl-d-def SYSTEM "classlib-domain.mod"> %cl-d-def;
```

Notice that the class library phrases are added to the element entity for keyword as well as for apiname. This addition makes the class library phrases available within topic structures that allow keywords and not just in topic structures that explicitly allow API names. In fact, the structures of the reference topic specify only keywords, but it's good practice to add the domain specialization entities to all ancestor elements.

Considerations for domain specialization

When you define new types of topics or domain elements, remember that the hierarchies for topic specialization and domain specialization must be distinct. A specialized topic cannot use a domain element in a content model. Similarly, a domain element can specialize only from an element in the base topic or in another domain. That is, a topic and domain cannot have dependencies. To combine topics and domains, use a shell DTD.

When specializing elements with internal structure including the `ul`, `ol`, and `dl` lists as well as `table` and `simpletable`, you should specialize the entire content element. Creating special types of pieces of the internal structure independently of the whole content structure usually doesn't make much sense. For example, you usually want to create a special type of list instead of a special type of `li` list item for ordinary `ul` and `ol` lists.

You should never specialize from the elements of the highlight domain. These style elements do not have a specific semantic. Although the formatting of the highlight styles might seem convenient, you might find you need to change the formatting later.

As noted previously, you should use element entities instead of literal element names in content models. The element entities are necessary to permit domain specialization.

The content model should allow for the possibility that the element entity might expand to a list. When applying a modifier to the element entity, you should enclose the element entity in parentheses. Otherwise, the modifier will apply only to the last element if the entity expands to a list. Similar issues affect an element entity in a sequence:

```
..., ( %classname; ), ...
... ( %classname; )? ...

... ( %classname; )* ...
... ( %classname; )+ ...
... | %classname; | ...
```

The parentheses aren't needed if the element entity is already in a list.

Generalizing a domain

As with topics, a specialized content element can be generalized to one of its ancestor elements. In the previous scenario, a `classname` can generalize to `apiname` or even `keyword`. As a result, documents using different domains but the same topics can be exchanged or merged without having to generalize the topics.

To return to the highlight style controversy mentioned in [Understanding the base domains](#) on page 158, a pragmatic document authored with highlight domain will contain phrases like the following:

```
... the <b>important</b> point is ...
```

When the document is generalized to the same topic but without the highlight domain, the pragmatic `b` element becomes a purist `ph` element, indicating that the phrase is special without introducing presentation:

```
... the <ph class="+ topic/ph hi-d/b ">important</ph> point is ...
```

In the previous scenario, the class library authors could send their topics to another DITA shop without the class library domain. The recipients would generalize the class library topics, converting the `classname` elements to `apiname` base elements. After generalization, the recipients could edit and process the class, field, and method names in the same way as any other API names. That is, the situation would be the same as if the senders had decided not to distinguish class, field, and method names and, instead, had marked up these names as generic API names.

As an alternative, the recipients could decide to add the class library domain to their definitions. In this approach, the senders would provide not only their topics but also the entity declaration and element definition files for the domain. The recipients would add the class library domain to their shell DTD. The recipients could then work with `classname` elements without having to generalize.

The recipients can use additional domains with no impact on interoperability. That is, the shell DTD for the recipients could use more domains than the shell DTD for the senders without creating any need to modify the topics.



Note: When defining specializations, you should avoid introducing a dependency on special processing that lacks a graceful fallback to the processing for the base element. In the scenario, special processing for the `classname` element might generate a literal “class” label in the output to save some typing and produce consistent labels. After automated generalization, however, the label would not be supplied by the base processing for the `apiname` element. Thus, the dependency would require a special generalization transform to append the literal “class” label to `classname` elements in the source file.

Summary

Through topic specialization and domains, DITA provides the following benefits:

- Simpler topic design.

The document designer can focus on the structure of the topic without having to foresee every variety of content used within the structure.

- Simpler topic hierarchies.

The document designer can add new types of content without having to add new types of topics.

- Extensible content for existing topics.

The document designer can reuse existing types of topics with new types of content.

- Semantic precision.

Content elements with more specific semantics can be derived from existing elements and used freely within documents.

- Simpler element lists for authors.

The document designer can select domains to minimize the element set. Authors can learn the elements that are appropriate for the document instead of learning to disregard unneeded elements.

In short, the DITA domain feature provides for great flexibility in extending and reusing information types. The highlight, programming, and UI domains provided with the base DITA release are only the beginning of what can be accomplished.

Notices

© Copyright International Business Machines Corp., 2002, 2003. All rights reserved.

The information provided in this document has not been submitted to any formal IBM test and is distributed "AS IS," without warranty of any kind, either express or implied. The use of this information or the implementation of any of these techniques described in this document is the reader's responsibility and depends on the reader's ability to evaluate and integrate them into their operating environment. Readers attempting to adapt these techniques to their own environments do so at their own risk.

How to define a formal information architecture with DITA map domains

The benefits of formal information typing are well known for the content of topics, but collections of topics also benefit from formal organizing structure. Such formal structures guide authors while they assemble collections of topics and ensure consistent large-scale patterns of information for the user. Using DITA map domains, a designer can define a formal information architecture that can be reused in many deliverables.

This article explains the design technique for creating DITA map domains. As an example, the article walks through the definition for assembling a set of topics as a how-to. Such a how-to could be one reusable design component within an information architecture.

Formal information architecture

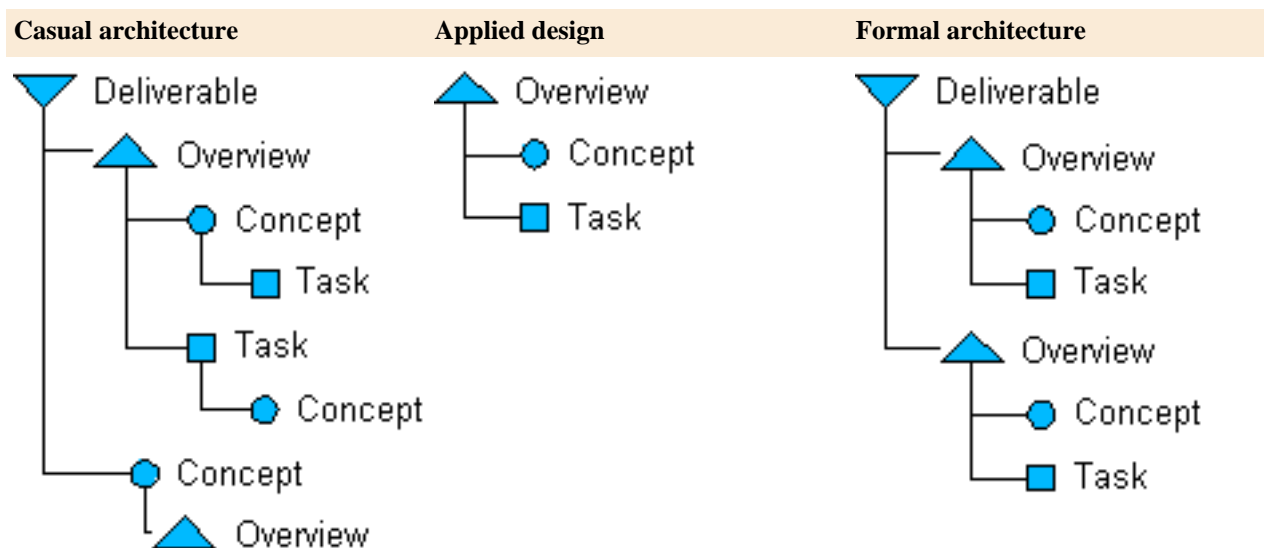
Information architecture can be summarized as the design discipline that organizes information and its navigation so an audience can acquire knowledge easily and efficiently. For instance, the information architecture of a web site often provides a hierarchy of web pages for drilling down from general to detailed information, different types of web pages for different purposes such as news and documentation, and so on.

An information architecture is subliminal when it works well. The lack of information architecture is glaring when it works poorly. The user cannot find information or, even worse, cannot recognize or assimilate information when by chance it is encountered. You probably have experience with websites that are poorly organized or uneven in their approach, so that conventions learned in part of the website have no application elsewhere. Extracting knowledge from such information resources is exhausting, and you quickly abandon the effort and seek the information elsewhere.

Currently, information architects work by defining the architecture through guidelines and instructions to the writer. A better approach is to formalize the architecture through an XML design that is validated by the XML editor or parser. This formal approach has the following benefits:

- Authors receive guidance from the markup while working.
- Information with the same purpose is consistent across deliverables.
- Information for a purpose is complete.
- Processing can rely on the structure of the information and operate on the declared semantics of the information.

The following drawings illustrate the gain in clarity and consistency by applying a design to produce a formal information architecture:



In short, the formal design acts as a kind of blueprint to be fulfilled by the writer.

Specializing topics and maps

DITA supports the definition of a formal information architecture through topics and map types. The topic type defines the information architecture within topics (the micro level) while the map type defines the information architecture across topics (the macro level).

The base topic and map types are general and flexible so they can accommodate a wide variety of readable information. You specialize these general types to define the restricted types required for your information architecture.

Topic

The topic type mandates the structure for the content of a topic. For instance, the DITA distribution includes a task type that mandates a list of steps as part of the topic content. This specialized topic type provides guidance to the author and ensures the consistency of all task topics. Processing can rely on this consistency and semantic precision. For instance, the processing for the task type could format the task steps as checkable boxes.

Map

The map type mandates the structure for a collection of topics. A map can define the navigation hierarchy for a help system or the sequence and nesting of topics in a book. For instance, the DITA distribution includes a bookmap demo that mandates a sequence of preface, chapter, and appendix roles for the top-level topics. This specialized map type ensures that the collection of topics conforms to a basic book structure.

Without formal types, the information architecture is defined only through editorial guidelines. Different authors may interpret or conform to the guidelines in varying degrees, resulting in inconsistency and unpredictability. By contrast, the formal types ensure that the design that can be repeated for many deliverables.

The how-to collection

One typical purpose for a collection of topics is explain how to accomplish a specific goal. A how-to assembles the relevant topics and arranges them in a typical sequence for one way to reach that goal. A standard design pattern for the how-to collection might consist of an introduction topic, some background concepts, some task and example topics, and a summary.

A help system or book might have several how-tos, for instance, on setting up web authentication, reading a database from a web application, and so on. Or, a web provider might publish an ongoing series of how-to articles on technical subjects. Thus, designing a formal how-to pattern would be useful so that all how-tos are consistent regardless of the writer.

Note that formalizing a collection doesn't prevent topic reuse but, instead, guides topic reuse so that appropriate types of topics are used at positions within the collection. For example, in the how-to, concept topics will appear only as background before the tasks rather than in the middle of the how-to.

Map specialization

Among the many capabilities added to maps by DITA 1.3 is specialization through map domains. Instead of packaging specializations of elements for topic content, however, you specialize elements for map content, typically the `topicref`. The specialized `topicref` element lets authors specify semantics or constraints on collections of topics. By packaging the `topicref` specializations as a map domain rather than as a map type, you can reuse the formal collection design in many different map types.

A specialized `topicref` can be used for the following purposes:

- To restrict the references to topics of a specialized type. For instance, a `conceptref` refers only to concept topics (including specialized concepts).
- To assign a topic to a role within a collection. For instance, the topic identified by a `summaryref` could provide the concluding explanation for a collection.
- To restrict the contents of the collection, requiring specific topic types or requiring topics to act in specific roles at specified positions within the collection.

Drawing on all of these capabilities, we can define a formal structure for a how-to collection.

Implementing a map domain

A map domain uses the same DTD design pattern as a topic domain. See [specializing domains](#) for the details on the domain design pattern, which aren't repeated here. Instead, this article summarizes the application of the domain DTD design pattern to maps.

1. Create a domain entities file to declare the elements extending the `topicref` element.
2. Create a domain definition module to define the elements including their element entities, content and attribute definitions, and the architectural class attribute.
3. Create a shell DTD that assembles the base map module and the domain entities file and definition module.
4. Create map collections from the shell DTD.

Declaring the map domain entities

The entities file for the how-to domain defines the `howto`, `conceptref`, `taskref`, and `exempleref` extensions for the `topicref` element and defines the how-to domain declaration for the domain attributes entity:

```
<!ENTITY % howto-d-topicref "howto">
<!ENTITY howto-d-att "(map howto-d)">
```

Defining the map domain module

The definition module for the how-to domain starts with the element entities so the new elements could, in turn, be extended by subsequent specializations. Of these new elements, only `howto` has been declared in the entities file because the other new elements should only appear in the child list of the `howto` element. (In fact, reference typing elements such as `conceptref` and `taskref` might also be defined in the entities file for reuse in other specialized child lists.)

```
<!ENTITY % howto "howto">
<!ENTITY % conceptref "conceptref">
<!ENTITY % taskref "taskref">
```



```
<!ENTITY % exempleref "exempleref">
<!ENTITY % summaryref "summaryref">
```

The definition module goes on to define the elements. The definition for the `howto` element restricts the content list for the collection to the metadata for the topic, references to any number of concept topics, references to task topics and optional example topics, and a topic acting in the role of a concluding summary. In addition, the `howto` element refers to the topic that provides an overview of the contents.

```
<!ELEMENT howto ((%topicmeta;)?, (%conceptref;)*, ((%taskref;),
  (%exempleref;)?)+,
  (%summaryref;))>
<!ATTLIST howto
  navtitle      CDATA      #IMPLIED
  id            ID         #IMPLIED
  href          CDATA      #IMPLIED
  keyref        CDATA      #IMPLIED
  query         CDATA      #IMPLIED
  conref        CDATA      #IMPLIED
  copy-to       CDATA      #IMPLIED
  %topicref-atts;
  %select-atts;>
```

The `conceptref` and `taskref` elements have a restricted type, meaning that validating processing is obligated to report an error if the referenced topic doesn't have the declared type (or a specialization from the declared type):

```
<!ELEMENT conceptref ((%topicmeta;)?, (%conceptref;)*)>
<!ATTLIST conceptref
  href          CDATA      #IMPLIED
  type          CDATA      "concept"
  ...>
<!ELEMENT taskref    ((%topicmeta;)?, (%taskref;)*)>
<!ATTLIST taskref
  href          CDATA      #IMPLIED
  type          CDATA      "task"
  ...>
```

The `exempleref` and `summaryref` elements don't restrict the type but, instead, assign roles to the referenced topics. Because the content list of the `howto` collection topic allows a topic to act as an example and requires a topic to act as a summary, the author is prompted to create topics in those roles, and the roles can be used in processing, for instance, to add a lead-in word to the emitted topic titles.

```
<!ELEMENT exempleref ((%topicmeta;)?, (%exempleref;)*)>
<!ATTLIST exempleref
  ...>
<!ELEMENT summaryref ((%topicmeta;)?>
<!ATTLIST summaryref
  ...>
```

On closer investigation, either or both of these particular roles may turn out to reflect a persistent topic structure or semantic, in which case it would be appropriate to define topic types and limit the corresponding `topicref` specialization to topics of those types. The general technique, however, of assigning a role to a topic in the context of a collection remains valid.

Finally, the definition module sets the class attribute to declare that the new elements derive from `topicref` and are provided by the `howto` package:

```
<!ATTLIST howto %global-atts;
  class CDATA "- map/topicref howto/howto ">
<!ATTLIST conceptref %global-atts;
  class CDATA "- map/topicref howto/conceptref ">
```

...

Assembling the shell DTD

As with topic domains, a shell DTD assembles the base map module with the entities file and definition module for the how-to domain:

```
<!--vocabulary declarations-->
<!ENTITY % howto-d-dec PUBLIC "-//IBM//ENTITIES DITA How To Map Domain//EN"
"howto.ent">
%howto-d-dec;
...

<!--vocabulary substitution (one for each extended base element,
with the names of the domains in which the extension was declared)-->
<!ENTITY % topicref "topicref | %mapgroup-d-topicref; | %howto-d-
topicref;">

<!--vocabulary attributes (must be declared ahead of the default definition)
-->
<!ENTITY included-domains "&mapgroup-d-att; &howto-d-att;">

<!--Embed map to get generic elements -->
<!ENTITY % map-type PUBLIC "-//IBM//Elements DITA Map//EN" "../..dtd/
map.mod">
%map-type;

<!--vocabulary definitions-->
...

<!ENTITY % howto-d-def PUBLIC "-//IBM//ELEMENTS DITA How To Map Domain//EN"
"howto.mod">
%howto-d-def;
```

Creating a collection with the domain

Using the shell DTD, a map could include one or more how-to collections, as in the following example:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE map PUBLIC "-//IBM//DTD DITA How To Map//EN"
"howtomap.dtd">
<map>
  <!-- how-to clusters can appear anywhere in a map hierarchy but always
follow a consistent information pattern within the how to -->
  <howto href="dita-mapdomains.xml">
    <conceptref href="informationArchitecture.xml"/>
    <conceptref href="mapBackground.xml"/>
    <conceptref href="formalCollection.xml"/>
    <conceptref href="mapSpecialization.xml"/>
    <taskref href="implementDomain.xml"/>
    <exempleref href="declareEntities.xml"/>
    <exempleref href="domainModule.xml"/>
    <exempleref href="assembledDTD.xml"/>
    <exempleref href="domainInstance.xml"/>
    <summaryref href="summary.xml"/>
  </howto>
</map>
```

In fact, this example is the map for the article that you're reading right now. That is, as you may well have noticed, this article conforms to the formal pattern for a how-to collection. Here's the list of topics in this how-to article but with the addition of the topic type or role and title:

- howto: How to define a formal information architecture with DITA map domains
 - concept: Formal information architecture
 - concept: Specializing topics and maps
 - concept: The how-to collection
 - concept: Map specialization
 - task: Implementing a map domain
 - example: Declaring the map domain entities
 - example: Defining the map domain module
 - example: Assembling the shell DTD
 - example: Creating a collection with the domain (this topic)
 - summary: Summary

While this article contains only a how-to collection, a how-to collection could be part of a larger deliverable. For instance, a help system could include multiple how-tos as part of a navigation hierarchy. Similarly, how-to collections could be used in books by creating a new shell DTD that combines the bookmap map type with the how-to map domain.

As you explore collection types, you'll find that, in addition to topics, a collection can aggregate smaller collections. For instance, you could create domains for a how-to collection, a case study collection, and a reference set collection. A product information collection could then require a product summary topic and at least one of each of these subordinate collections in that order.

You'll also find that, to represent a high-level relationship with a collection, you can create a relationship to the root topic for the collection branch. As the introduction and entry point for the collection, the root topic should provide the most statement of the content of the collection. That is, you can treat the set of topics as a collective content object, using the root topic to represent the collection as a whole for navigation and cross references.

Summary

In this article, you've learned how to specialize the `topicref` element to mandate a specific collection of topics. For complete, single-purpose collections such as functional specifications and quick reference guides, you might package these specialized `topicref` elements with a new map type. For building-block collections (such as how-tos or case studies) that can appear within a large deliverable, especially when different designers might create different collection types, you might want to package the specialized `topicref` elements as a map domain.

By specializing a DITA map in this way, you can implement a formal information architecture not just at the micro level within topics but at the macro level across topics. By defining such large-scale collective content objects, you can provide guidance to authors and declare semantics for processors with the end result that users have consistent and complete information deliverables.

