IBM

IBM Cúram Social Program Management

# Cúram DocMaker Internals

Version 6.0.4

**Note**

Before using this information and the product it supports, read the information in Notices at the back of this guide.

# Table of Contents

# Chapter 1

## Introduction

## 1.1 What is DocMaker?

DocMaker is a tool for generating standardized Cúram product documentation from content written in the DocBook XML format. it supports the rendering of DocBook XML documents to HTML Help and PDF formats.

## 1.2 What is DocBook?

DocBook is a set of tags for describing technical documentation. DocMaker supports the DocBook XML 4.2 open standard maintained by OASIS [http://www.oasis-open.org/]. It is similar to HTML in many ways, as both are "markup" languages. In markup languages, tags are inserted into the document text to describe the structure of the document and to indicate what the text represents (e.g., a paragraph, an emphasized phrase, a hyperlink, etc.). Here is an example of a simple DocBook XML document. It is a `book` that contains a `chapter` with one `section`.

```
<book>
  <title>My First Document</title>
  <chapter>
    <title>My First Chapter</title>
    <section>
      <title>My First Section</title>
      <para>
        This is the first paragraph of the first section of
        the first chapter of my new book. Text within a
        paragraph can be marked up further. For example,
        text can be <quote>quoted</quote> or
        <emphasis>emphasized</emphasis>.
      </para>
    </section>
  </chapter>
</book>
```

Example 1.1 A Simple DocBook Document

## 1.3 What's Wrong with the Way Things are Now?

In traditional documentation, the author selects a template and writes a document. If the template includes paragraph and character styles, most of the formatting will be taken care of by the documentation software. However, the author must remain concerned about the appearance of the document as well as its content. If the document is required in a number of different formats (e.g., HTML, PDF, and paper), much work is required to convert from the original document into the new formats. An author who is not well versed in the use of the template and the styles, may produce a document that is difficult to convert into another format automatically.

For example, if a program is required to automatically convert a document from a word-processor format into HTML, it may be difficult to identify different heading levels if the author just changed the font size of some text to create a heading. If the output must be split into separate HTML pages for each section of the document, it may not be possible to identify what text is included in a section. Careful use of the template's paragraph styles makes this process easier, but most documentation software does not enforce the use of the styles and they may be used in unconventional ways by individual authors, thus further complicating matters.

Another problem with word-processor formats is that most are closed formats. Unless the word-processing software comes with the ability to export files to HTML format, it may not be possible to do any automatic conversion. If there is an export capability, it may not produce output in the required style and may not be configurable to change its default behavior.

The core of these all problems is that a conversion program can have great difficulty determining what the content of the original document *means*. It can only guess at the meaning by examining what the document looks like.

## 1.4 How Does DocBook Solve These Problems?

The purpose of DocBook is to place the author's focus on what elements of the document *mean* and not how they should look. In Example 1.1, *A Simple DocBook Document*, the author has only indicated the structure of the document: a book with a chapter, sections, paragraphs, titles, etc.; some of the text is quoted and some is emphasized. There is no indication that the sections should be numbered, that the book's title should be on a separate page, that the paragraph should be spaced a certain distance from the section title, or that double or single quotation marks should be used. These are not the author's priorities. After all, the author is probably not a graphic designer or typographer skilled in these matters, rather a person with knowledge about the content of the document.

While authors may wonder why such emphasis is placed on structure and content and not on how the document looks, the benefits soon become apparent: once a document has been written and all its structure and content

clearly identified: it is possible to impose *any* look (or even feel or sound) required. A conversion program can examine the document and clearly identify its structure. Text can be examined and words that should be emphasized, quoted, or considered in other ways, can be handled appropriately.

When authors are finished defining the content of documents, they are finished with the documents. All further work is performed by processing the document automatically to produce the required output format or formats. This final step is called "rendering". The same document could be printed as a standard paper manual, provided in a "large print" format, distributed in an electronic format such as PDF or HTML, converted to speech by a computer, printed in Braille, or even produced in all these formats simultaneously. This is possible because the authors have clearly identified each element of the document. Rather than indicating that text should be in italics, they have indicated that it should be "emphasized". For a printed manual this may indeed result in the use of an italic typeface, but for an audio translation, it might mean use a higher-pitched or louder voice.

Using DocBook markup, the time-consuming process of document conversion and rendering can be entirely automated. As the format is open and in plain text, it is easy to integrate into source-code control systems and can be edited concurrently by multiple authors. The manual editing of repetitive documentation, such as lists of error messages, can be replaced by automatic document generation.

Authors are required to pay more attention to document structure and the meaning of the content, but are freed from the requirement to consider the formatting. The resulting flexibility and automation allowed by the DocBook XML format more than compensates for this increased "duty of care".

## 1.5 What is this Manual?

This manual aims to provide all the information necessary for authors to write documentation in DocBook XML format and to use DocMaker to render these documents into a suitable finished format. The focus of the manual is on the creation of Cúram product documentation; it includes guidelines on the selection and use of DocBook elements to impose the correct structure and style on these documents.

# Chapter 2

## Installing DocMaker

## 2.1  Introduction

DocMaker is released internally in Cúram in a different form to that re-
leased to customers. Customers use DocMaker via the Documentation Cen-
ter development environment; they do not install DocMaker as a separate
product. Within Cúram, DocMaker releases are made available to support
its integration with the Documentation Center or for use in automated build
scripts. Other internal users may just be interested in installing and running
DocMaker as a separate product for testing purposes. This chapter describes
the installation steps for the internal release of DocMaker and includes de-
tails on what it contains and how it can be integrated into build scripts and
other systems. This information is relevant to internal users only.

Some of the tools developed for DocMaker can also be used in other
projects. See Appendix A, *DocMaker Tools* for more details.

## 2.2  Installation

DocMaker is released as a simple ZIP file that expands to a `DocMaker`
folder containing all of the necessary files. When replacing an existing in-
stallation of DocMaker, the old installation should be deleted before ex-
panding the new ZIP file. The installation steps are as follows:

1.  Ensure that you have a Java Runtime Environment (JRE) or Java De-
    velopment Kit (JDK) with version 1.3 or higher installed.

2.  Obtain the DocMaker installation ZIP file.

3.  Extract the ZIP file to your hard disk. The DocMaker files will auto-
    matically be placed in a `DocMaker` folder.

4.  Set the `DOCMAKER_HOME` environment variable to point to your Doc-
    Maker installation folder. For example, if you installed DocMaker in

the `C:\DocMaker` folder, you should set the `DOCMAKER_HOME` environment variable to `C:\DocMaker`.

> **Important**
>
> When executing commands at a command prompt, it is usual to place quotation marks around the names of files and folders that contain space characters. However, when setting the `DOC-MAKER_HOME` environment variable, you *must not* include quotation marks around the name of the installation folder.

5.  Add DocMaker to your system path by editing the `PATH` system environment variable and adding `%DOCMAKER_HOME%\bin` to it.

The contents of a DocMaker release (with paths relative to the `DocMaker` folder) are shown in the table below. Not all of the files that are released are necessary for all operations and some should not be released externally, as they are not supported for use by customers.

| File or Folder | Description |
| --- | --- |
| `bin/` | Contains the Ant scripts and Windows batch scripts for executing DocMaker. |
| `bin/docmaker.bat` | The Windows shell script that executes DocMaker directly from the command-line. Where DocMaker is invoked directly from other scripts, this file is may not be necessary. |
| `bin/docmaker.xml` | The main DocMaker Ant script that defines all of the supported DocMaker targets. |
| `bin/dm-internal.xml` | The DocMaker Ant script that defines the internal, unsupported DocMaker targets, including plain HTML generation and *FrameMaker* file conversion. This should not be shipped to customers. |
| `bin/docmaker.properties` | An Ant properties file that defines some values that are used by the DocMaker Ant script. |
| `bin/wrapexit.bat` | A special Windows shell script that can be used to correctly detect errors if the `docmaker.bat` script is invoked from an Ant script. |
| `doc/` | The DocMaker documentation in PDF and HTML Help format. |
| `doc/`*format*`/internal/` | The *Cúram DocMaker Internals* manual covering all aspects of Doc- |

| File or Folder | Description |
|---|---|
| | Maker use including features only supported internally. |
| doc/*format*/manual/ | The *Cúram DocMaker Manual*, an abridged version of the DocMaker manual for distribution to customers as part of the Documentation Center development environment. |
| doc/help/images/ | The boilerplate images and icons that must be redistributed with any HTML Help version of the DocMaker manuals. |
| lib/ | All of the stylesheets and tools used by DocMaker. The files in this folder include the main stylesheets for pre-processing the DocBook content before rendering and the Java libraries for DocMaker Ant tasks and the HTML Help Applet. Files matching the pattern fm*.* should not be shipped to customers, as they are for *FrameMaker* file conversion only. |
| lib/docmaker.jar | The tools used by DocMaker when rendering documents. Some of the classes in this JAR file are also found in help-applet.jar, as they are also needed by the Help Applet. |
| lib/help-applet.jar | The Help Applet classes and the tools necessary to support the operation of the applet. To keep this JAR file small, a small subset of the tools found in docmaker.jar are included, but that JAR file is not required by the applet. |
| lib/help-server.jar | The Help Server classes used to support a server-side alternative to the Help Applet. Management of the table of contents, searching and context sensitive look-ups are performed by Java Servlets deployed in a J2EE web container rather than by the Help Applet. The Help Server shares much of its implementation with the Help Applet, so the applet JAR file must be available on the server. |

| File or Folder | Description |
|---|---|
| `lib/jde-commons-ext.jar` | A special build of the JDE Commons library that uses the Java 1.5 compiler to ensure that all timing facilities are included to enable accurate profiling (see Appendix A, *DocMaker Tools* for more details). This need not be shipped to customers. |
| `lib/profile.jar` | Special profiling classes used internally for performance metrics, etc. (see Appendix A, *DocMaker Tools* for more details). This need not be shipped to customers. |
| `lib/profile-ant.jar` | Special profiling classes used internally for performance metrics, etc. (see Appendix A, *DocMaker Tools* for more details). This need not be shipped to customers. |
| `lib/apache-ant/` | The Apache Ant Java-based build tool. DocMaker includes this for the benefit of users who do not have a typical Cúram development environment available. |
| `lib/ant-contrib/` | Additional Ant tasks from the Ant-Contrib project. |
| `lib/docbook-xml/` | The DocBook XML 4.2 DTD from OASIS used when validating content to avoid the need to download the DTD dynamically using the URL at the top of every DocBook XML file. |
| `lib/docbook-xsl/` | The DocBook XSL stylesheet distribution from the Open DocBook Repository. Most of the rendering to HTML Help and XSLFO format are performed by these stylesheets. |
| `lib/dtd-custom/` | A custom extension to the DocBook XML 4.2 DTD that permits the use of `xi:include` elements in DocBook content. |
| `lib/fonts/` | Font description file for the Trebuchet MS and Arial Narrow fonts that Apache FOP requires for PDF rendering. |
| `lib/fop/` | The Apache FOP library that converts XSLFO documents to PDF |

| File or Folder | Description |
|---|---|
| | documents. |
| `lib/images/` | The boilerplate icons, logos, and other images used when rendering documents. |
| `lib/lucene/` | The Apache Lucene library that performs the indexing and searching functions required by the Help Applet. |
| `lib/xalan-j/` | The Apache Xalan XSLT processor that performs XSLT stylesheet transformations. |
| `lib/xsl-custom/` | Contains the stylesheets developed to customize the Open DocBook Repository's stylesheets to produce output suitable for Cúram. |
| `lib/xsl-custom/common/` | Configuration setting, localized strings, and performance tweaks applied to the third-party stylesheets and used for all output formats. |
| `lib/xsl-custom/fo/` | Customizations for XSLFO output used in PDF generation. |
| `lib/xsl-custom/help/` | Customizations for HTML Help output used for online help. |
| `lib/xsl-custom/html/` | Unsupported customizations for plain HTML output. This need not be shipped externally. |
| `templates/` | Simple DocBook XML files that can be used as templates when creating new content. |
| `README.txt` | A text file indicating the location of the DocMaker manual within the installation. There is no need to ship this externally. |
| `VERSION.txt` | A text file containing only the version number of this release of DocMaker. This can be useful when build scripts need to test if the installed version number matches the expected version number. |

Table 2.1 Contents of a DocMaker Release

## 2.3  System Requirements

DocMaker is a combination of Ant scripts, Ant tasks, XSLT stylesheets, and other third-party libraries that are combined to implement a tool that can convert DocBook XML content to PDF documents and online HTML Help content. DocMaker is supplied with most of the tools that it depends on, but there are some external dependencies.

A Java Runtime Environment (JRE) version 1.3 or later is required to run DocMaker. While Ant will report a warning if it cannot find a Java compiler when it is executed with a JRE instead of a Java Development Kit (JDK), this warning can be ignored. DocMaker does not require any of the extra features of a JDK, though a JDK can be used instead of a JRE if desired.

DocMaker's Ant scripts are compatible with Apache Ant 1.6.2. DocMaker includes this version of Ant and the DocMaker shell scripts use this installed version when invoking DocMaker. However, if the DocMaker Ant scripts are invoked directly from other Ant scripts or shell scripts, then the version of Ant used must be compatible with the version required by DocMaker.

Spelling checks are performed by invoking GNU Aspell from the Ant scripts. The scripts assume that **aspell.exe** is on the system path and will fail if this is not the case. Most developer machines have Aspell installed and configured correctly as part of their ConMan installation. Custom deployments of DocMaker to other environments need to consider how Aspell should be installed for use with DocMaker. If spelling checks are not required, Aspell need not be installed. The spelling check operations cannot be executed on non-Windows platforms.

When generating PDF documents, DocMaker, through the Apache FOP library, requires font metric data for the fonts that it will use in the PDF. The metrics data for the "Trebuchet MS" and "Arial Narrow" fonts is supplied with DocMaker to complement the font metrics data supplied with FOP. However, these Microsoft fonts are not embedded in the PDF document, so the fonts cannot be displayed on machines that do not have these fonts installed (e.g., Unix machines). In these cases, the font metrics information will be used by *Adobe Acrobat Reader* (or other PDF viewer) to replace the preferred font with a suitable alternative font of exactly the same dimensions, so that formatting and layout are not affected.

The `help` target used to launch DocMaker's own online HTML help content is not compatible with execution on non-Windows platforms. The is no such restriction on the generation of online HTML help content. All of the targets are described in Chapter 3, *Using DocMaker*.

DocMaker requires 512MB of heap space to support the rendering of large PDF documents (up to 1,000 pages). Extremely large or complex documents may require extra memory and smaller documents can be rendered with much less. By default, the DocMaker shell script set the maximum Java heap space to 512MB.

## 2.4  Version Numbers

DocMaker releases use three-part version numbers: the major number, minor number, and patch number. These are changed under the following conditions (presented in reverse order):

**Patch Number**

The patch number is incremented for releases that fix bugs, correct documentation errata, or address other minor issues. No patch release will ever require changes to existing DocBook XML content. The patch number is reset when the minor number is incremented.

**Minor Number**

The minor number is incremented for releases that introduce new functionality, make significant changes to the rendered output, or include significant changes to the DocMaker documentation. The same types of changes that are included in a patch release may also be covered under a minor release. No minor release will ever require changes to existing valid DocBook XML content; however, optional changes may be required to exploit some new functionality. Minor releases may also include new rules in the `check` target, but these are likely to fail only for content that is already considered unacceptable or to enforce existing conventions that have not been enforced automatically thus far. The minor number is reset when the major number is incremented.

**Major Number**

The major number is incremented when a release requires existing valid DocBook XML content to be changed or when major changes to functionality or features are made that may impact the integration of Doc-Maker with other tools. Due to the nature of DocBook XML content, the former condition is unlikely to occur and every effort will be made to introduce new features in a backward-compatible manner. Should content changes be required, an upgrade tool will probably be provided. The latter condition is more likely, but its impact is generally confined to a small number of integration points where DocMaker is used within other build systems. In certain certain circumstances, a release that meets the conditions for a minor number increase may be released as a major release if the additional functionality is considered of major importance, even if it has no impact on existing content.

## 2.5 Build Script Integration

There are a two ways to integrate DocMaker into another Ant build script: by executing a Windows **cmd.exe** shell to invoke the `docmaker.bat` file, or by invoking the DocMaker Ant script directly. The choice of method may be influenced by a number of factors.

You might execute the shell script if you want to isolate the DocMaker build from the JVM that is used for your Ant script or if you invoke DocMaker from another shell script. DocMaker builds (particularly for PDF) have significant memory requirements that might be difficult to manage if the Ant

script is invoked directly from another Ant script. This might also be necessary if the version of Ant being used for the main build scripts is not compatible with DocMaker Ant scripts (see Section 2.3, *System Requirements* above). Incompatible versions of the other tools and libraries that DocMaker uses could also be introduced unwittingly if the DocMaker Ant scripts are invoked directly from another Ant script.

You might execute the DocMaker Ant scripts directly if you are confident that the tools and libraries on the classpath are compatible with DocMaker's own and that DocMaker's memory requirements can be satisfied while running in the same JVM as other Ant targets. This can also be convenient on non-Windows platforms, as DocMaker is supplied only with Windows shell scripts.

The Windows shell may be unique in that it does not return the exit code of the last command executed as its own exit code. If a command fails, the shell exits with a successful exit code. An Ant script that invokes the DocMaker shell scripts might look like this (ignoring folders and paths for now):

```
<target name="run-docmaker">
  <exec executable="cmd.exe" failonerror="true">
    <arg line="/c &quot;docmaker.bat&quot; generate-pdf" />
  </exec>
</target>
```

Example 2.1 Ant Script Invoking DocMaker (Wrong)

However, if `docmaker.bat` fails, Ant will not fail, as **cmd.exe** will not return an error code. To compensate for this, DocMaker comes with another script called wrapexit.bat that ensures that the error code is propagated correctly. This script invokes the `docmaker.bat` script and then sets the **cmd.exe** return code to the correct error code using the **exit** command. The main DocMaker script does not do this by default, as executing the **exit** command to return the correct error code would close the command window if run interactively.

Before learning the correct ways to invoke DocMaker from an Ant script, you need to understand some of the environment settings that DocMaker requires. There are three folder locations that must be specified: the Ant base folder, the DocMaker home (installation) folder and the DocMaker output folder. The Ant base folder controls where DocMaker looks for the DocBook XML content files that will be processed. Each book to be processed should have a file called `book.xml` and each `book.xml` file should be in a separate folder exactly one folder level below the Ant base folder. The DocMaker home folder is requires for DocMaker to locale the stylesheets, libraries, and other resources that it requires for execution. The DocMaker build output folder is the location into which all generated output will be placed.

DocMaker's home folder must be set for DocMaker to be executed. This is the DocMaker installation folder that is called `DocMaker` by default when the ZIP file is expanded. The full path to this folder must be specified using

either the `DOCMAKER_HOME` environment variable or the `docmaker.dir` Ant property.

By default, the Ant base folder is set to the folder from which DocMaker is executed; this is the standard Ant convention. When executing DocMaker via its shell script, this can be overridden by setting the `DOC-BOOK_SRC_DIR` environment variable. When executing DocMaker's Ant script directly, Ant's `basedir` property will override the default value.

💡 Command-line Arguments

The `docmaker.bat` script sets a number of environment variables and Ant options, but passes all command-line arguments straight to Ant. Any command-line arguments supported by Ant can be used, including tracing, property setting, multiple targets, and project help.

All of the generated output is placed below the output folder that, by default, is called `doc` and created within the base folder. Because this is the default value, it is not safe to create a `book.xml` content file in a folder called `doc` within the base folder. The output folder location can be overridden by setting the `DOCMAKER_OUT_DIR` environment variable or the `out.dir` Ant property. When creating the output folder, DocMaker creates a sub-folder called `dm-mark` to mark this as DocMaker's output folder. When the `clean` target is called to delete the output folder, DocMaker will not delete the output folder or its contents unless it contains this marker folder, as it is possible that the folder may be a `doc` folder that was not created by Doc-Maker and is not safe to delete. The `force-clean` target will override this safety check and always delete the output folder.

| Environment Variable | Ant Property | Description |
| --- | --- | --- |
| DOCMAKER_HOME | docmaker.dir | The path to the Doc-Maker installation. Must be set. |
| DOCMAKER_OUT_DIR | out.dir | The path to the folder used for all generated output. Defaults to `doc` (relative to the base/input folder). |
| DOCBOOK_SRC_DIR | basedir | The path to the folder containing the folders in which the DocBook XML content for each book is located. By default, the shell script's current folder or Ant's base folder will be used. |

Table 2.2 Environment Variables and Ant Properties

> **!** Overriding the Output Folder
>
> When specifying an alternative output folder, some care is required to avoid situations where DocMaker may confuse output files with input files and rebuild its own output recursively. If the specified output folder is below the document root folder, it must be specified using a path that is relative to the document root folder. If the folder is outside the document root folder, a relative or absolute path may be used. The reason for this is that DocMaker avoids processing its own output by excluding the output folder using the path to the folder as a pattern and resolving this pattern relative to the document root folder. If the output folder path is an absolute path, then the pattern matching will fail. This is not a problem if the output folder is not below the document root folder, as it would not be included anyway; however, if it is within the document root folder, the pattern must match or there will be problems. As there is little reason why you would want to override the output folder to another location within the document root folder, this minor requirement that the path be relative in that case should not arise in the normal course of customizing DocMaker builds.

Using these environment variables and property settings, here are two ways to invoke DocMaker from an Ant script (only the DocMaker home folder setting is required, but all are shown for completeness):

```
<target name="run-docmaker-shell">
  <exec executable="cmd.exe" failonerror="true">
    <arg value="/c" />
    <arg value="${my.docmaker.dir}\bin\wrapexit.bat" />
    <arg value="generate-pdf" />
    <env key="DOCMAKER_HOME"    value="${my.docmaker.dir}" />
    <env key="DOCMAKER_OUT_DIR" value="${my.out.dir}" />
    <env key="DOCBOOK_SRC_DIR"  value="${my.src.dir}" />
  </exec>
</target>

<target name="run-docmaker-ant">
  <ant antfile="${my.docmaker.dir}/bin/docmaker.xml"
      target="generate-pdf" inheritall="false">
    <property name="docmaker.dir" value="${my.docmaker.dir}" />
    <property name="out.dir"      value="${my.out.dir}" />
    <property name="basedir"      value="${my.src.dir}" />
  </ant>
</target>
```

Example 2.2 Ant Script Invoking DocMaker

When invoking the shell script, the ANT_OPTS environment variable can be used to pass additional JVM arguments. By default, if the variable is not set, it will be set to -Xmx512m (maximum heap space of 512MB). This option can be overridden, and additional options set, by setting the ANT_OPTS variable appropriately. The ANT_ARGS environment variable is set by the DocMaker shell script and any existing value will be ignore. Additional Ant arguments should be passed via extra command-line arguments to the script. The DocMaker script uses the values of the environment variables to set the corresponding Ant properties in the ANT_ARGS variable, so the effect of setting these properties on the command-line of the script is indeterminate.

When setting the environment variables, do not use any double quotation marks around folder names that contain space characters, as this is already done by the `docmaker.bat` script and extra quotation marks will cause problems.

# Chapter 3

## Using DocMaker

## 3.1 Introduction

DocMaker is a command-line tool that is very easy to use. DocMaker requires that you follow simple patterns to organize and name DocBook XML content files and, as a result, no configuration files or cryptic command-line arguments are necessary. Once you learn what patterns must be followed, you will learn how to run DocMaker to process your documentation.

## 3.2 Folder Structure Patterns

DocMaker can render your DocBook XML documents into a number of formats, but it must first be able to locate that content and identify the purpose of each file. A rigid folder structure is imposed and it must be followed for DocMaker to work correctly.

DocMaker only supports one style of document: a book. Each book can have chapters and appendices and these can be subdivided into sections. Depending on the output format chosen, DocMaker will manage the creation of a table of contents and the addition of cover pages, logos, search indexes and other boilerplate content. DocMaker will render all of the books it finds in a single invocation, so they must be organized in a manner that lets DocMaker identify the content that belongs to each book.

All books in a group of books that are to be rendered together are located below a folder called the *document root folder*. This folder can be located anywhere and given any name. You can create more than one document root folder if you wish to maintain several separate book groups, each is entirely independent of the other. For example, you could use `C:\Books` as your document root folder.

The document root folder must contain one sub-folder for each book in the group. These are called the *book folders*. Even if you only have one book, it must still be located in a book folder. For example, you might place your

user manual in the `C:\Books\UserManual` folder. You can have as many of these book folders as you require, one for each book. You can use any folder names you want except the name "doc", as DocMaker will create a `doc` folder within the document root folder to contain all of the rendered output and this would overwrite your book folder of the same name.

## 3.3  File Naming Patterns

Inside each book folder, the DocBook XML file that contains the `book` root element must be called `book.xml`. This is called the *book file*. You can create as many other files and folders within the book folder as you want, but only the book file will be processed by DocMaker. A book file must not be placed in a sub-folder of a book folder: it must be contained directly within the book folder. This restriction does not apply to other content files, such as those you might use for chapters and sections.

You can use *XML inclusions* to organize the content of a book into multiple files. These content files must conform to a simple naming pattern. *Chapter files* containing `chapter` root elements should be named `c_*.xml`; *appendix files* containing `appendix` root elements, `a_*.xml`; and *section files* containing `section` root elements, `s_*.xml`. You can substitute any name for the "*" character in these patterns, but it is probably better not to use names like `c_chapter-1.xml` or `s_section-3-7.xml`, as the reorganization of content (simply by reorganizing the XML inclusions or by inserting new content) would make the numbered file names obsolete. A name like `c_introduction.xml` or `s_configuration.xml` might be a better choice, but you are free to use your own names once the prefix is used correctly.

An example of a book file that uses XML inclusions to include two chapter files is shown below.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE book
  PUBLIC "-//OASIS//DTD DocBook XML V4.2//EN"
  "http://www.oasis-open.org/docbook/xml/4.2/docbookx.dtd">
<book>

  <title>My First Book</title>

  <xi:include href="c_introduction.xml"
          xmlns:xi="http://www.w3.org/2003/XInclude"/>

  <xi:include href="c_main-topic.xml"
          xmlns:xi="http://www.w3.org/2003/XInclude"/>

</book>
```

Example 3.1 Sample Book Using XML Inclusions

The example shows the correct `DOCTYPE` declaration for a DocBook XML document. This should appear at the top of all book files. The `xi:include` elements use the `href` attribute to specify the location of the chapter files relative to the book file. In this example, they are located in the same book folder, so no path is required. They could, if desired, be loc-

ated in sub-folders of the book folder. The namespace declaration on these elements is also required. The individual chapter files are written like in the example below and have a similar DOCTYPE, but with a different root element specified. Note that without the DOCTYPE, the files cannot be validated effectively for conformance with the DocBook XML standard.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE chapter
  PUBLIC "-//OASIS//DTD DocBook XML V4.2//EN"
  "http://www.oasis-open.org/docbook/xml/4.2/docbookx.dtd">
<chapter>
  <title>Introduction</title>

  <section>
    <title>About This Book</title>

    <para>This book is....</para>
  </section>
</chapter>
```

Example 3.2 Sample Chapter File

A chapter file, or other content file, must have a single root element or it will not be a well-formed XML file and will cause errors during processing. You can only have one chapter root element in a chapter file, and likewise for appendices and sections.

> **Support for XML Inclusions**
>
> XML inclusions can, in theory, allow book files to include chapter files and chapter files to include section files and section files to include section files that become sub-sections. However, there are limitations in the current version of the Apache Xerces parser used by DocMaker that cause XML inclusions to fail under certain circumstances. Limiting their use to including content directly with the relevant root element of the document and not using more than two levels of inclusions will avoid most of these limitations until improved support for XML inclusions in the Apache Xerces parser becomes available.

If your document includes resource files such as images, these can be located in your book folder; in a sub-folder within your book folder; or even in another sub-folder of the document root folder, if they are shared between books. For the HTML Help output format, resources like images need to be distributed as separate files along with the HTML files. Therefore, DocMaker copies these resources to the relevant output folders, but does not copy DocBook XML content files, as you do not need (and probably do not want) to distribute these. DocMaker distinguishes between content files and resource files solely based on the naming convention described. If the file name does not match a book file name or other content file name pattern, the file is not copied.

The references you make to resource files should be specified relative to your book.xml file, not to the file that is referring to the resource. For example, if you create an image file C:\Books\UserManual\img1.gif, your reference to that image

from `C:\Books\UserManual\book.xml` is just " `img1.gif` ". If you create `C:\Books\UserManual\img\img2.gif`, the reference will be " `img/img2.gif` ". However, if the content for a chapter is stored in `C:\Books\UserManual\chap\c_intro.xml` and it includes the latter image, the reference will also be " `img/img2.gif` ", *not* " `../img/img2.gif` ", as it must be relative to the book file, not the chapter content file. If you create an image and want to share it between two books, you can copy it to `C:\Books\UserManual\img\img3.gif` and reference it as " `../img/img3.gif` ". The reason for this is that the XML inclusions are resolved before the resource references are resolved: all of the content files are combined into a single book file during processing and the references must be relative to that book file.

> **ℹ File Separator Characters**
>
> Although path names on your system (if you are using a Microsoft Windows system) are normally specified with backslash characters ("\") separating elements of the path to the file, the references in DocBook XML are universal resource indicators (URIs) and the solidus ("/") character must be used instead.

The following chapters will describe how to write your DocBook XML documents and how you might use resource files, such as images, but first you will learn how to run DocMaker.

## 3.4 Running DocMaker

Running DocMaker could not be easier, you simply open a command prompt, change to the document root folder, and run the **docmaker** command. DocMaker is based on the Apache Ant build tool, the same tool used for other Cúram build tasks and the operations it performs are invoked by specifying *targets* as command-line arguments to the **docmaker** command. The targets include operations such as spell-checking, PDF generation, and the cleaning of generated output files. By default, the **docmaker** command will invoke a target that displays some version information and an overview of its usage. To display a list of the available targets, run DocMaker with the `-projecthelp` or `-p` option as follows:

```
docmaker -projecthelp
```

A brief description of all the targets will be displayed. The available targets are shown in the table below.

| Target | Description |
| --- | --- |
| usage | Prints the usage information for the **docmaker** command. |
| version | Prints the DocMaker version number and installation folder location and the version and location of Apache Ant and Java that will be used. |

| Target | Description |
|---|---|
| clean | Removes the generated output folder and all of its contents. By default, this is the doc folder within the document root folder. |
| force-clean | If the output folder does not contain a folder named dm-mark, DocMaker will refuse to delete it when the clean target is invoked. This is to protect against the deletion of the incorrect folder by mistake. This target will remove the output folder even if the marker folder is not present. |
| help | On Microsoft Windows systems, this opens the HTML Help version of this DocMaker manual in a web browser. |
| generate-pdf | Generates a PDF document for each book. |
| generate-help | Generates an online HTML Help system for each book and also a single HTML Help system that combines all of the books. |
| generate-fo | Generates intermediate XSLFO output that is used to produce PDF documents. |
| generate-all | Generates both PDF and HTML Help output. |
| spell | Runs the GNU Aspell interactive spell-checking application if it is installed. |
| autospell | Runs the GNU Aspell non-interactive spell-checking application if it is installed. |
| check | Validates the DocBook XML content for conformance with the DocBook XML standard and for conformance with additional rules imposed for Cúram documentation. |
| new-simple-book | Copies a template for a simple DocBook XML book file to the current folder. The book file does not use XML inclusions. |
| new-book | Copies templates for a DocBook XML book file and chapter file to the current folder. The book file uses an XML inclusion to include the chapter file. |
| new-chapter | Copies a template for a DocBook XML chapter file to the current folder. |

Table 3.1 DocMaker Targets

To run a target, type **docmaker** followed by the name of target. For example, to generate PDF file, you would enter this command:

```
docmaker generate-pdf
```

You can run more than one target at a time by entering the target names on the command-line after the **docmaker** command and separating the names by spaces. Some targets, such as `generate-all`, will automatically run several other targets for you.

As DocMaker runs, it displays information on the tasks it is performing to achieve its target. Some of these tasks produce quite a lot of output, but all you need to observe are the last few lines of output. If the target was achieved, you will see the text "BUILD SUCCESSFUL" followed by a line displaying the time taken to complete the tasks. If the target could not be achieved, you will see "BUILD FAILED" displayed followed by an error message and the time taken. The error message is usually a summary of the problems encountered by the last unsuccessful task; you can find out more detail by reading back through the detailed messages to locate the source of the problem. Typical problems include invalid document structure, misspelled words, unresolved cross-references, etc. Once you correct the problem, you can execute the target again.

Files created by DocMaker will be placed in the `doc` folder within your documentation folder by default. This folder will be created if it does not exist. Within the `doc` folder, folders are created with names matching the name of your chosen rendering format. Inside each *format folder*, you will find copies of your book folders containing the output files in the new format.

The rest of this section will introduce the different DocMaker targets by leading you through examples of their use. The examples all assume that the document root folder is `C:\Books` and that the book folder is `C:\Books\UserManual`. The example commands are shown with the command prompt indicating the folder from which they should be executed. You do not type the folder name, only the **docmaker** command and its options.

> ### Running DocMaker from the Wrong Folder
>
> The most common mistake made by new users of DocMaker is running the **docmaker** command from the wrong folder. All of the common documentation tasks, other than the creation of new content files from the DocMaker content templates, are performed by running the command from the document root folder, *not the book folder*. This is clearly shown in the examples below.
>
> If you try to generate output files by running DocMaker from a book folder, you will probably get errors reporting that images or other resources cannot be found. If you then run DocMaker from the document root folder without first deleting the `doc` folder that was incorrectly created within the book folder, you will get countless error messages. These errors are caused when DocMaker searches for `book.xml` files and finds `book.xml` files among the temporary files created in the erroneous `doc` folder. DocMaker automatically ignores the `doc` folder located correctly within the document root folder, but any others must be removed manually.

The simplest way to begin writing a book is to use a book template. There are two book templates available: a simple book file that contains a legal notice and two skeleton chapters, and a compound book where the two skeleton chapters are in separate chapter files and are included into the book file. The templates are used by running one of the following commands:

```
C:\Books\UserManual> docmaker new-book
```

```
C:\Books\UserManual>        docmaker        new-
simple-book
```

The former command will create a compound book with three files. The first, `book.xml`, contains the book title, the legal notice, and references to the two chapter files. The file `c_introduction.xml` contains an outline introduction chapter, and `c_topic.xml` contains an outline for a chapter in the body of the book. You can rename the chapter files, but, if you do, you must update the references in the XML inclusions in the book file.

The latter command will create only the `book.xml` file and the content of the chapters appear directly within that file.

For both templates, you will probably want to start by changing the book title in the `book.xml` file. After that, you can make whatever other changes you require.

If you have an existing book and you want to add another chapter to it, you can run the command below to create a new `c_topic.xml` file. You will have to add the reference to this file to your book manually.

```
C:\Books\UserManual> docmaker new-chapter
```

Now that you have a book, you can edit it and then render it to your preferred format. DocBook renders all books in your project at the same time, so you run the rendering targets from the root folder of your documentation project. Never run the rendering targets from within a book folder. To render your books to PDF format, run this command:

```
C:\Books> docmaker generate-pdf
```

When DocMaker has finished, you will find your PDF book in the file `C:\Books\doc\pdf\UserManual\book.pdf`.

If you want to render your documentation to a HTML Help format that uses a Java applet to provide a table of contents and a search function, run this command:

```
C:\Books> docmaker generate-help
```

When DocMaker has finished, you will find your main HTML file will be `C:\Books\doc\help\UserManual\index.html`. This file is a HTML frame-set that adds the Help Applet that controls the access to your generated documentation. There may be a large number of other files, as HTML is not a self-contained format like PDF. If you want to distribute your HTML Help content, you must distribute all the files. In addition, Doc-Maker will place some shared files in the `C:\Books\doc\help\images` folder. This will also need to be distributed and must remain in the same location relative to your HTML Help files, i.e., in `../images` relative to your `index.html` file. DocMaker will also create a single HTML Help system that combines all of your books, so that a single table of contents and search function can be used for all. The main frame-set file for this will be `C:\Books\doc\help\index.html`. This file will be created even if you only have one book.

DocMaker can also perform some validation checks on your books to test compliance with Cúram documentation conventions. For example, the proper location of block elements and the length of lines in program listings will be checked. To run the validation checks, enter the following command:

```
C:\Books> docmaker check
```

DocMaker first validates the books against the DocBook XML document type definition (DTD), only when the books are valid will the additional rules be checked.

As DocMaker runs, errors and warnings will be displayed if rules are broken. When it has finished, you will find a copy of these messages in the file `C:\Books\doc\check\UserManual\book.txt`. The messages do not refer to errors in the content by file name and line number, as the rules are run against a version of the content that has been assembled into a single file in the output folder. However, the messages describe the locations of the errors in the content using the titles of books, chapters, sections, etc., to guide you to the invalid content.

If you have the *GNU Aspell* application installed on your computer (and accessible on the system path), DocMaker can use it to spell-check the content of all of your books. DocMaker will check all DocBook XML content files that follow the file naming patterns for spelling errors; resource files will not be checked. A supplementary word-list file is used for words that are valid in your books but that are not in the main dictionary. This file, `word-list.txt`, will be created (or reused, if it is present) in the document root folder and is shared for all books. You can edit it directly, or just let *Aspell* add new words. Take care if adding a word to the end of this word list using a text editor: *Aspell* requires a new-line character after the last word, or it will not be recognized as a word and will be removed from the word list without warning. To run the spelling check, enter the following command:

```
C:\Books> docmaker spell
```

DocMaker will open a new window for each file it finds and then start *Aspell* to let you correct the spellings automatically. If there are no spelling errors, the window will be closed automatically and the next file will be processed. Refer to your GNU Aspell documentation to learn how to use the Aspell application.

Although primarily for use from other automated build scripts, you can execute the spelling check non-interactively. Non-interactive checking will check the content files and stop when the first content file containing spelling errors is encountered. The list of unrecognized words will be displayed. To run the non-interactive spelling check, enter the following command:

```
C:\Books> docmaker autospell
```

DocMaker will open a new window for each file it finds and then start *Aspell* to let you correct the spellings automatically. If there are no spelling errors, the window will be closed automatically and the next file will be processed. Refer to your GNU Aspell documentation to learn how to use the Aspell application.

If you want to clean up your documentation folder by removing all the generated files, run this command:

```
C:\Books> docmaker clean
```

This will delete the `doc` folder and all its contents. This is why it is important not to give one of your book folders this name. DocMaker does, however, try to be careful when it is deleting this folder. When creating files, it will also create a folder within the `doc` folder called `dm-mark`; this is used to mark the `doc` folder as one created by DocMaker. If the `dm-mark` folder is not present, DocMaker will not delete the `doc` folder and will display a warning message. If you are sure you want to proceed, run this command:

```
C:\Books> docmaker force-clean
```

This user manual is available in PDF and HTML Help format and installed with DocMaker. You can access the online HTML Help by running this command:

```
C:\Books> docmaker help
```

On Microsoft Windows systems, this will open the main help page in your web browser.

You can combine several targets at once to perform a sequence of operations. A typical sequence might be executed as follows:

```
C:\Books> docmaker check spell generate-
pdf
```

## 3.5 Dependency Checking

Before rendering documents, DocMaker performs some dependency checking to ensure that the documents need to be rendered again. There is no need, for example, to render a document again if the content has not changed. DocMaker detects these changes by comparing the modification time stamps of your content files to the modification time stamps of the output files created when the content was last rendered. This can save considerable time when you are working on one book in a large group of books. The first time you perform a rendering operation such as `generate-pdf`, all of the PDF documents are created for all of the books. If you change one book and run `generate-pdf` again, only the book you changed will be rendered to a new PDF document, as all of the other PDF documents are already up to date.

This dependency checking is limited in two ways: dependencies on resource files are not calculated, and dependencies caused by XML inclusions are only calculated from the book files. If your book file includes a chapter file and you change the chapter file, DocMaker will render your book again, as the content has probably changed. However, if a book file includes a chapter file and that chapter file includes a section file, a change to the section file will not be detected and DocMaker will not render the book again. Similarly, changes to resources such as image files are not detected. Most Cúram books include few images and are organized as book files that include only chapter files, so this limitation is not too severe.

If you find that DocMaker has not detected a change and has not rendered a document as expected, then you can run the `clean` target to remove the old generated output and run the rendering process again, or you can update the time stamp on the relevant book file. If you have a large number of books in a group, the latter option will save you time. You may have a utility like the **touch** command available to update the time stamp of the book file, or you can just open the book file in an editor and save it again. Some editors do not save the file if you have not changed it, so making a simple change and then reversing that change (for example, typing a character and then deleting it again) before saving the file is usually sufficient to update the modification time stamp. Once this is done, DocMaker should detect the change and render your document again.

# Chapter 4

## The Document Hierarchy

## 4.1 Introduction

To write a structured document, you first need to know what elements are used to describe the document's *hierarchy*. This chapter will describe the hierarchy of a typical DocBook XML document and show you how and when to use the hierarchy elements.

## 4.2 Hierarchy Elements

The hierarchy elements organize a document into a logical structure. Typically, you will use the `book` element as the *root element* of your document. Within a `book`, the elements `part`, `chapter`, and `appendix` are used to define the general structure of the document. The `chapter` and `appendix` elements can be further sub-divided using the `section` element.

> **The Root Element**
>
> All DocBook XML documents are required to contain a single element in which all other elements are contained. This element is called the "root element". A document that contains more than one root element is invalid.

All the hierarchy elements that you will use for Cúram documentation have some features in common. Each element must contain a `title` element that contains the title text for that element. A `title` element placed within a `book` element defines the book title, a `title` element within a `chapter` defines the chapter title, etc. The `title` element should be the first element to appear within a hierarchy element.

The relationships between hierarchy elements are defined by containment. A `chapter` element within a `book` element is considered a chapter of that book. A `chapter` element could also be placed in a `part` element and the `part` element placed in the `book` element. This is all self explanatory until

section elements are placed within section elements. The nesting of section elements is covered below.

## 4.3 Books

### 4.3.1 Book Structure

A book is a group of parts, chapters, and appendices that together form a document. While there are other types of document in DocBook XML such as an article, Cúram documentation uses the book element exclusively.

```
<book>
  <title>My First Book</title>
  <bookinfo>
    ...
  </bookinfo>
  <chapter>
    ...
  </chapter>
  <chapter>
    ...
  </chapter>
</book>
```

In the above example, the book has a title, some information about the book element, and two chapters. The title element should appear first, followed by the bookinfo element, and then the other hierarchy elements.

### 4.3.2 Book Information

The bookinfo element is used to hold important information about the book. This information includes the author information, legal notices, copyright statements, and the printing history. There are many different elements that can be included here, but in Cúram documents the contents are quite formulaic. The example below shows the standard information for a Cúram product guide.

```
<bookinfo>

  <copyright>
    <year>2005</year>
    <holder>Cúram Software Limited</holder>
  </copyright>

  <productname>
    <trademark class="registered">Cúram</trademark> 4.0
  </productname>

  <abstract>
    <para>
      A short description of the contents of this book.
    <para>
  </abstract>

  <legalnotice>
    <para>
      No part of this publication may be stored in a retrieval
      system, transmitted, or reproduced in any way, including
      but not limited to photocopy, photograph, magnetic or
```

```
        other record, without the prior agreement and written
        permission of Cúram Software Ltd.
      </para>
      <para>
        The information in this book is distributed on an
        <quote>as is</quote> basis, without warranty. While every
        precaution has been taken in the preparation of this book,
        neither the authors nor Cúram Software Ltd. shall have any
        liability to any person or entity with respect to any
        liability, loss or damage caused or alleged to be caused
        directly or indirectly by instructions contained in this
        book or by the computer software or hardware products
        described herein.
      </para>
      <para>
        Cúram is a registered trademark of Cúram Software Ltd.
        All Cúram products and service names are trademarks and
        property of Cúram Software Ltd. All Rights Reserved. All
        third-party logos, products and service names are
        property of their respective owners.
      </para>
    </legalnotice>

</bookinfo>
```

In general, you will use the `bookinfo` element as shown changing only the copyright year as appropriate.

The `abstract` element is optional and allows a short description of the book to be included. This will not be rendered in the PDF output, but the HTML Help output will display the abstract on the cover page for the book.

### 4.3.3  Book Categories

Cúram books usually fall into one of a number of categories. DocMaker recognizes several categories and will customize the cover page of the book as appropriate when generating PDF output. To indicate the category to which a book belongs, you need to set the `role` attribute of the book element as described in the table below.

| Category of Book | Value of **role** Attribute |
| --- | --- |
| User Guides | user |
| Developer Guides | developer |
| Administrator Guides | administrator |
| Business Analyst Guides | analyst |
| Business Guides | business |
| Functional Specifications | functional |
| Design Specifications | design |

If you do not specify a `role` attribute for a book, then a plain cover page will be generated.The example below shows how to indicate that a book is in the "User Guides" category.

```
<book role="user">
  <title>How to Read a Manual</title>
```

```
  ...
</book>
```

## 4.3.4 Revision Histories

A revision history is a record of the changes made to a document over time. While most documents do not need a detailed revision history, DocBook XML defines elements for revision histories and DocMaker supports these conditionally. DocMaker will render a revision history as a table of revisions in the front matter of a book. The revision history will only be rendered in the PDF output and only when the book element has its status attribute set to draft. When the status attribute is set to another value, or is not present, the revision history will not be included in the output.

The revision history is included by adding a revhistory element to the book's bookinfo element; revision histories for elements other than the book element are not supported by DocMaker. The revhistory element can contain one or more revision elements, each describing the details of a change to the document. When authors make changes, they should add a new revision element to the revhistory.

A revision element describes a change using a number of child elements; the order of these elements is important. The first is the revnumber element that defines the revision number; next is the date element for the revision date; this is followed by any number of optional author or authorinitials elements; and, finally, a single optional revremark or revdescription element can be used to provide a short or long description of the change.

Here is an example:

```
<book role="user" status="draft">
  <title>How to Read a Manual</title>

  <bookinfo>

    <copyright>
      <year>2004</year>
      <holder>Cúram Software Limited</holder>
    </copyright>

    <revhistory>
      <revision>
        <revnumber>1.0</revnumber>
        <date>01-Nov-2004</date>
        <authorinitials>XY</authorinitials>
        <revremark>
          First draft of the book.
        </revremark>
      </revision>

      <revision>
        <revnumber>1.1</revnumber>
        <date>04-Nov-2004</date>
        <authorinitials>YZ</authorinitials>
        <authorinitials>WX</authorinitials>
        <author>
          <firstname>John</firstname>
```

```
        <surname>Smith</surname>
      </author>
      <revremark>
        Changes applied by review committee.
      </revremark>
    </revision>
  </revhistory>

  ...
  </bookinfo>

</book>
```

Note that the `status` attribute of the `book` element must be set to `draft` for the revision history to be rendered. If you want to render a copy of the document without the revision history, you can leave the revision history details in the content and just change the value of the `status` attribute to `final`, or delete the attribute completely. Regardless of the value of the `status` attribute, the revision history will never appear in the HTML Help output.

When writing revision history entries, the `revremark` element allows you to enter a simple comment; you can include plain text and use the `emphasis` element, if required. For more elaborate comments, you can use the `revdescription` element. This element allows you to include most of the block elements described in the next chapter; you can use lists, tables, figures, paragraphs, and other element, though it is unlikely that they will be required.

### 4.3.5  Books with Long Titles

Sometimes, the formal title of a book may be so long that it may cause some formatting problems during rendering. When generating PDF output, for example, DocMaker uses the book title in the page header, so a long title can cause the text in the header to wrap onto a new line and look unsightly. While a long formal title may be unavoidable, DocBook XML allows you to define an abbreviated title for a book for use when the title may cause formatting problems. This abbreviated title is defined using the `titleabbrev` element. The element is placed immediately after the book's `title` element. If DocMaker finds a `titleabbrev` element, it will use it in the page header during rendering, but it will still use the full title on the cover page. Here is an example:

```
<book role="user">
  <title>
    Cúram Deployment Guide for IBM WebSphere
    Application Server on Microsoft Windows 2000
  </title>
  <titleabbrev>
    Deployment Guide for WAS on Windows 2000
  </titleabbrev>
  ...
</book>
```

## 4.4  Parts

A `part` is an optional element within a book that can be used to combine `chapter` or `appendix` elements into related groups. A `part` can only appear at the top level of a `book`. If you decide to divide your book into parts, you can still place `chapter` and `appendix` elements outside parts (i.e., directly under the `book` element.) For example, it is common for the first `chapter` element in a book to introduce the structure of the book and describe the parts contained within but not be in a `part` element itself.

```
<book>
  ...
  <part>
    <title>Part One</title>

    <partintro>
      <para>This is the first part of this book.</para>
    </partintro>

    <chapter>
      ...
    </chapter>
    <chapter>
      ...
    </chapter>
  </part>

  <part>
    ...
  </part>

</book>
```

In the example above, the `book` element contains two `part` elements. Like all other hierarchy elements, the first element within the `part` is its `title`. This is followed by the optional `partintro` element that introduces the contents of the `part`. The `partintro` element can be quite elaborate and incorporate much the same content as a `chapter` element, but the contents will probably be determined by the style of book you want to write. Cúram documents are not divided into parts.

## 4.5  Chapters

Most books will be organized into a number of chapters. The `chapter` element can contain many different elements, but in Cúram documents, a `chapter` must contain only a `title` element followed by a number of `section` elements. All other content should be placed within the sections.

```
<book>
  ...
  <chapter>
    <title>My First Chapter</title>
    <section>
      ...
    </section>
    <section>
      ...
    </section>
  </chapter>
  ...
</book>
```

Using only a `title` and some `section` elements makes the structure of the `chapter` element easier to manage and is sufficient for almost all purposes.

## 4.6 Appendices

Supplementary and reference data is often included at the end of a book in one or more appendices. The `appendix` element is used to define an appendix. Other than the name, the `appendix` element is identical to the `chapter` element and should be structured in the same manner with a `title` element and `section` elements.

```
<book>
  ...
  <chapter>
     ...
  </chapter>
  ...
  <appendix>
    <title>My First Appendix</title>
    <section>
      ...
    </section>
    <section>
      ...
    </section>
  </appendix>
  ...
</book>
```

## 4.7 Sections

### 4.7.1 Section Structure

The `section` element is used to organize the contents of a `chapter`. Like other hierarchy elements, the `title` element should appear first and be followed by the content of the section. The most common content element is the `para` element. It contains the text of a paragraph. Here is an example:

```
<book>
  ...
  <chapter>
    ...
    <section>
      <title>My First Section</title>
      <para>
        I can finally start writing my document within
        this section.
      </para>
      ...
    </section>
  </chapter>
  ...
</book>
```

As the `section` elements contain the majority of the document content,

there are more options for the contents of a `section` than for any of the other hierarchy elements. The next chapter discusses the block elements that can appear with sections.

### 4.7.2 Nesting Sections

While DocBook allows section levels to be explicitly designated using elements such as `sect1`, `sect2`, etc., Cúram documents *must* use the `section` element for all sections. If one `section` is nested within another `section`, it will automatically be identified as a sub-section. This nesting can be continued to any level required, but nesting to a depth of more than three levels is usually a sign that sections are being used where lists or even simple paragraphs would be more appropriate.

The nesting of `section` elements allows more flexibility in the way sections are included in documents or reused in other contexts. This example shows a `chapter` containing a `section` that contains two nested sub-sections.

```
<chapter>
  ...
  <section>
    <title>My First Section</title>
    <section>
      <title>Introduction</title>
      ...
    </section>
    <section>
      <title>Another Sub-Section</title>
      ...
    </section>
  </section>
</chapter>
```

# Chapter 5

## Block Elements

## 5.1 Introduction

Block elements are those that are typically started on a new line in the output text and span one or more lines. Common block elements include paragraphs, lists, figures, examples, and tables. Some block elements can contain other block elements, such as when paragraphs are included in a table cell. This chapter describes the typical block elements that you can add to a `section` element, and how you should structure other block elements within them.

The examples presented here show a DocBook XML fragment followed by a rendering of that fragment within a `blockquote` to set it off from the normal text. The rendered example should be considered only an approximation of the actual rendering in your chosen environment.

## 5.2 Paragraphs

### 5.2.1 Overview

Paragraphs are the most common block element. DocBook defines three types of paragraph element: `para`, `formalpara`, and `simpara`. The use of `para` and `formalpara` paragraphs is straightforward and is describe below. The `simpara` element is just a paragraph that cannot contain any other block elements.

> **Important**
>
> Cúram documents should only use `para` and `formalpara` paragraph elements. Block elements should *not* be placed inside paragraphs.

### 5.2.2 `para`

The standard paragraph element. It can contain text and other elements. Use this element when you want a paragraph.

```
<para>
  This is text within a paragraph.
</para>
```

This is text within a paragraph.

### 5.2.3 `formalpara`

A formal paragraph that contains a `title` element and a `para` element. The title text is usually rendered in a stronger font at the beginning of the paragraph text.

```
<formalpara>
  <title>A Formal Paragraph</title>
  <para>
    This is text within a formal paragraph.
  </para>
</formalpara>
```

**A Formal Paragraph.** This is text within a formal paragraph.

## 5.3 Lists

### 5.3.1 Overview

Lists allow information to be arranged in sequence and help to organize information effectively. There are a number of different list types in Doc-Book: `itemizedlist`, `orderedlist`, `variablelist`, `segmentedlist`, `simplelist`, and `calloutlist`. Only the first three list types should be used in Cúram documents, as the others are better represented using tables.

> **Important**
>
> Cúram documents should *not* use the `segmentedlist`, `simplelist`, and `calloutlist` types. A `table` or `informaltable` element should be use instead if required.

A list can include a optional `title` element as its first child element. This will usually appear as a heading above the list when the list is rendered.

Each item within a list is contained in a `listitem` element. The `listitem` can contain other block elements such as paragraphs or other lists. List items cannot have titles, but the effect can be achieved by making the first element in the list item a `formalpara` instead of a `para`. Alternatively, the `variablelist` can be used for a similar effect if the list does

not need to be numbered or bulleted.

## 5.3.2 `itemizedlist`

This list is typically rendered as a bulleted list. It is used for list of items where the order of the items is not important.

```
<itemizedlist>
  <listitem>
    <para>This is the first list item.</para>
  </listitem>
  <listitem>
    <para>This is the second list item.</para>
    <para>The second list item has two paragraphs.</para>
  </listitem>
  <listitem>
    <para>This is the third list item.</para>
  </listitem>
</itemizedlist>
```

- This is the first list item.

- This is the second list item.

  The second list item has two paragraphs.

- This is the third list item.

A list item can also contain another list.

```
<itemizedlist>
  <listitem>
    <para>This is the first list item.</para>
  </listitem>
  <listitem>
    <para>The second list item contains a nested list.</para>
    <itemizedlist>
      <listitem>
        <para>The first list item in the nested list.</para>
      </listitem>
      <listitem>
        <para>The second list item in the nested list.</para>
      </listitem>
    </itemizedlist>
  </listitem>
  <listitem>
    <para>This is the third list item.</para>
  </listitem>
</itemizedlist>
```

- This is the first list item.

- The second list item contains a nested list.

  - The first list item in the nested list.

  - The second list item in the nested list.

- This is the third list item.

### 5.3.3 `orderedlist`

This list is typically rendered as a numbered list. It is used for list of items where the order of the items is important, for example, where the steps in a process are being described. It is used just like an `itemizedlist` element.

In this example, the optional list `title` is used and the last list item includes a `formalpara` element. This can also be used in the other list types.

```
<orderedlist>
  <title>An Ordered List</title>
  <listitem>
    <para>This is the first list item.</para>
  </listitem>
  <listitem>
    <para>This is the second list item.</para>
  </listitem>
  <listitem>
    <formalpara>
      <title>An Important Item</title>
      <para>This is the third list item.</para>
    </formalpara>
  </listitem>
</orderedlist>
```

An Ordered List

1.   This is the first list item.

2.   This is the second list item.

3.   **An Important Item.** This is the third list item.

A list item can also contain another list. When nesting lists, different types of list can be mixed freely.

### 5.3.4 `variablelist`

This list is typically rendered as a list of terms and descriptions. More than one term can be associated with a description. It is structured slightly differently to the itemized and ordered lists. Each item in the list is contained within a `varlistentry`. The `varlistentry` then contains one or more `term` elements and then the usual `listitem` element. When rendered, the terms are usually listed on a single line and comma-separated.

```
<variablelist>
  <varlistentry>
    <term>Much Ado About Nothing</term>
    <listitem>
      <para>Comedy</para>
    </listitem>
  <varlistentry>
  <varlistentry>
    <term>King Lear</term>
    <term>Macbeth</term>
```

```
    <listitem>
      <para>Tragedies</para>
    </listitem>
  <varlistentry>
  <varlistentry>
    <term>Titanic: The Movie</term>
    <listitem>
      <para>Travesty</para>
    </listitem>
  <varlistentry>
<variablelist/>
```

**Much Ado About Nothing**

> Comedy

**King Lear, Macbeth**

> Tragedies

**Titanic: The Movie**

> Travesty

Just like the other list types, the list can have a `title` and the `listitem` elements can contain block elements including other lists.

## 5.4   Figures and Examples

### 5.4.1   Overview

Figures and examples are formal element that contain a `title` and some block elements. Typically, an example will present some program code and a figure will present a diagram or image.

You should always use a `figure` or `example` element if you intend to cross-reference the element from another part of the document. Even if not cross-referenced, a `figure` or `example` element should be used to maintain consistency if other figures or examples of equal importance in the document are the targets of cross-references. While you can add images and program code to your documents without formal titles, it is not possible to create a proper cross-reference to untitled content.

For examples and figures that are not cross-referenced and that are used informally within the document, the `informalfigure` and `informalexample` elements can be used. They are just like their more formal namesakes, but do not contain a `title` element.

> **Important**
>
> Cúram documents should use `example` and `figure` elements for all program code samples and images respectively. If the examples or figures are used informally, will not be cross-referenced, and do not need a `title` element, the `informalexample` or `informalfigure` elements can be used.

### 5.4.2 **figure**

The `figure` element must contain a `title` element and a block element such as a `mediaobject` or `graphic`. In Cúram documents, only a `mediaobject` should be used.

```
<section>
  ...
  <figure>
    <title>A Sample Image</title>
    <mediaobject>
      <imageobject>
        <imagedata fileref="images/logo.gif" format="GIF"/>
      </imageobject>
      ...
    </mediaobject>
  </figure>
  ...
</section>
```



Figure 5.1 A Sample Image

### 5.4.3 **mediaobject**

When including a `figure` element, a `mediaobject` element is used to refer to the image or diagram to be included in the document. The `mediaobject` element supports not only images but also text, video, and audio objects. In Cúram documentation, only the image and text objects should be used. While the basic use of the element is simple, there are a number of aspects to the `mediaobject` element that warrant elaboration.

> **Important**
>
> A `mediaobject` element should *never* appear outside a `figure` or `informalfigure` element in a Cúram document.

To include an image in the document, an `imageobject` should be added to the `mediaobject`. The `imageobject` element should contain an `imagedata` element. The `imagedata` element makes the reference to the file containing the image to be included.

```
<mediaobject>
  <imageobject>
    <imagedata fileref="images/logo.gif" format="GIF"/>
  </imageobject>
  ...
</mediaobject>
```

The `imagedata` element has several attributes of which the first two are the most important. While DocBook XML defines more than just those attributes described below, these are the only attributes supported by Doc-Maker.

**fileref**

> The path to the file to be included specified relative to the location of the document file.

**format**

> The format of the image file. While DocBook supports a large number of formats, for Cúram documentation, a limited number are supported. For bitmap images such as screen-shots, icons, user-interface examples, etc., the `GIF` format should be used; for photographic images, the `JPEG` format should be used; and for scalable, vector-based images such as diagrams, flow-charts, etc., the `SVG` format should be used.
>
> It is quite acceptable to maintain the image (particularly vector-based diagrams) in the preferred format of the application in which they were created, but they should be exported to one of the above formats before being referenced from a DocBook document.

**width**

> Specifies the horizontal dimension of the view-port into which the image will be inserted. This value will only affect images in PDF documents; it will have no effect on HTML output. The value should be expressed in millimeters using the `mm` suffix, e.g., `120mm`.

**depth**

> Specifies the vertical dimension of the view-port into which the image will be inserted. The value is specified in the same manner as for the `width` attribute.

In general, Cúram documentation will only use the `fileref` and `format` attributes. The `width` and `depth` attributes should only be used if problems occur when trying to render large images to PDF.

> **Large Images**
>
> The Apache FOP library used to render the PDF documents does not automatically scale images to fit on a page if they are too large. Large images will cause the rendering to fail with a message stating that an infinite loop was detected when trying to lay out a page. To overcome this, you will need to explicitly scale the image to fit on the page. For tall images, set the value of the `depth` attribute to `150mm` to ensure that it is not taller than the content area of the page. Similarly, for wide images, use the `width` attribute, starting with a value of about `120mm`, to reduce the size of the image. You only need to set one of these attributes; the aspect ratio will be preserved automatically.

### 5.4.4 Accessibility

In many document formats—HTML, for example—an image typically has an alternative text description that is used in place of the image if the image cannot be displayed or used by assistive technology to identify the purpose of the image to a visually impaired reader. This is often referred to a the "alt text" and it can be specified via a `textobject` element within the `mediaobject`. The `textobject` element should contain a `phrase` element containing the short textual description of the image.

Sometimes the "alt text" is a little too short to be useful: graphs or diagrams may require longer, more detailed descriptions to aid visually impaired readers. To support this, you can add another `textobject` to the `mediaobject` and add one or more `para` elements to it to describe the image in detail. To distinguish the "alt text" from the long description, add `role` attributes to the `textobject` elements and set them to `alt` and `desc` respectively. Always place the "alt text" before the long description.

> **Important**
>
> All Cúram user guides *must* include "alt text" and long descriptions for all images to ensure that rendered output is accessible to all readers. They are optional for other guides where accessibility is not a requirement.

```
<mediaobject>
  <imageobject>
    <imagedata fileref="images/logo.gif" format="GIF"/>
  </imageobject>
  <textobject role="alt">
    <phrase>Cúram Software logo</phrase>
  </textobject>
  <textobject role="desc">
    <para>
      The blue and white Cúram Software logo displaying the word
      <quote>Cúram</quote> in a Celtic-style font over the word
      <quote>software</quote> in capital letters and a sans-serif
      font.
    </para>
  </textobject>
</mediaobject>
```



### 5.4.5 example

The `example` element must contain a `title` element and a block element such as a `programlisting`, a paragraph, or a list. In Cúram documents, most simple examples should contain only a `programlisting`. However, there may be situations where additional elements need to be added (the examples in this chapter are `informalexample` elements containing a `programlisting` element and a `blockquote` element containing the rendering of the example shown in the `programlisting`).

```
<section>
```

```
  ...
  <example>
    <title>An Example of Some Java Code</title>
    <programlisting>void main(String[] arguments) {
  // Some snazzy Java code.
}</programlisting>
  </example>
  ...
</section>
```

```
        void main(String[] arguments) {
          // Some snazzy Java code.
        }
```

Example 5.1 An Example of Some Java Code

### 5.4.6  `programlisting`

A `programlisting` element should be used within an `example` or `informalexample` element. When rendered, the whitespace within the `programlisting` element will be preserved so that line breaks, formatting, and indentation are not impacted. This is the main reason to use a `programlisting` element rather than just place the program code in normal paragraphs.

> **Important**
>
> A `programlisting` element should *never* appear outside an `example` or `informalexample` element in a Cúram document.

> **Tip**
>
> A `programlisting` element allows the code to be marked up using many of the DocBook inline elements. For example, you can use the `emphasis` element to highlight areas of the code. More details on inline elements follow in the next chapter.

In this example, the program code is not indented to the expected level within the `programlisting` element, but is indented relative to the left margin. If typical XML indentation is applied, both that indentation and the program code indentation will be preserved and this is rarely what is intended. Notice that there are no line breaks after the opening tag or before the closing tag of the `programlisting` element: these would be preserved in the output if they were used.

```
<informalexample>
  <programlisting>void main(String[] arguments) {
  // Some snazzy Java code.
}</programlisting>
</informalexample>
```

```
        void main(String[] arguments) {
          // Some snazzy Java code.
        }
```

The code that you enter within the `programlisting` must not cause your document to become invalid or not well-formed. This might happen if you entered sample XML code that conflicted with the XML code of the Doc-Book document or if the sample code contained one of the five reserved XML characters. There are two ways to overcome this.

The first way is to escape all reserved XML characters in the code using the appropriate character entity. Samples of XML code will obviously contain many reserved characters, so escaping them and maintaining the document may become difficult. When non-XML code is presented, the task is usually easier. In Java code, for example, only operators that use the greater-than, less-than, and ampersand characters will need to be escaped and there may not be many of them.

```
<informalexample>
  <programlisting>void main(String[] arguments) {
  if (arguments.length &lt; 3) {
    // ...
  }
}</programlisting>
</informalexample>
```

```
        void main(String[] arguments)
        {
          if (arguments.length < 3) {
            // ...
          }
        }
```

The second method is more appropriate for XML code sample, but can equally be applied to any other program code. It is probably the best all-round solution for all program examples as it requires the least effort to implement. XML supports the definition of regions of character data within a document that should be treated as text by the XML parser. The region is called a *CDATA block*. Reserved XML characters within the CDATA block will not cause any errors. A CDATA block begins with the character sequence " `<![CDATA[` " and ends with the sequence " `]]>` ". Because the line break characters both outside and inside the CDATA block are preserved, the formatting of a `programlisting` with a CDATA block is typically modified to avoid extra line breaks in the output.

```
<informalexample>
  <programlisting><![CDATA[<para>
  A sample DocBook paragraph.
</para>]]></programlisting>
</informalexample>
```

```
        <para>
          A sample DocBook paragraph.
        </para>
```

The only time this method breaks down is when example code presents an example of using a CDATA block. In that situation, a combination of

CDATA blocks and character entities can be used.

## 5.5 Tables

### 5.5.1 Overview

A `table` is a formal element that contains a `title` and a `tgroup` element that defines a two-dimensional grid of cells that contain some document content. The top row of the table contains the headings of each of the columns in the table. The other rows contain the information being tabulated. Like the `example` and `figure` elements, the `table` element has a corresponding `informaltable` element that can be used when a title is not required.

> **Important**
>
> All tables used in Cúram documents should use the `table` element with a `title`, not the `informaltable` element, unless the table is used only to assist the formatting of simple lists of data.

### 5.5.2 `table`

The structure of the `table` element is probably the most complex of any of the DocBook elements. However, in Cúram documentation, tables are generally fairly simple grids and the basic use of the `table` element is not difficult to master.

A `table` element contains a `title` element and a `tgroup` element. The `tgroup` element defines the layout of the table. There can be more than one `tgroup` element, but that configuration is not supported in Cúram documentation. The `tgroup` element must have a `cols` attribute specifying the number of columns in the table.

Within the `tgroup` element you place one `thead` and one `tbody` element. The `thead` element contains a single `row` element that defines the column headings for the table. If you do not want headings, you can leave out the `thead` element. The `tbody` element contains one or more `row` elements that make up the main body of the table.

Each `row` can contain `entry` elements that define the cells within that row. The `entry` elements can contain plain text or block elements such as paragraphs, images, lists, etc. In Cúram documentation, you should generally only place plain text or `para` elements in the `entry` elements.

```
<section>
  ...
  <table>
    <title>The Phone List</title>
    <tgroup cols="2">
      <thead>
        <row>
          <entry>Name</entry>
          <entry>Extension</entry>
```

```
        </row>
      </thead>
      <tbody>
        <row>
          <entry>Bob</entry>
          <entry>219</entry>
        </row>
        <row>
          <entry>Alice</entry>
          <entry>260</entry>
        </row>
      </tbody>
    </tgroup>
  </table>
  ...
</section>
```

| Name | Extension |
|------|-----------|
| Bob | 219 |
| Alice | 260 |

Table 5.1 The Phone List

### 5.5.3 `colspec`

The `colspec` element is optional but can be added to the `tgroup` element to control the widths of the columns in the table. There should be one `colspec` element for each column in the table. The `colspec` elements are placed within the `tgroup` element before the `thead` element. They should contain two attributes: `colnum` specifying the number of the column to which this `colspec` applies (starting with `1` for the first column), and `colwidth` defining the width of the indicated column.

The width of the column can be specified in a number of different units (inches, millimeters, points, percent, etc.) However, for consistency, Cúram documentation should only use *proportional widths*. Proportional widths simply specify column widths as ratios of the total table width. The value is a number followed by an asterisk ("*") character. For example, if the table contains two columns, and you want the first column to be twice the width of the second column (i.e., a ratio of 2:1), the proportional widths will be `2*` and `1*` respectively.

The fraction of the total table width allocated to a column can be calculated by dividing the column's width by the sum of all the proportional column width values. A ratio of 5:2:2:1 represents five tenths (a half), two-tenths (a fifth), two-tenths, and one tenth and the proportional width values of `5*`, `2*`, `2*`, and `1*` can be used.

There is no need to set one of the numbers to `1*` and calculate from there. As percentages are just fractions, they can be used instead. For example, the 5:2:2:1 ratio represents 50%, 20%, 20%, and 10%, so the proportional widths of `50*`, `20*`, `20*`, and `10*` can be used. The 2:1 ratio is (approximately) 66.67% to 33.33%, so `2*` and `1*` are easier to enter, but `66*` and `33*` could be used to represent the percentages. Though they only

add to 99 and not 100, this 66:33 ratio is actually more accurate than the percentages specified to only two decimal places.

❗ Narrow Column Widths

It is possible to select column widths that are too narrow for the content of the column and this can cause problems when generating PDF output. The typical symptom is an endlessly repeating message from the *FOP* library stating, "area contents overflows area in line". To remedy this, locate the table that is causing the problem and make the narrowest column slightly wider. It can sometimes be difficult to identify the culprit, but it is usually a table you have just changed prior to the error occurring. However, you may have to temporarily comment out parts of your document to eliminate possible candidates from your search.

```
<section>
  ...
  <table>
    <title>The Phone List Again</title>
    <tgroup cols="2">
      <colspec colnum="1" colwidth="3*"/>
      <colspec colnum="2" colwidth="2*"/>
      <thead>
        <row>
          <entry>Name</entry>
          <entry>Extension</entry>
        </row>
      </thead>
      <tbody>
        <row>
          <entry>Bob</entry>
          <entry>219</entry>
        </row>
        <row>
          <entry>Alice</entry>
          <entry>260</entry>
        </row>
      </tbody>
    </tgroup>
  </table>
  ...
</section>
```

| Name | Extension |
|------|-----------|
| Bob | 219 |
| Alice | 260 |

Table 5.2 The Phone List Again

### 5.5.4 Accessibility

Accessibility guidelines require that each table in a Cúram user guide has associated summary text describing the purpose and contents of the table. A table summary is used when rendering to HTML format, so that a screen-reading application can read the table summary to the user before reading the table's entries.

> **!** Important
>
> All Cúram user guides *must* have table summaries. They are optional for other documents.

To insert a table summary, add a `blockinfo` element to the `table` before the table's `title` element and add an `abstract` element to the `blockinfo` element. The `abstract` can contain an optional `title` element and one or more `para` elements containing the description of the table.

```
<section>
  ...
  <table>
    <blockinfo>
      <abstract>
        <para>
          A table of employees' extension numbers with names in
          the first column and extension numbers in the second
          column. The rows are in no particular order.
        </para>
      </abstract>
    </blockinfo>
    <title>Another Phone List</title>
    <tgroup cols="2">
      <colspec colnum="1" colwidth="3*"/>
      <colspec colnum="2" colwidth="2*"/>
      <thead>
        <row>
          <entry>Name</entry>
          <entry>Extension</entry>
        </row>
      </thead>
      <tbody>
        <row>
          <entry>Bob</entry>
          <entry>219</entry>
        </row>
        <row>
          <entry>Alice</entry>
          <entry>260</entry>
        </row>
      </tbody>
    </tgroup>
  </table>
  ...
</section>
```

| Name | Extension |
|------|-----------|
| Bob | 219 |
| Alice | 260 |

Table 5.3 Another Phone List

### 5.5.5 The Peculiar Content of `entry` Elements

The `entry` element is used to represent a table cell. It also has the distinction of being the only element defined in the DocBook XML standard that can contain block elements or inline elements (including text) but not a mix of both. After an opening `<entry>` tag, if the next node is not a block element, then inline elements and text are expected. However, character con-

tent, even whitespace, is considered a text node, so any character content between the opening `<entry>` tag and the opening block element tag renders the block element invalid. Similarly, you cannot place character content between subsequent block elements within an `entry`. You can read more about this on the reference page for the `entry` element in DocBook: The Definitive Guide [http://www.docbook.org/tdg/en/html/entry.html].

> **Using para within entry**
>
> Most of the examples in this chapter do not use a `para` element within an `entry` element. However, as you will be writing documents with tables more complex than those shown here, you should use the `para` element to contain the content of `entry` elements in the table body. For simple column headings defined in the table head, the `para` element is not as important.

Here are some examples of what is valid:

```
<entry>Bob</entry>

<entry><para>Bob</para></entry>

<entry>
  <personname>Bob</personname>
</entry>

<entry><para>
    Bob sits near Carol.
  </para><para>
    Alice sits near Dave.
  </para></entry>
```

Example 5.2 Valid Content of **entry**

The first `entry` contains only text, so it is valid. The second `entry` contains only a block element and is also valid. The third `entry` contains whitespace around a `personname` element, but this is an inline element (see Chapter 6, *Inline Elements*), not a block element, and it is permitted to mix character content with inline elements. The last `entry` contains two `para` elements and there is no character content within the `entry` element that is not within these block elements, so this is valid.

When a block element like `para` is used, there can be no character content within the `entry` element, only other block elements like `para`. Whitespace must be avoided, not only between the `entry` element's tags and the `para` elements' tags, but also between the `para` elements' tags, as whitespace there would be considered character content of the `entry` element and could not be mixed with block elements. Some of these invalid uses are shown below:

```
<entry>
  Bob
  <para>sits near Carol.</para>
</entry>

<entry>
  <para>
    Bob sits near Carol.
  </para>
```

```
  <para>
    Alice sits near Dave.
  </para>
</entry>

<entry><para>
    Bob sits near Carol.
  </para>
  <para>
    Alice sits near Dave.
  </para></entry>
```

Example 5.3 Invalid Content of **entry**

In the first `entry` element there is a mixture of character content ("Bob" and whitespace) and block elements (`para`). In the second, the same mix exists, only this time the character content is just whitespace. In the third, there is still whitespace between the `para` elements that is considered character content of the `entry` element.

> 💡 Use XXE
>
> If you use the XMLmind XML Editor (*XXE)* as your DocBook XML editor, all of these peculiar requirements of the `entry` element are managed automatically. *XXE* reformats saved files to conform to these rules, so you can use it to "fix" a file created using a text editor.

## 5.5.6 Spanning Columns

Occasionally, you will want a single heading above one or more columns in the body of a table, or you may want to split one or more table entries in two. These conditions require the use of *spans*: where an `entry` element spans more than one column. Spans are specified by naming the columns and then using those names to identify the start and end columns for an `entry`. The column names are defined by adding a `colname` attribute to the `colspec`. The spans are defined by adding `namest` and `nameend` attributes to an `entry` element; the attribute values are set to the names of the starting column and the ending column in the span respectively.

If you have a single heading above two columns, then the heading `entry` spans those two columns.

```
<section>
  ...
  <table>
    <title>External Phone List</title>
    <tgroup cols="3">
      <colspec colnum="1" colwidth="6*"/>
      <colspec colnum="2" colwidth="2*" colname="area-code"/>
      <colspec colnum="3" colwidth="5*" colname="number"/>
      <thead>
        <row>
          <entry>Name</entry>
          <entry namest="area-code" nameend="number">
            Phone Number
          </entry>
        </row>
      </thead>
```

```
        <tbody>
          <row>
            <entry>Bob</entry>
            <entry>01</entry>
            <entry>555-9876</entry>
          </row>
          <row>
            <entry>Alice</entry>
            <entry>021</entry>
            <entry>555-6789</entry>
          </row>
        </tbody>
      </tgroup>
  </table>
  ...
</section>
```

| Name | Phone Number | |
|---|---|---|
| Bob | 01 | 555-9876 |
| Alice | 02<br>1 | 555-6789 |

Table 5.4 External Phone List

The table was defined to have three columns. The entries in the table's body are the same as before, but there are only two entry elements in the table's heading row, and the second is defined to start in the area-code column and finish in the number column. This makes the heading span those columns.

Sometimes the spanning mechanism can be confusing, but it helps to remember that it always works like the above. For example, if you have a column where a single entry is split in two, the situation is no different: you actually have two columns and all entry elements except the "split" entry are span those columns. The "split" entry is actually two ordinary entry elements; the only ones that do not specify a span. There is no concept of a "split" entry in DocBook: an entry is either in a single column or it spans multiple columns.

```
<section>
  ...
  <table>
    <title>Yet Another Phone List</title>
    <tgroup cols="3">
      <colspec colnum="1" colwidth="6*"/>
      <colspec colnum="2" colwidth="2*" colname="direct-dial"/>
      <colspec colnum="3" colwidth="5*" colname="number"/>
      <thead>
        <row>
          <entry>Name</entry>
          <entry namest="direct-dial" nameend="number">
            Extension
          </entry>
        </row>
      </thead>
      <tbody>
        <row>
          <entry>Alice</entry>
          <entry namest="direct-dial" nameend="number">260</entry>
        </row>
        <row>
```

```
        <entry>Bob</entry>
        <entry>*</entry>
        <entry>219</entry>
      </row>
      <row>
        <entry>Carol</entry>
        <entry namest="direct-dial" nameend="number">288</entry>
      </row>
      <row>
        <entry>Dave</entry>
        <entry namest="direct-dial" nameend="number">235</entry>
      </row>
    </tbody>
  </tgroup>
</table>
...
</section>
```

| Name | Extension | |
|------|-----------|---|
| Alice | 260 | |
| Bob | * | 219 |
| Carol | 288 | |
| Dave | 235 | |

Table 5.5 Yet Another Phone List

In this table, Bob's number appears to be the odd one out: it is the only one that has an asterisk to indicate a direct dialing option for his number and appears to be split in two. However, the DocBook XML code shows that there are two columns defined for the extension number "column" and that these entries are the only ones that do not span those two columns: all the other ones are odd.

## 5.6 Admonitions

### 5.6.1 Overview

Admonitions are notes set off from the main text that highlight pertinent information. DocBook defines several types of admonition, each using its own element: `note`, `tip`, `important`, `warning`, and `caution`. When rendered, an admonition often has a title or icon identifying its type. All admonitions are used the same way, so a single example should suffice.

```
<section>
  ...
  <tip>
    <para>
      Here is a useful tip for blah, blah, blah.
    </para>
  </tip>
  ...
</section>
```

 Tip

> Here is a useful tip: read the fine manual before go-
> ing any further.

## 5.6.2 Titles

Admonitions can include a `title` element if desired. If a `title` element is not present, a default title may be added during rendering. For example, if a `warning` element has no title, DocMaker will generate a PDF file showing the title as "Warning"; if a title is present, it will be used.

```
<warning>
  <title>Warn Your Readers</title>
  <para>
    Be sure to use a <sgmltag>warning</sgmltag> element when
    instructing readers to perform tasks that could result in
    the loss of data.
  </para>
</warning>
```

> **❗ Warn Your Readers**
>
> Be sure to use a `warning` element when instructing readers to perform tasks that could result in the loss of data..

## 5.6.3 Choosing an Admonition

If you decide you want to use an admonition, use the following guide to select the appropriate admonition element. The `caution` element is not supported for use in Cúram documentation.

**note**

Information that is supplementary to the narrative but is useful to the reader. A `note` element can be considered a more emphatic alternative to a `footnote` element. If the note is lengthy (i.e., more than a few sentences long), consider using a `sidebar` element instead.

**tip**

Useful advice on the use of Cúram, or on other aspects of the subject that could save the reader time and effort later. The reader does not *need* to know this information, but it is helpful nonetheless.

**important**

An important piece of information of which the reader should be aware. This could be information about prerequisite tasks that must be performed before proceeding, assumptions that have been made, or may indicate that it is possible that the task or procedure will not work and time may be wasted correcting the problem if the information is not acted upon.

**warning**

A warning that care should be taken or problems, including data-loss, could occur.

## 5.7 Sidebars

### 5.7.1 Overview

A `sidebar` element contains text set off from the main content of the document. It can be used to supply supplementary information that would otherwise interrupt the flow of the narrative, for example, it can be used for digressions that are interesting, though not directly relevant to the text. If the information is brief, consider using an admonition, but if there are more than a few sentences, use the `sidebar` element instead. If the information is likely to occupy more than half a page, consider providing the information in a `section` of its own.

### 5.7.2 **sidebar**

The `sidebar` element structure is like that of a `section`. It can include any of the block elements described in this chapter except for another `sidebar` element.

> **Important**
>
> Cúram documents should always include a `title` element in a `sidebar` element.

```
<section>
  ...
  <sidebar>
    <title>My Sidebar</title>
    <para>
      This is an example of a sidebar element.
    </para>
  </sidebar>
  ...
</section>
```

> My Sidebar
>
> This is an example of a sidebar element.

## 5.8 Block Quotes

### 5.8.1 Overview

A `blockquote` element contains text (often a quotation) set off from the main content of the document. It is usually rendered in place within the

main flow but indented, or styled to identify it as separate from the rest of the text. Its use within Cúram documents will probably be limited, but it may be useful when quoting passages from other sources or for other formatting requirements.

### 5.8.2 **blockquote**

The `blockquote` element structure is like that of a `sidebar`. It can include any of the block elements described in this chapter including another `blockquote` element. The `title` element is optional and the quote can also include an `attribution` element containing the details of the source of the quotation.

> **ⓘ Note**
>
> The `attribution` (if supplied) should appear *before* the body of the quotation. This may be rendered out of place in a WYSIWYG editor where you would expect the attribution to be rendered at the end of the quotation and not at the start, but the final rendering of the document will be handled correctly.

```
<section>
  ...
  <blockquote>
    <attribution>Bill Gates</attribution>
    <para>
      There are people who don't like capitalism, and people
      who don't like PCs. But there's no one who likes the
      PC who doesn't like Microsoft.
    </para>
  </blockquote>
  ...
</section>
```

> There are people who don't like capitalism, and people who don't like PCs. But there's no one who likes the PC who doesn't like Microsoft.
>
> —Bill Gates

## 5.9 Literal Layout

### 5.9.1 Overview

While a `programlisting` element respects the whitespace used to format a block of text, it is usually rendered in a mono-spaced font suitable for program source code. If you require the whitespace in the text to be respected but do not want the font to change, the `literallayout` element can be used. It may, for example, be used within a `blockquote` element where the quoted text contains significant line breaks (such as in poetry). It is unlikely to be used often in Cúram documentation, but may prove useful in exceptional circumstances.

❗ Important

Although whitespace is preserved in a literallayout element, the use of whitespace for formatting other than line breaks (e.g., for indentation) is likely to be rendered unexpectedly as the font used in the editor may be different from the proportional font used for final rendering.

### 5.9.2 literallayout

The literallayout element structure is like that of a programlist-ing. It should not contain other block elements.

```
<section>
  ...
  <literallayout>
Mary had a little lamb.
Its fleece was white as my new 90gsm copier paper.
  </literallayout>
  ...
</section>
```

Mary had a little lamb.
Its fleece was white as my new 90gsm copier paper.

# Chapter 6

## Inline Elements

## 6.1 Introduction

Inline elements are elements used within block elements to identify words or phrases that should be treated differently from the main body of the text. Common examples include emphasis, superscripts and subscripts, foreign phrases, hyperlinks, cross-references, programming language elements, user-interface components, and citations. Unlike block elements, they do not start a new line and rarely require extra vertical space in the document.

The effective use of inline elements provides many opportunities for automatic document processing. As such, it is important that they are used extensively and accurately in Cúram documents. This chapter describes the inline elements that you should use and when you should use them.

## 6.2 Using Inline Elements

Inline elements are all used the same way. They are inserted into the text and typically surround a word or phrase. When using an ordinary text editor, they should not be indented like block elements.

```
<para>
  You <emphasis>must</emphasis> do this first.
</para>
```

> You *must* do this first.

Inline elements are usually nested within block elements, but can also appear inside other inline elements.

```
<para>
  She said, <quote>You <emphasis>must</emphasis> do this
  first.</quote>
</para>
```

She said, "You *must* do this first.".

Some inline elements accept attributes that modify or refine their meaning. The sgmltag element, for example, can have a class attribute with a value of starttag to indicate that a tag name is used in the context of it being a starting tag. When the document is rendered, the tag name will be formatted as a starting tag with opening and closing angle brackets added automatically.

```
<para>
  Start a <sgmltag>para</sgmltag> element with the
  <sgmltag class="starttag">para</sgmltag> tag.
</para>
```
Start a para element with the <para> tag.

The next section presents the basic inline elements that should be used in Cúram documents wherever relevant. Later, the more complex inline elements will be presented in their own sections.

## 6.3 Basic Inline Elements

### 6.3.1 Overview

There are a number of inline element that you will find yourself using very regularly. The inline elements described here *must* be used in Cúram documents when appropriate.

### 6.3.2 **quote**

This is used for an inline quotation. This is appropriate when quoting small passages from other texts, or when introducing new words or phrases. If a quotation is more extensive, a blockquote element should be used.

### Do Not Use Quotation Mark Characters

Cúram documents should *never* include quotation mark characters around normal text. The quote element should *always* be used instead. In the rendered output, the appropriate style (single or double quotation marks) will be used and, depending on the target and encoding, the correct characters for opening and closing quotes will be applied. Many editors do not support such "smart quotes", others use the wrong characters, different authors seem to have different preferences too, so use the quote element to avoid all of these problems.

If you are writing sample source code, you can use literal quotation marks appropriate for the programming language in question. Similarly, if you are instructing a user to enter certain characters, say on a

command line, you can use literal quotation marks. These cases are exceptions only because you specifically require a particular quotation mark character for syntactical reasons.

### 6.3.3 **emphasis**

Use this element to stress the importance of a word or a phrase. *Never* use emphasis for any other reason. It is *not* a generic way to apply an italic font. If extra emphasis is required, an admonition block element, such as `warning` or `important`, is probably more appropriate.

### 6.3.4 **firstterm**

Use this element to highlight a new word or phrase when it is first introduced in the text. Using this element is preferred over using the `quote` or `emphasis` element for this purpose.

### 6.3.5 **envar**

Use this for the name of an environment variable.

### 6.3.6 **filename**

Use this for the name of a file. If the name presented represents a pattern for a file name, combine this element with the `replaceable` element.

### 6.3.7 **replaceable**

Use this to indicate that the text is a placeholder that should be replaced with an appropriate value. For example, descriptions of commands might use this around the text "file-name" to indicate that the text "file-name" should be replaced with the an appropriate real file name when typing the command. It can also be used when using naming patterns, for example:

```
<para>
  Find the file called
  <filename><replaceable>server-name</replaceable>.log</filename>
</para>
```

Find the file called *server-name*.log.

### 6.3.8 **citetitle**

Use this for the title of a book or other document referenced from the text. If the reference is an actual hyperlink to the document, nest the `citetitle` element within the appropriate type of link element.

### 6.3.9 **ulink**

Use this for a hypertext link to an external source specified by a URL. The `url` attribute should be set to the target URL of the link. Typically, the text within the body of the element will be rendered as an active hypertext link. If the text is the title of a document, you must mark it by using a `cite-title` element within the `ulink` element. Similarly, for text that falls into any other category normally marked up, the appropriate markup should be nested within the `ulink` element. If you do not supply any text in the body of the element, the URL is used instead. Here are some examples:

```
<para>
  Use <ulink url="http://www.google.com/">Google</ulink> to
  find answers to your questions.
</para>
<para>
  Cúram Software provides more information about the Cúram
  Reference Application on their website at
  <ulink url="http://www.curamsoftware.com/cu.ra.php" />.
</para>
<para>
  See
  <ulink url="http://www.docbook.org/tdg/"><citetitle>DocBook:
  The Definitive Guide</citetitle></ulink> for more on the
  DocBook XML format.
</para>
```

Use Google [http://www.google.com/] to find answers to your questions.

Cúram Software provides more information about the Cúram Reference Model on their website at http://www.curamsoftware.com/cu.ra.php.

See *DocBook: The Definitive Guide* [http://www.docbook.org/tdg/] for more on the DocBook XML format.

In this example it is worth noting the care that has to be taken with whitespace characters when using inline elements. In the second paragraph, there is no whitespace between the end of the `ulink` element and the full stop. Similarly, in the last paragraph, there is no whitespace between the `ulink` element's tags and the `citetitle` element's tags, or between the `citetitle` element's tags and the title text. Whitespace between the words in the title is permitted: it will be reduced to a single space character between each word; but, if the hyperlink is underlined, whitespace between the tags may result in underlined spaces before and after the hyperlink text.

If you want to include a URL in a document that is not a hyperlink—perhaps it is just an example of a URL—see Section 6.3.11, *systemitem* .

### 6.3.10 `email`

Use this to mark an e-mail address in the text. This allows an e-mail address to be rendered as a hyperlink when the output format allows.

### 6.3.11 **systemitem**

The systemitem element is a catchall element for content describing things related to computer systems. To refine its meaning, you set its class attribute to one of the values shown in the table below. Its main use in Cúram documents is for URLs or URIs that are not actual hyperlinks (perhaps they are just examples), for network addresses, domain names, and the like. A future version of the DocBook XML standard will include a new uri element, but use systemitem with the class set to resource for now.

| Attribute Value | Usage |
| --- | --- |
| resource | The name of a resource. Use this for URLs or URIs that are not hyperlinks, just examples. |
| systemname | The name (or "host name") of a computer system that is unique to that system within a network domain. |
| domainname | The name of a local or top-level domain in a network. This is the part of name of a system that is common to other systems on the same network. For example, the example.com domain, or the top-level domain com. |
| fqdomainname | A fully-qualified domain name. This is the name of a system on a network that includes both the system name and the domain name. For example, www.example.com, where www is the system name. |
| ipaddress | A numeric network IP address value used when configuring a system's network settings. For example, 192.0.34.166. |
| netmask | A network mask value used when configuring a system's network settings. For example, 255.255.255.0. |
| etheraddress | An ethernet or MAC (media access control) address. This is the unique address assigned to the controller card in a system on an ethernet network. This is not the same thing as an IP address. |
| newsgroup | The name of a newsgroup. |
| osname | The name of an operating system. |
| filesystem | The name of a type (or format) of filesystem. For example, JFS, UFS, NTFS, or FAT32. |
| groupname | The name of a group of users on a system. This can also be used for any kind of group or role name in a security system. |
| username | The name used by a computer system to identify a |

| Attribute Value | Usage |
|---|---|
| | user. |
| library | The name of a library. This can be used to distinguish ordinary files from library files, or when a library is embedded in an archive file. |

Table 6.1 Values of the **class** Attribute of **systemitem**

The are a few other possible classes defined by DocBook, but you are unlikely to require them.

### 6.3.12 **database**

Like the systemitem element, the database element is a catchall element for content describing things related to database. To refine its meaning, you set its class attribute to one of the values shown in the table below (only the important values are shown).

| Attribute Value | Usage |
|---|---|
| name | The name of a database instance (not the name of the database product). |
| table | The name of database table. |
| record | The name or value of a database record or row. |
| field | The name of a field or column of a database table, or the value of a field within a record. |
| key1 | The name or value of a primary key field. |
| key2 | The name or value of a secondary key field. |

Table 6.2 Values of the **class** Attribute of **database**

### 6.3.13 **foreignphrase**

Use this for a foreign word or phrase in common usage. For example, *a priori*, *en masse*, *ad hoc*, etc. This markup may help translators who are localizing a document and will ensure that the phrase is rendered appropriately (in print media, an italic font is typically used).

### 6.3.14 **command**

Use this for the name of a command, usually a simple command executed at the command line. The application element can be used when describing software that is not usually run from a command line.

```
<para>
  Build the application by running the
  <command>build</command>.
```

```
</para>
```

Build the application by running the **build** command.

### 6.3.15 `option`

Use this for an option to a command. This will often be used in combination with a `command` element . For example:

```
<para>
  Check the spelling of your document by running
  <command>docmaker</command> <option>spell</option>.
</para>
```

Check the spelling of your document by running **docmaker**
`spell`.

It is more usual to combine the `command` and `option` elements within a `userinput` element if your are directing a user to type in the text. See Section 6.5, *Command-line Environments*, for more information.

### 6.3.16 `application`

Use this for the name of an software application. This is reserved for major applications, not simple commands. For example, *OpenOffice.org Writer* is an application but **dir** is a command.

### 6.3.17 `productname`

Use this for the formal name of a product (not an application or command). For example, *Microsoft Office* is a product, but *Microsoft Excel*, depending on context, could be a product or an application. The `class` attribute can be used with the same values as the `trademark` element to add an appropriate trademark symbol, though you should not have to use that attribute in normal circumstances. However, as the default class is `trade` and the appearance of a trademark symbol for every occurrence of a product name is not desirable, no trademark symbol will be shown when the `trade` class is specified or shown by default when no class is specified. In general, you only need to use this element if the formal product name is a trademark that may need to be acknowledged in the document's legal notice.

The trademark symbol (or other symbol depending on the class specified) will be rendered when the `productname` element is defined within the `legalnotice` element, but not elsewhere. To override this behavior, use the more explicit `trademark` element instead.

### 6.3.18 `trademark`

Use this for word or phrase that is trademarked. The `class` attribute can be set to a number of values if required: `trade` for a trademark, `registered` for a registered trademark, `service` for a service mark, and `copyright` for a copyright. The renderer will then insert the appropriate symbol. If no `class` value is specified, Cúram documents will assume the value `trade`. This element should rarely be used outside of a document's legal notice. You will typically use `productname`, `application`, or other, more specific element.

### 6.3.19 `acronym`

Use this for a word, often pronounceable, made by the initial or selected letters of the words in a phrase. For example, UIM, SDEJ. If the acronym is the name of a product, a trademark, or other term in common usage, if should *not* be marked with this element. There are more appropriate elements available for that content.

### 6.3.20 `abbrev`

Use this for an abbreviation, often one that ends with a period. Only use abbreviations when they are in common usage and always mark them with this element. For example, in informal internal documents, Tech. Inf. could be used as an abbreviation of "Technical Infrastructure". By using this element and the `acronym` element correctly, it will be possible to automatically generate a list of abbreviations for a document.

### 6.3.21 `medialabel`

Use this for the label on some physical medium, for example, the label on an installation CD-ROM.

```
<para>
  Insert the disk labeled <medialabel>Disk 1</medialabel>.
</para>
```

Insert the disk labeled *Disk 1*.

### 6.3.22 `remark`

Use this for a remark or comment in the text of the document that you do not intend to be rendered in the final output, though it may appear in draft documents. This is useful for placeholders or review comments while a document is being written. This element can also be used as a block element in almost any context. You should not nest one `remark` element within another.

> **Important**
>
> If you include remarks in a Cúram document, make sure that you

end the remark with your name or initials, so other authors can identify you as the author.

### 6.3.23  **superscript**

Use this for a superscript, for example "$x^2$". You are unlikely to use this often.

### 6.3.24  **subscript**

Use this for a subscript, for example "$H_2O$". You are unlikely to use this often.

## 6.4  Program Source Code

### 6.4.1  Overview

Documentation may include `programlisting` block elements containing samples of program source code. In Cúram documents, this source code will *not* be marked up with inline elements defining the elements of the code (though markup can be added to emphasize code). However, where the body of the text refers to keywords, identifiers, or values in the source code presented, or in documentation for APIs, appropriate inline elements should be used.

DocBook supports two major sets of markup elements for program source code. One is suited to object-oriented programming languages (e.g., Java or C++), and the other to procedural languages (e.g., C). There are too many elements to be easily manageable, however, so a generalized subset of all the elements will be used in Cúram documents as defined below.

When describing XML or HTML markup, a single `sgmltag` element is used for all references to syntax components.

### 6.4.2  **sgmltag**

Use this element when documenting the syntax of SGML, XML, or HTML documents. The DocBook format was originally defined in terms of SGML, and XML is a subset of SGML, so the `sgmltag` element is used when describing any related markup languages. The DocBook XML standard may be extended in future with more explicit support for XML-specific markup. The `sgmltag` element can be used to mark element names, attribute names, attribute values, etc., by setting the value of the `class` attribute appropriately.

> 💡 The Default Class for sgmltag
>
> For Cúram documents, the convention is to omit the `class` attribute when just marking an element name, but to set an appropriate

value of the attribute for everything else. This convention is supported by DocMaker; the DocBook standard defines no such default value.

The available values of the `class` attribute are as follows:

| Attribute Value | Usage |
| --- | --- |
| attribute | The name of an attribute. For example, the `class` attribute. |
| attvalue | The value of an attribute. For example, this is the `attvalue` value of the `class` attribute. |
| element | The name of an element. DocMaker will assume this value if you omit the `class` attribute. |
| emptytag | A tag used for an element with no body content. For example, the HTML element `<br/>` used for a line-break. |
| endtag | A closing tag used for an element with body content. For example, `</para>`. |
| genentity | The name of a general entity. These can be external general entities used to include the content of external files into a document, or internal general entities that associate a name with a character or string. For example, the internal general entity `&euro;` is used to represent a Euro currency symbol. |
| numcharref | A numeric character reference. These are an alternative to named entities for characters. For example, `&#169;` is the reference for the copyright symbol. |
| paramentity | The name of a parameter entity. These are used for advanced DTD authoring, so you should find little use for them in Cúram documents. |
| pi | An SGML processing instruction. As Cúram does not include any SGML content, you probably want to use `xmlpi`, not this class. |
| sgmlcomment | A comment in an SGML or XML document. For example, `<!--My comment-->`. |
| starttag | An opening tag used for an element with body content. For example, `<para>`. |
| xmlpi | An XML processing instruction. Processing instructions look a bit like elements with attributes, but the attributes are only pseudo-attributes: they are not separate document nodes. Everything between the opening "<?" and closing "?>" of a processing instruction is the processing instruction "data". You should type in this data, includ- |

| Attribute Value | Usage |
|---|---|
| | ing the equals signs and quotation marks for the attributes, and mark everything with a single `sgmltag` element. For example, `<?dbhh topicname="HomePage"?>`. |

Table 6.3 Values of the **class** Attribute of **sgmltag**

You should not add extra, syntax-related characters to the content of document when using the sgmltag element; angle brackets, quotation marks, ampersands, hashes, slashes, and other characters will be added automatically during rendering. Here are some examples:

```
<para>
  The <sgmltag>book</sgmltag> element starts with an opening
  <sgmltag class="starttag">book</sgmltag> tag. You should set
  the value of the <sgmltag class="attribute">role</sgmltag>
  attribute to <sgmltag class="attvalue">user</sgmltag> if the
  book is a user guide. If there is an ampersand in the text,
  use the <sgmltag class="genentity">amp</sgmltag> entity in
  its place.
</para>
```

> The book element starts with an opening <book> tag. You should set the value of the role attribute to user if the book is a user guide. If there is an ampersand in the text, use the &amp; entity in its place.

### 6.4.3 **classname**

Use this for the name of an object-oriented class or exception; or the name of a C or C++ struct.

### 6.4.4 **interfacename**

Use this for the name of an object-oriented interface.

### 6.4.5 **methodname**

Use this for the name of a method, member function, function, procedure, subroutine, or other equivalent for the programming language in question.

### 6.4.6 **type**

Use this for the name of a type. This should be used for primitive data types such as int or float, or for C or C++ typedefs. You can also use this for the names of domain definitions. Use the classname element for types implemented as classes.

For stereotype names used in UML modeling, DocMaker supports the use

of the `role` attribute with the value `stereotype`. This will render the conventional angle bracket characters around the stereotype name as in this example:

```
<para>
  A standard readmulti operation has a stereotype
  <type role="stereotype">readmulti</type>.
</para>
```

A standard readmulti operation has a stereotype `<<readmulti>>`.

### 6.4.7 **parameter**

Use this for the name of a parameter or argument to a method, member function, procedure, subroutine, or other equivalent for the programming language in question.

### 6.4.8 **varname**

Use this for the name of a local variable, member variable (C++), `struct` field (C), or field (Java). Environment variables should use the `envar` element, not this element.

### 6.4.9 **constant**

Use this for the name of a constant. For constants related to error messages, use the `errorname` element instead.

### 6.4.10 **property**

Use this for the name of a property. This could be a property in a configuration file or resource file, a JavaBeans property, or a system registry key name.

### 6.4.11 **literal**

Use this for a literal value. This could be a primitive data value, a return value from a function, or the value of a variable, property, or constant, etc. For example, "set the `width` property to `17px`". Do not use this for anything other than these values. It is not a generic element for something related to programming!

### 6.4.12 **symbol**

Use this for a keyword in a programming language (other than primitive type names).

### 6.4.13 **errorcode**

Use this for the unique number identifying an error.

### 6.4.14 **errortext**

Use this for the text of an error message. Many Cúram error messages use placeholders like "%1s", these placeholders should be marked with the `re-placeable` element and the whole message marked with the `error-text` element.

### 6.4.15 **errorname**

Use this for the name of an error. This is not the same as the `errortext` element. For example, you can have a *File Not Found* error (that is its name) with the error text, ERROR: File not found: hello.txt. In Cúram documents, you should also use this element to mark the constants associated with error messages.

### 6.4.16 **errortype**

Use this for the type or category of an error. For example, errors might fall into the categories *warning*, *error*, or *fatal*.

## 6.5 Command-line Environments

### 6.5.1 Overview

When describing command-line environments, it is important to have a consistent way of specifying what the user will see on the screen and what text they should type. The following inline elements can help (and can be used in combination with some of the basic elements introduced earlier).

### 6.5.2 **prompt**

Use this for a prompt that appears in a command-line environment. For example, "wait for the `login:` prompt to appear".

### 6.5.3 **userinput**

Use this for a word or phrase that the user should type at the command-line. For example:

```
<para>
  At the <prompt>password:</prompt> prompt, type
  <userinput>tiger</userinput> and hit
  <keycap>Enter</keycap>.
```

```
</para>
```

At the `password:` prompt, type **tiger** and hit *Enter*.

You can also use this element to group more complex content, for example, the description of a command and its options that should be typed in and executed.

### 6.5.4 **computeroutput**

Output from a computer program. This can be used for messages or other responses received after running a command or interacting with the system in some other way.

### 6.5.5 Combining Elements

For more complex user-input at a command line, use a `blockquote` element to contain the content and use `para` elements within that for each line of input. Build up the command using the appropriate inline elements.

```
<para>
  Run the following command from the shell (replacing
  <replaceable>year</replaceable> as appropriate):
</para>
<blockquote>
  <para>
    <prompt>$</prompt>
    <userinput>
      <command>dir</command> <option>/x</option>
      <filename>tax-<replaceable>year</replaceable>.xls<filename>
    </userinput>
  <para>
</blockquote>
```

Run the following command from the shell (replacing *year* as appropriate):

$ **dir /x tax-*year*.xls**

## 6.6 User Interfaces and User Interaction

### 6.6.1 Overview

When describing user-interfaces and user interactions with user-interfaces, there are a number of inline elements that can be used. If you use these elements correctly, you will guarantee that your documents will allow the generation of consistent, well-formatted presentation of content such as menu choices and keyboard shortcuts.

### 6.6.2 **mousebutton**

Use this to refer to a mouse button. The body of the `mousebutton` element should contain the button name: use only left, right, or middle.

### 6.6.3 `keycap`

Use this for the text appearing on a key on the keyboard. For letter keys, always use the upper-case letter, as that is what is printed on the key. For keys displaying words, use a space between the words, for example, *Num Lock*, or *Page Up*. Some keys do not have names or characters printed on them, so use the one of the following in those cases: *Enter*, *Space*, *Backspace*, *Tab*, *Shift*, *Up Arrow*, *Down Arrow*, *Left Arrow*, *Right Arrow*, *Option*, etc.

### 6.6.4 `keycombo`

When instructing a reader to press certain combinations of keys, you should nest the relevant `keycap` elements within a `keycombo` element. This will ensure that the correct separator is added during rendering; you do not have to invent your own! Here are some examples:

```
<para>
  Press
  <keycombo>
    <keycap>Ctrl</keycap>
    <keycap>S</keycap>
  </keycombo>
  to save your file.
</para>
```

> Press *Ctrl-S* to save your file.

The whitespace characters within the `keycombo` element are not significant, so it is easy to format them in a text editor so that they are easy to identify. Be careful, however, not to place whitespace between the `</keycombo>` closing tag and any following punctuation character.

It is not unusual to use certain keys in combination with mouse-button clicks, and this can also be represented with the `keycombo` element as follows:

```
<para>
  Click
  <keycombo>
    <keycap>Ctrl</keycap>
    <mousebutton>left</mousebutton>
  </keycombo>
  to extend your selection.
</para>
```

> Click *Ctrl*-left to save your file.

For keys that have more than one symbol on them, usually punctuation characters, you do not have to indicate that the *Shift* key should be held to while pressing the key. For letters, however, include the *Shift* key in the key combination to indicate if the letter should be upper-case, as the key is only

identified by the upper-case letter appearing on it and pressing it without a modifier key would yield a lower-case letter.

DocBook also supports the concept of key combinations where keys are pressed simultaneously (like the common *Ctrl-S* to save a file), and where keys are pressed in sequence, but this distinction is not supported in Cúram documents.

### 6.6.5 `guibutton`

Use this for the name of a button in a dialog box, on a web page, or elsewhere. Buttons include push-buttons, check-boxes and radio-buttons.

Take care to use the correct case for the button name; the label on an *OK* button, for example, is almost always all in upper-case; of course, sometimes there really is an *Ok* button, just to keep you on your toes.

### 6.6.6 `guilabel`

Use this for the label of a component in a dialog box or window other than buttons, icons and menus. Typically, it is used for the labels associated with text fields and list boxes. You should also use it for the values that appear in list boxes.

### 6.6.7 `guiicon`

Use this for the text or image associated with an icon or tool-bar button. If the icon is to be included, use the `inlinemediaobject` element.

### 6.6.8 `guimenu`

Use this for the name of a menu on the main menu bar of an application or for a pop-up menu. For example, "open the *File* menu".

### 6.6.9 `guisubmenu`

Use this for the name of a sub-menu within a menu of an application. For example, "open the *New* sub-menu of the *File* menu". Sub-menus normally appear in menus with an arrow indicating that another menu will be displayed when the sub-menu is selected.

### 6.6.10 `guimenuitem`

Use this for the name of a menu item within a menu or sub-menu that triggers an action of some kind.

It is common practice in user-interfaces to append an ellipsis ("...") to menu items that trigger actions that require further user input (usually via a dialog box). As this is rarely an actual ellipsis character, just use three full stops; don't leave them out. Of course, some menu items do not include an ellipsis

when they should; you must also omit them in this case.

```
<para>
  Open the <guimenu>File</guimenu> and select the
  <guimenuitem>Save As...</guimenuitem> menu item.
</para>
```

Open the *File* menu and select the *Save As...* menu item.

### 6.6.11 `menuchoice`

It is common to direct a user to a menu item via a menu or sub-menu. To ensure that such directions are presented uniformly throughout all documents, the menuchoice element should be used. This allows a path to a menu item to be defined with ease and provides a consistent look by adding the separating characters or glyphs automatically upon rendering. Do not include the separators in the content.

Like the keycombo element, the whitespace used within the menuchoice element is not significant.

```
<para>
  To create a new document select
  <menuchoice>
    <guimenu>File</guimenu>
    <guisubmenu>New</guisubmenu>
    <guimenuitem>Document...</guimenuitem>
  </menuchoice>
  from the menu. Alternatively, you can press
  <keycombo>
    <keycap>Alt</keycap>
    <keycap>Shift</keycap>
    <keycap>N</keycap>
  <keycombo>.
</para>
```

To create a new document select *File→New→Document...* from the menu. Alternatively, you can press the *Alt-Shift-N* keys.

## 6.7 Footnotes

Footnotes are notes set off from the main text that usually provide an explanation, a definition, or a relevant reference for a word or phrase appearing in the text. They are usually rendered as a numbered superscript within the text and a correspondingly numbered block of text at the bottom of the page or in a list of notes at the end of a book or chapter. In DocBook, the content of a footnote element is inserted inline in the text where the superscript marker will appear (though the footnote element can contain block elements).

Using Footnotes

In Cúram documents, a `footnote` element should contain only one `para` element. If you feel the need to add further paragraphs, you are probably overusing footnotes and should instead use an admonition element, a `sidebar` element, or a cross-reference to more detailed information in an `appendix`; or consider just including the content in place.

Here is an example, you should be able to use the superscript of the note in the text to locate the content:

```
<para>
  An application model is defined in <acronym>UML</acronym>
  <footnote>
    <para>
      <acronym>UML</acronym> stands for <quote>Unified
      Modeling Language</quote>.
    </para>
  </footnote>
  using a <acronym>UML</acronym> modeling tool.
</para>
```

An application model is defined in UML[1] using a UML modeling tool.

Footnotes tend to be a distraction; the above would probably be better written like this:

```
<para>
  An application model is defined in the Unified Modeling
  Language (<acronym>UML</acronym>) using a
  <acronym>UML</acronym> modeling tool.
</para>
```

An application model is defined in the Unified Modeling Language (UML) using a UML modeling tool.

DocMaker renders footnotes differently depending on the output format. In PDF output documents, the footnotes are rendered as *endnotes*: all of the footnotes in a chapter or appendix are listed on a separate "Notes" page at the end of that chapter or appendix. The superscript reference mark that appears in the text is a hyperlink to the note content on the "Notes" page. The "Notes" page does not appear in the table of contents or in the PDF bookmarks as it is only incidental to the structure of the document. In HTML output, the footnotes are rendered at the bottom of the respective HTML page and the superscript reference marks in the text are hyperlinks to the footnote content.

## 6.8 Cross-References

### 6.8.1 Introduction

Cross-references are used to direct a reader to another location in a docu-

ment or set of documents. Usually, the text of the reference is not explicitly defined in the source content by the author, but generated during rendering from the title of the target element. Cross-references in the output documents are automatically rendered as hyperlinks to the target content. DocMaker supports internal cross-references where a reference is made to another element within the same book; the cross-reference can target almost any element that can contain a `title` element.

DocMaker also supports a mechanism to make external cross-references to targets outside of the current book, but these only become hyperlinks when rendered to the HTML Help format, and the link text is not generated automatically. External cross-references are the mechanism used to support context-sensitive access to online help.

## 6.8.2 Internal Cross-References

There are a number of different cross-reference elements defined by the DocBook XML standard, but for cross-references internal to a document, Cúram documents will only use the `olink` element (the `ulink` element, described in Section 6.3.9, *ulink* , can be used for external hypertext links). Typically, a cross-reference requires that the reference refer to a specially marked target element elsewhere in the document. For Cúram documents, the marking of the target is automatic. Cross-references can target the following elements: `chapter`, `appendix`, `section`, `example`, `figure`, and `table`. Cross-references can also target the following elements if they include a `title` element: `note`, `tip`, `important`, `warning`, `variablelist`, `itemizedlist`, and `orderedlist`.

Unlike the `ulink` element, the `olink` element does not need to contain the link text. The text for the cross-reference will be automatically generated in a standard format during rendering. If necessary, you can override the automatically generated text by placing alternative text in the body of the `olink` element, as you would in the body of the `ulink` element.

To specify the target of the `olink` cross-reference, the `targetptr` attribute is used. The value of this attribute uses the `title` element value of the target element to identify the target of the cross-reference. The `targetptr` attribute should contain the type of cross-reference (`chapter`, `appendix`, `section`, `example`, `figure`, `table`, `note`, `tip`, `important`, `warning`, `variablelist`, `itemizedlist`, or `orderedlist`) followed by a colon and the text of the title of the element. The title need not be the full title but should contain enough of the title to uniquely identify it. Matching of the title is case-sensitive and words should be separated by a single space. Any markup in the element's title is ignored in the cross-reference look-up. The following shows an example of a chapter cross-reference:

```
<olink targetptr="chapter:Sample.txt File"/>
```

The following chapter title (containing markup) will be matched by the above cross-reference:

```
<book>
  ...
  <chapter>
    <title>The <filename>Sample.txt</filename> File</title>
    ...
  </chapter>
  ...
```

Document processing will halt if more than one title matching a cross-reference is found. Each of the matching titles will be listed and you should choose an unambiguous title string. If the titles are identical, or if the target title is a substring of another title, it will not be possible to disambiguate the cross-reference. In this case, you must replace the value of the targetptr attribute with the unique identifier shown in square brackets after the title text in the error message. This unique identifier is based on the title of the target element and of all its ancestor elements. If a target element has the same title as another element at the same level (a sibling element), then it will not be possible to cross-reference it without changing the titles so that they do not conflict.

Here is an example of a chapter reference:

```
<book>
  ...
  <chapter>
    <title>My First Chapter</title>
    ...
  </chapter>
  ...
  <chapter>
    <title>My Second Chapter</title>
    <section>
      <title>Introduction</title>
      <para>
        Before proceeding,
        read <olink targetptr="chapter:First"/>.
      </para>
    </section>
  </chapter>
  ....
</book>
```

Here is an example of a section reference:

```
<book>
  ...
  <chapter>
    <title>The Facts</title>
    <section>
      <title>Introduction</title>
      <para>
        There are some important facts presented in
        <olink targetptr="section:Facts"/>.
      </para>
    </section>
    <section>
      <title>Important Facts</title>
      <para>
        Some important facts.
      </para>
    </section>
  </chapter>
  ....
</book>
```

If the target is an `appendix`, `table`, `example`, `figure`, or other sup-ported element, the same scheme is used.

### 6.8.3 Context-Sensitivity

DocMaker allows you to associate arbitrary *topic names* with parts of your DocBook content. These topic names are used by an application when launching context-sensitive help, as the application has no knowledge of the URLs for each help page. An application only needs to indicate the required topic when launching the online help; the Help Applet will ensure that the correct HTML page is displayed and that the respective entry in table of contents is selected automatically. The topic names used for context-sens-itive online help can also be used when making cross-references from within one book to content within another book in the same group. You will see this later.

In Cúram applications, the ID of the current page is used automatically as the topic name when online help is invoked from within an application page. When you write content that will be used in the online help, you must set the appropriate page ID as the topic name for the chapter, appendix, or sec-tion in the content that should be displayed when help for that page is re-quested.

The DocBook XML format does not have any explicit support for topic names, so DocMaker supports the use of XML *processing instructions* to associate topic names with the content. Here is an example:

```
<section>
  <?dbhh topicname="Application_home"?>
  <title>Application Home Page</title>
  <para>
    Read on to find out what you can do on this page.
  </para>
</section>
```

The processing instruction `<?dbhh topic-name="Application_home"?>` in the example above associates the page ID "Application_home" with the section "Application Home Page". Every time you need to associate a topic name with some content, just insert a processing instruction before the `title` element and set the relevant topic name. DocMaker will automatically build up an index for the Help Applet that associates topic names with the names of the generated HTML file con-taining the content for that topic.

#### Processing Instructions in XXE

Adding processing instructions to a document in a text editor is simple: just type the instruction as shown in the example above. The XXE editor also allows you to insert processing instructions, but, as these are not XML elements, the method used to insert one is a little different than normal.

To insert a processing instruction using XXE, place the cursor in the

text of the relevant title (or select the title element) and choose *Edit→Processing Instruction→Insert Processing Instruction Before* from the main menu. The processing instruction will appear above the title. Place the cursor within the new processing instruction and type **topicname="*MyPageID*"**, replacing *MyPageID* with the required page ID. With the cursor still in the processing instruction, choose *Edit→Processing Instruction→Change Processing Instruction Target...* from the main menu and enter **dbhh** in the text field of the dialog box that appears and hit the *OK* button. To save time, you can now copy and paste this processing instruction when needed and just change the topic name as appropriate.

You can associate more than one topic name with the same content by adding extra processing instructions; just add them, one after another, before the appropriate `title` element. Topic names are case-sensitive and should not contain "=", ":", or whitespace characters. If you need to include these characters in a topic name, precede each with a "\" character.

### 6.8.4 External Cross-References

You do not always have to use page identifiers as topic names. You can use arbitrary names to make cross-references to other books in a group. There are more restrictions on this method than there are for internal cross-references: you must supply the link text, as it is not automatically generated; and you can only target external content marked with a HTML Help topic name described above.

External cross-references are resolved dynamically by the Help Applet deployed with HTML Help output. The Help Applet must be configured to present all of the books that may be targets for external cross-references, though this is done automatically when you generate HTML Help output for several books together. In PDF output, the link text is displayed, but is not a hyperlink. You should use the `citetitle` element to mark the link text so that it appears appropriately whether or not it is a hyperlink.

Once you have met all of these requirements, making external cross-references is straightforward: you just add an `olink` element to your content and set the `targetptr` attribute to the target topic name prefixed with "topicname:" and include the link text in the body of the `olink` element.

Here is an example: the first book contains a cross-reference to external content identified by a topic name; note that the `olink` contains the link text.

```
<book>
  <title>My First Book</title>
  ...
  <chapter>
    <title>The Facts</title>
    <section>
      <title>Introduction</title>
      <para>
        There are some important facts presented in
        <olink targetptr="topicname:ImportantFacts">the
```

```
        section, <citetitle>Important Facts</citetitle>
        in <citetitle>My Second Book</citetitle></olink>.
      </para>
    </section>
  </chapter>
  ....
</book>
```

The above external reference will resolve to the section shown in the second book below that is marked with the matching topic name.

```
<book>
  <title>My Second Book</title>
  ...
  <chapter>
    <section>
      <?dbhh topicname="ImportantFacts"?>
      <title>Important Facts</title>
      <para>
        Some important facts.
      </para>
    </section>
  </chapter>
  ....
</book>
```

This external cross-reference mechanism will work when the target is within the same book (like this reference to the *Introduction* section of this chapter), but you should use the internal cross-reference mechanism in preference, as the link text is automatically generated and a hyperlink will be created in all output formats.

## 6.9  Inline Images

Occasionally, documents may require the insertion of images inline in the text. For example, documentation for a tool-bar may display a table of tool-bar button images and descriptions, or include the images in instructions to the user to press the button. To insert such images, use the `inline-mediaobject` element. The content of the element is identical to that of the `mediaobject` block element (see Section 5.4.3, *mediaobject* ).

## Notes

[1] UML stands for "Unified Modeling Language".

# Chapter 7

## Character Entities

## 7.1 Entities

Entities are references to characters, strings, or files. They are used when it is not possible, or not preferable, to use the referenced object directly. In the world of SGML and XML, there are general entities and parameter entities and you can have internal and external forms of either; here, only internal general entities are described.

A common use of entities is to allow words like *Cúram* to be included in documents: on some systems it can be difficult to type the "ú" character, or the document file may not support the encoding of that character. In these cases, the character can be replaced with the corresponding character entity `&uacute;`. When the document is formatted, the entity will be replaced with the appropriate character. You can see it in action in this example:

```
<blockquote>
  <para>Use a character entity like this: C&uacute;ram.</para>
</blockquote>
```

That example will be rendered like this:

> Use a character entity like this: Cúram.

The name of a character entity must be defined in a document type definition (DTD). The DTD associates the name with the numeric code allocated to that character in the character encoding scheme. If the DTD does not define a named entity for a character that you require, you can use a numeric character reference instead of a character entity. For example, the "ú" can be represented as `&#250;`, where "250" is the decimal code for the character. If you want to use the hexadecimal code, you add an "x" before the code like this: `&#xFA;`. The DocBook XML DTD defines hundreds of named character entities, so you will rarely need to use the numeric references. The decimal codes used in character references can include codes for almost any

Unicode character.

### Entities in the XXE Editor

The *XXE* editor will automatically convert numeric character references to named character entities where the entities are defined in the DocBook XML DTD. *XXE* will also convert entities and character references into ordinary characters where a document's encoding supports those characters.

### Typing Accented and Other Characters

On some operating systems, you can type any of vowels with an acute accent by using the *Alt Gr* key. For example, *Alt Gr-U* for "ú" or *Alt Gr-Shift-U* for "Ú".

On *Microsoft Windows* operating systems, you can also type characters if you know their codes. The codes are entered on the numeric keypad and *Num Lock* must be turned on. While holding down the *Alt* key, type *0* (zero) on the numeric keypad followed by the digits in the character code, then release the *Alt* key. The character will appear (if your text editor has not mapped the combination as a shortcut to some operation). The codes used are those defined in the ANSI character encoding, but as this is a superset of the ISO Latin 1 encoding, it can be used with a little care. It will not work for Unicode characters above 255 and should not be used for codes in the range 128-159, as these are not valid in ISO Latin 1 or Unicode, only in ANSI.

### Character Representation

Although you can represent any character you want in your source document using entities or character references, your chosen character may not be represented correctly when you render your document to another format that does not use entities. DocMaker renders output documents that use the UTF-8 character encoding and replaces entities in the ISO Latin 1 input documents with the corresponding UTF-8 character code. If the font used to view the output document does not define a glyph for that character code, you will not see the character as you expected. This does not mean that the document has been corrupted, only that the font used is not suited to the content.

When you use a text editor, you must take note of the character encoding used for your document (see Chapter 9, *Editing DocBook XML Documents* for details). As Cúram documents use the ISO Latin 1 encoding, you will not need to use `&uacute;`: it is included in that encoding. However, there are other characters that you may want to use that are not included in this encoding or are not as easy to type on the keyboard. The following sections list some of the common character entities that you can use in your DocBook documents. When using *XXE*, you can select characters visually, but it helps to know the correct code to ensure that you are using the correct character—different dash characters can be hard to tell apart, for example. These

lists are not exhaustive: they contain only the characters that you are likely to need in Cúram documents.

## 7.2 Punctuation Marks and Accented Characters

The accented characters all follow the same naming convention: the unaccented character in upper-case or lower-case followed by the name of the type of accent in lower-case is used. For example, "Ú" is represented using `&Uacute;` (note the upper-case "U") and "è" is represented using `&egrave;` (lower-case "e").

The full list of the available characters is too long to present in this overview, so consult any ISO Latin 1 character set reference should you need the codes. The online Unicode code charts, http://www.unicode.org/charts/, are a good reference. The "Basic Latin" and "Latin-1 Supplement" charts cover the ISO Latin 1 character set.

| Entity Name | Code | Representation | Description |
|---|---|---|---|
| mdash | 8212 | — | Em dash. |
| ndash | 8211 | – | En dash. |
| hellip | 8230 | … | Horizontal ellipsis. |
| incare | 8453 | # | Care of. |
| uacute | 250 | ú | Lower-case letter u with acute accent. |
| Uacute | 218 | Ú | Upper-case letter U with acute accent. |

Table 7.1 Punctuation Marks and Accented Characters

## 7.3 Numeric and Symbol Entities

The first five entities in this table are the reserved entities defined for all XML documents. They are used to indicate to an XML parser that the character is meant literally and is not related to its syntactical use in XML. You must use these codes in your content and in the values of attributes to XML elements to avoid syntax errors.

| Entity Name | Code | Representation | Description |
|---|---|---|---|
| lt | 60 | < | Less-than sign. |
| gt | 62 | > | Greater-than sign. |
| amp | 38 | & | Ampersand. |
| quot | 34 | " | Double quotation mark. |

| Entity Name | Code | Representation | Description |
|---|---|---|---|
| apos | 39 | ' | Apostrophe. |
| plusmn | 177 | ± | Plus-minus sign. |
| divide | 247 | ÷ | Division sign. |
| times | 215 | × | Multiplication sign; not the same as the letter "x" |
| euro | 8364 | € | Euro currency symbol. |
| cent | 162 | ¢ | Cent symbol for US currency. |
| yen | 165 | ¥ | Japanese Yen currency symbol. |
| micro | 181 | µ | Micro sign. |
| deg | 176 | ° | Degree sign. |
| para | 182 | ¶ | Pilcrow or paragraph sign. |
| copy | 169 | © | Copyright symbol. |
| reg | 174 | ® | Registered trademark symbol. |
| trade | 8482 | ™ | Trademark symbol. |
| nbsp | 160 |  | Non-breaking space. |
| shy | 173 | - | Soft hyphen. |

Table 7.2 Numeric and Symbol Entities

# Chapter 8

## Output Formats

## 8.1 Introduction

DocMaker can convert DocBook XML content into a number of different formats. At present, the generation of output in the portable document format (PDF) and the HTML Help format are officially supported and actively maintained. While DocMaker can also generate plain HTML output, this feature is not yet officially supported.

Your content in DocBook XML format is used as the source for all output formats: you do not need to maintain different content to support each output format. Generating output in different formats is as simple as running the appropriate DocMaker target: `generate-pdf` for PDF and `generate-help` for HTML Help. However, as these output formats fulfill different purposes and are used in different ways, DocMaker includes some features to help you get the most out of your content: you can make your help system context-sensitive, and you can include or exclude content depending on the output format selected.

## 8.2 Portable Document Format

The Adobe portable document format (PDF) is a popular format for distributing high-quality documentation. Files can be opened using the free Adobe *Acrobat Reader* application, or other compatible readers. PDF documents differ from HTML documents in their presentation; unlike HTML, PDF documents are *paginated* and are ready for printing to paper if required. DocMaker supports many PDF features to make these documents as useful as possible: the table of contents is generated automatically; bookmarks are created to allow the PDF reader application to display the table of contents independently beside the document content; all entries in the table of contents, all cross-references and all external references are active hyperlinks; headers and footers with titles and page numbers are inserted; and cover

pages and color schemes are selected automatically to match the category of the document. All of these features are supported automatically by Doc-Maker and only the latter feature requires any configuration. To modify the cover page and color scheme, you must set the `role` attribute of the `book` element; see Section 4.3.3, *Book Categories* for more details.

## 8.3 HTML Help Format

### 8.3.1 Overview

DocMaker can convert DocBook XML content into HTML Help format. HTML Help a popular format and the files produced by DocMaker are compatible with many third-party help authoring systems from Microsoft, Macromedia and others. However, integration with such third-party systems is not the primary purpose of DocMaker; DocMaker provides everything required to create and deploy a fully-functional online help system without the need for any other tools.

DocMaker not only generates the HTML Help files, it also includes two systems that can be used to access that help content: a Java Help Applet that displays the table of contents and search controls for the help content in the browser and allows applications to invoke the online help in a context-sensitive manner; and a Help Server, a server-side implementation providing the same functionality as the applet but that is deployed and executed in a J2EE web container. The applet has the advantage that it can be executed without a web container or web server—for example, from a CD-ROM—but, while it is accessible and meets modern accessibility guidelines, it does not meet the older W3C Web Content Accessibility Guidelines (WCAG) 1.0 that do not permit applets. The applet also requires that JavaScript is enabled for context-sensitive access to the help system. The Help Server requires a J2EE web container, but does not require an applet or JavaScript to support context-sensitive access to the help system or for any of its basic functionality. It is meets the WCAG 1.0 guidelines for accessibility. However, for improved usability, JavaScript should be enabled, as it is required to support the collapsing and expanding of the entries in the table of contents and for the synchronization of the table of contents to the current help content page. The Help Applet can be deployed along with the Help Server and both can be used simultaneously if desired, though a Cúram application can only be configured to launch context-sensitive help using one or the other.

For the most part, there is little that you have to do to generate online HTML Help other than to execute DocMaker's `generate-help` target. However, to support context-sensitivity and accessibility, some extra information is required in the DocBook XML content.

### 8.3.2 Accessibility

The online help for Cúram applications is required to be accessible by dis-

abled users. There are a number of extra requirements that your DocBook XML content must meet to ensure the online help will be accessible:

- Each table must have a summary describing its contents. See Section 5.5.4, *Accessibility* for more details.

- Each image must have an alternative text representation and a longer description describing its contents. See Section 5.4.4, *Accessibility* for more details.

When you run DocMaker's `check` target, and the `role` attribute of your `book` element is set to `user`, the accessibility of your content will be checked automatically. Once you meet these requirements, DocMaker will generate help content that is accessible.

### 8.3.3 Organization of HTML Output Files

Unlike the PDF output where a single PDF file is created, the HTML Help output is comprised of a large number of HTML files, Java archive (JAR) files and image files. All of these are created and located automatically, so there is no need for any manual intervention. However, it is important to be able to identify the main files and to understand how they are organized so that you can deploy your HTML Help content successfully.

The files are created below the `help` folder within the DocMaker output folder (usually called `doc`). Within the `help` folder, there will be one folder for each book generated. The `help` folder itself will contain the *top-level* help system that can be used to present all of the books in the set. The top-level help system is configured to run both the Help Applet and the Help Server. Each book folder also contains the necessary files to distribute help using the Help Applet only; the Help Server is not configured at this level. To make the Help Server easier to deploy, a single *help.war* archive file is located in the help folder and it contains a fully-configured J2EE application that can be deployed to a web container. This WAR file also contains the Help Applet, so, on deploying the help application, both the Help Server and the Help Applet can be served simultaneously. The WAR file contains all of the files in the help folder and sub folders, so, even if it is not to be deployed to a web container, it can be used to distribute the help system. Its contents could, for example, be extracted to a CD-ROM used to distribute applet-based help.

There are five main types of files created in the `help` folder and in the book sub-folders:

**HTML Content Files**

These are created in the book folders only, not at the top level. For each chapter and top-level section in your DocBook source content, a separate HTML file is created. These names of these files are generated automatically; they usually begin with the letters "DM" followed by a unique node number. Each file represents a single page of your HTML Help content.

**HTML Frameset Files**

To organize all the HTML content files, a HTML *frameset* is used. A frameset is a HTML file that defines how the display area of a web browser should be sub-divided into separate areas, each of which will display the contents of a separate HTML file. These areas are called *frames*. There are two frameset files, one for applet-based help and one for server-based help. The framesets are defined in the `index.html` and `index.jsp` files respectively. This `index.html` file defines two frames: on the left, a frame for the Help Applet configured in the `applet.html` file; and on the right, a frame for the help content that defaults to displaying the `book-index.html` file that contains the book title and legal notice. The `index.jsp` file contains four frames: on the left are three frames, one for the logo, one for the table of contents and one for the search results; the frame on the right is the same as that used for applet-based help. Both the top-level `help` folder and the book folders will contain an `index.html` file that can be used to start applet-based help. The `index.jsp` file appears only in the top-level `help` folder.

**Image Files**

Image files include both the images you have referenced from your DocBook content files and other images that are included automatically during rendering, such as the Cúram logo, the icons for tips and warnings, and navigation buttons. These latter images are shared by all of the generated books and are placed in a separate folder from the images explicitly referenced from your content.

**HTML Help Meta-data Files**

DocMaker generates a number of files that contain meta-data relevant to the development of a HTML Help system. These files use standard names are in standard formats that can be interpreted by any help development tools that support the HTML Help format. DocMaker only requires two of these meta-data files: the `toc.hhc` file containing the table of contents information that associates chapter and section titles with HTML content files; and the `alias.h` file that associates topic names with HTML content files. The other meta-data files are deleted automatically, but they can be made available if required.

**Java Archive (JAR) Files**

The Help Applet used to present the table of contents and support context-sensitivity is stored in a JAR file included in the DocMaker installation. This JAR file is copied into the HTML Help output folder for each book and the meta-data files are moved inside the archive. When a user opens the frameset containing the applet frame and the content frame, the JAR file is downloaded and the applet is executed. The applet then loads the table of contents and the topic names from these meta-data files and begins operating. The help server uses many of the same facilities as the help applet, but the servlet classes are packaged in an additional JAR file.

If you want to deploy your HTML Help content, you must preserve the way the files are organized when they are created by DocMaker. All of the files are created below the `help` folder of your output folder (the output folder is called `doc`, unless your build system has overridden the name). Within the help folder, you will see an `images` folder and one folder for each book, just as you would see them in your documentation root folder. The `images` folder contains the common icons supplied by DocMaker, and within each book folder, you will find the HTML content files, frameset files and the JAR file for the book; the image files referenced by a book are organized within the folder exactly as they were in the original DocBook book folder.

When deploying your help content, you must copy the relevant book folders to the required location and you must copy the `images` folder and place it in the same location relative to your book folders, i.e., the `images` folder must be copied to the same location as the book folders. If you do not preserve this relative location, the image references in the HTML content files will be invalid.

The HTML Help content can be deployed to a web server, or to a local filesystem (including a CD-ROM). The applet-based help system contains no dynamic content and does not require the execution of any scripts on a web server. The system is managed entirely by the Help Applet that is downloaded via the frameset. To access the content for a book, only the `index.html` frameset file needs to be opened. A link to this file can contain a topic name parameter, as described in Section 6.8.3, *Context-Sensitivity*, but this parameter is interpreted by the applet, so, again, no web server is required. The Help Server does require a web container to operate, but is otherwise similar to the Help Applet; the `index.jsp` frameset file can be passed the same topic name parameter as would be passed to the Help Applet (see below).

If you have several books containing HTML Help content, you can access them individually, as above, or as a single set of books. In the output `help` folder, along with the `images` folder and the output book folders, you will find another copy of the applet JAR file and some frameset files. The `index.html` file located here will access the help content using an applet that contains copies of the meta-data files for *all* of the books and thus displays a table of contents that shows each book at the root level and allows the user to navigate to any page in any book. Context-sensitive invocations also work as before. When deploying this Help Applet, it must be located in the same position relative to the book folders, as it only links to the HTML content files in those folders and the relative links must be preserved. The Help Server does not support help for a single book in a set of books, only top-level help for the entire set (though a set can contain only one book).

## 8.3.4  Invoking Context-Sensitive Help

Context-sensitivity and the use of page IDs as topic names was covered in Section 6.8.3, *Context-Sensitivity*. The details here describe how you can integrate context-sensitive help into an application.

When an application launches the context sensitive help, it must supply the topic name to the Help Applet or Help Server. This is accomplished by simply linking to one of two files, `index.html` or `index.jsp` respectively, and adding a topic name parameter to the URI for that link. The choice of Help Server or Help Applet and the generation of these links is handled automatically within a Cúram client application. For example, to request the online help (using the Help Applet) for the "Application_home" page, the URI would be `index.html?topicname=Application_home`, as long as the online help contains some content is marked with that page ID as the topic name. Depending on how the online HTML Help has been deployed, the full URL may include the name of the HTTP server and the path to the `index.html` file. For the URL to be valid, the topic name must be encoded using the method specified in RFC 1738 [http://www.ietf.org/rfc/rfc1738.txt]; however, most topic names will be unchanged, as the encoding is only necessary for certain reserved characters. Parameters are passed to the `index.jsp` file in the same manner.

> **ⓘ  Backward Compatibility**
>
> Older Cúram client applications used online help generated by Quadralay *WebWorks*. The URL for a context-sensitive link did not include the `topicname` parameter and its value, only the actual topic name; for example, `index.html?Application_home`. For backward compatibility with this scheme, DocMaker's Help Applet and Help Server support the omission of the `topicname` parameter and will then interpret the first parameter name as the name of the topic, but only when no other parameters are passed.

The Help Applet used for online HTML Help requires a browser that supports the Sun Microsystems Java Plug-in. If this plug-in is not installed, the table of contents will not be available and context-sensitive help will not work. Context-sensitive access to the help content also requires that the browser have JavaScript, and the scripting of Java applets, enabled, though the table of contents will work without either.

The Help Server does not require JavaScript support for context-sensitive help, though it does enhance the operation of the help system.

## 8.3.5  Online Help User Interface

### Instructions on Using the Help System

When delivering your online help content, you may want to provide your readers with instructions on how to navigate through the content. The navigation features available are described here so that you may include them in your own content or training materials. As this guide is a developer guide, it is not appropriate to assume that all users will have access to this content, or that they would realize that this is where they should go to learn about using the help system. Most users accessing the help system will be application

users, so your application help content is the most appropriate place to provide these instructions.

### Keyboard Shortcuts

The keyboard shortcuts used to navigate through the table of contents and the search results differ between the Help Applet and the Help Server. This inconsistency is not the result of a defect or failing of either system, but is simply the result of having two different user interfaces that each have different default keyboard shortcuts. The Help Applet uses a Java Swing graphical user interface and the keyboard shortcuts are the default ones that users will expect when using such an interface. Similarly, the Help Server delivers content in HTML format to the browser and the keyboard shortcuts are simply the standard shortcuts implemented by the browser to navigate through HTML content. No attempt has been made to make the shortcuts consistent, as this would likely be more confusing for users than the default—but different—shortcuts that they would expect.

## Help Applet

The Help Applet is one of two systems used to display the table of contents for one or more books and provide a means to search the content. It appears on the left of the browser window in its own frame. The table of contents is displayed as a tree of book, chapter, and section titles mirroring the hierarchical structure of the book. The results of a search are displayed in a list in order of relevance; each entry is the title of a section that matches the search terms and a tool-tip provides more information about the book and chapter in which that section was found.

Within the Help Applet, the mouse or the *Tab* key can be used to move the focus between the controls. To change between the Contents and the Search tabs using the keyboard, set the focus to the label of the active tab and use the *Left* or *Right* arrow key to switch to the other tab. The size of the font used in the Help Applet can be changed within certain limits: use *Ctrl--* to decrease the font size and *Ctrl-=* to increase it.

The *Contents* tab presents a tree control with a hierarchical view of the contents of the online help documentation. The use of the table of contents tree is very straightforward: double-clicking on a title, or clicking a "+" icon beside a title, will expand that title to display the titles of the contained elements; single-clicking on a title will open the corresponding help page. If using the keyboard, the arrow keys can be used to navigate the tree: the *Up* and *Down* arrow keys move through the visible tree entries and the *Right* and *Left* arrow keys expand and collapse the tree nodes respectively. If only one book is being managed by the Help Applet, then the book title in the table of contents will be expanded automatically to display the chapter titles when the applet first loads; otherwise, two or more unexpanded book titles will be displayed.

Searching is performed on the *Search* tab by entering some search terms in the *Search* text field and then pressing the *Enter* key or clicking the *Search* button. The results, if any, are then displayed in the list and the number of results is shown in the *Results* field below the list. The mouse or the *Up* and *Down* arrow keys can be used to select entries in the list. Selecting an entry in the results list will open the corresponding help page.

## Help Server

The Help Server, the alternative to the Help Applet, displays the table of contents and the search controls in separate frames to the left of the help content. The browser's keyboard shortcuts are used to navigate through the content in these frames. As these shortcuts are browser-dependent, the user should be directed to their browser's help information for details on their use. However, in most browsers, the *Tab* key is used to move from link-to-link within the frames and the *Enter* key is used to activate the selected link.

The Contents frame and the Search frame both display a heading at the top of the frame to indicate their purpose. When JavaScript is enabled, a button showing up and down arrows appears to the left of these headings and can be used to automatically resize the frames. Pressing the button in either frame will resize the frames so that the height of the larger of the two frames is decreased to the height of its heading; the height of the other frame will be increased to use all of the remaining space. This makes it easy to resize the frames so that the maximum amount of space is used to display the table of contents or the search results, as the user prefers. The contents of the frames will not be scrolled by this operation, so when a frame is enlarged again, the content will be in the same position as it was when the frame was contracted. With or without JavaScript, the frames can be resized manually to any size by dragging the frame borders with the mouse. By default, when the help system is first started, the frames will be of equal size.

The appearance of the table of contents differs depending on the availability of JavaScript. Without JavaScript, the table of contents appears as a hierarchy of entries with chapters indented below books and sections indented below chapters. Where an entry has some related content at a lower level (for example, a section containing sub-sections) it appears with a plain bullet icon to the left.

When JavaScript is enabled, the table of contents tree is modified automatically when the page is loaded. The tree is "collapsed" to show only the book titles and the plain bullet icons are replaced with "plus" icons to indicate that the nodes can be expanded if the icon is clicked. When expanded, the icon changes to a "minus" icon and the lower-level entries are displayed. Clicking the icon will now collapse the entry, hiding the lower-level entries and changing the icon back to a "plus" icon. This behavior is the same as the behavior of the table of contents tree in the Help Applet. For keyboard users, an entry can be expanded or collapsed using the + and - keys when the link for the entry has the input focus. In most browsers, the *Up* and

*Down* arrow keys can be used to scroll the contents of the frame.

Searching the content is similar to searching using the Help Applet: search terms are entered in the search text field and the search button is activated. Search results are displayed below the search text field. Each result displays a link to the matching section and the name of the book and chapter containing that section. The score displayed indicates the relevance of the result and the results are sorted with highest scores first. This is the same order as that used by the Help Applet, though the applet does not display the score values.

## Help Content Pages

One help page is generated for each top-level section in a book. The first section in each chapter appears on the same page as the title for that chapter. A single page is also generated containing the title of the book, the abstract of the book (if present), and the legal notice.

On each help page, a navigation tool-bar is displayed above and below the content of the page. The tool-bar at the bottom is always located so that it cannot be seen without first scrolling down. This avoids cluttering the presentation with two tool-bars when the content is short. The tool-bar is shown below. Not all of the tool-bar buttons are always visible; their presence depends on the location of the current page within the book or the availability of JavaScript. The buttons on the tool-bar are hyperlinked images that link to other pages or execute scripts. For accessibility, each button has an associated *access key* that can be used to change the input focus to that button without using the mouse. To activate a button using an access key, press the *Alt* key and the access key together to change the focus to the button and then press the *Enter* key. In some browsers, pressing the access key will immediately activate the link and you do not have to press *Enter*.



Figure 8.1 The Online HTML Help Tool-bar

From left-to-right, the buttons are as follows:

**Home**

Opens the home page for the current book containing the book title, abstract and legal notice. This button is not displayed on a home page. The access key for this button is *H*.

**Up**

Opens the page containing the content for the parent element of that presented on the current page. For example, when viewing a page containing the content for a section, the relevant chapter page (containing the chapter title and the first section) is opened; when viewing a chapter page, the book home page is opened. This button is not displayed on a home page. The access key for this button is *U*.

**Synchronize Table of Contents**

Selects and expands the title in the table of contents corresponding to the currently displayed page. The title will be scrolled into view automatically if it is not already visible. This is useful when a hyperlinked cross-reference or a search result jumps to another page and the user wants to determine the context for that page or read related pages. This button is only displayed when JavaScript is available, as synchronization requires JavaScript to perform the operation. The access key for this button is *S*. Synchronization works for both the Help Applet and the Help Server. When synchronizing the table of contents provided by the Help Server, the relevant entry in the table of contents is highlighted with a yellow background that quickly fades back to the normal background color. This helps to draw the user's attention to the entry, as the sudden jump in the table of contents can otherwise be disorienting.

**Previous Page**

Opens the previous page. This is the previous page in the context of a book, it is not the same as the browser's back button. This button is not displayed on a book home page, as that is the first page in a book and has no previous page. The access key for this button is *P*.

**Next Page**

Opens the next page. This is the next page in the context of a book, it is not the same as the browser's forward button. This button is not displayed on the last page of a book. The access key for this button is *N*.

When navigating through the online help content using the tool-bar buttons or hyperlinked internal cross-references, the table of contents is not automatically synchronized with the page content; a user can use the tool-bar button to do this manually at any time. However, when starting the online help using a topic name for context-sensitivity, or when activating a hyperlinked external cross-reference to a different book within a set of books displayed by the Help Applet, the table of contents is synchronized automatically to the target page. This synchronization is not automatic when using the Help Server.

Your browser will determine which keys you need to use to change the input focus between the control frames on the left and the help page frame, but few, if any, browsers provide a means to change the input focus to an applet or out of an applet. To assist you, a button is provided at the top of the applet frame that, when activated, will cause the input focus to change to the Help Applet. The access key for this button is *A*.

## 8.3.6 Suppressing Entries in the Table of Contents

By default, the table of contents includes entries for all chapters, appendices, and sections. However, for some types of online help, it may be desirable to suppress the entries for some of the content. For example, you may not want to display entries for an appendix that contains hundreds of sections, one for each possible error message that the system can report. To

suppress the inclusion of these entries, the `role` attribute of any `chapter`, `appendix` or `section` element can be set to `no-toc`. This will suppress the entry in the table of contents for the element and all of its descendant elements. When used on `section` elements, it is possible to suppress only some `section` elements within a `chapter` or `appendix`, and not all.

Setting the `role` attribute to `no-toc` on any other type of element will have no effect; you cannot apply this to a `book` element, for example. It will also have no effect on the table of contents displayed for output formats other than HTML Help.

### 8.3.7 Localizing Online Help

The Help Applet, Help Server and the boilerplate text in the help content can be localized to match the language used in the content. The localization of the Help Applet is achieved by adding a Java-style properties file to the applet's JAR file using Java's standard locale-specific naming convention. The boilerplate text in the help content is localized by adding entries to an XML properties file used by the XSLT stylesheets that generate the HTML.

To localize the Help Applet, you need to add an appropriate properties file to the Help Applet JAR file `lib\help-applet.jar` in your DocMaker installation. The English language properties file is already present, so follow these steps to add your localized text:

1.  Expand the `lib\help-applet.jar` file in a temporary location and locate the contained `HelpUI.properties` file in the extracted `curam\docmaker\tools\help` folder.

2.  Make a copy of the `HelpUI.properties` file in the same folder and append the locale name to the end of the new file name, separating the locale from the existing name using an underscore character. For example, for the "fr_CA" locale, the file should be named `HelpApplet_fr_CA.properties`.

3.  Edit this new file and localize the text for each property.

4.  Create a new JAR file from the entire folder structure extracted from the original JAR file, a folder structure that now includes the new properties file, and replace the existing `lib\help-applet.jar` file with this new JAR file.

To generate content with the correctly localized boilerplate text, the localized text needs to be added to the file `lib\xsl-custom\common\l10n.xml` in your DocMaker installation. This file already contains the basic English language text, so perform the following steps:

1.  Locate the `l:l10n` element and duplicate the element and all of its contents creating a second element alongside the first.

2.  In the copy of the `l:l10n` element, change the `language` attribute

to the name of your locale (just the language code can be used if the country code is not significant).

3. Within the `l:l10n` element, locate each `l:gentext` or `l:template` element (the latter are nested within `l:context` elements) and change the value of the `text` attribute of each to the text appropriate for the language being used.

There are only a limited number of elements that need to be localized in the `lib\xsl-custom\common\l10n.xml` file. Most of the localization for many common languages is already done. However, if there is no file in the `lib\docbook-xsl\common` folder for your required language (the files are named using the locale name and the `.xml` extension), you will have to create a file there by copying an existing file and making the appropriate changes to the `text` attributes. If this is required, you should also add an entry to the `lib\docbook-xsl\common\l10n.xml` file for your new locale following the format of the other entries.

By default DocMaker will generate content using the "en" locale (though "en_US" is used for spell-checking), but this can be overridden by setting the `docmaker.help.locale` property on the command line. For example, to set the locale to Spanish, the following command is used:

```
docmaker -Ddocmaker.help.locale=es gener-
ate-help
```

## 8.4 Conditional Content

### 8.4.1 Overview

When using a single source document in DocBook XML format to generate both PDF and HTML Help output, there is often a need to make small changes to the content appropriate to the way the output will be used. For example, a PDF document may refer to "page 26" or contain phrases such as, "In this book" or, "In this chapter"; while the HTML Help has no page numbering and might use the phrase, "In this online help". DocMaker automatically modifies the text generated for cross-references dependent on certain factors; for example, if the output format is paginated, or if section numbering is used. Alternative phrasing used in the content cannot be substituted automatically, but DocMaker supports *conditional content*: content that is explicitly marked by the author for use under specific conditions.

### 8.4.2 Conditions

DocMaker supports two conditions: *online-only* and *print-only*. Content marked as online-only will only appear in HTML Help files and not in PDF documents. Conversely, content marked as print-only will only appear in PDF documents and not in HTML Help file.

To mark content with a condition, you just add a `condition` attribute to the appropriate element and set the value to either `online-only` or `print-only`. These conditions can be applied to any DocBook XML element, though their use in the control of alternative phrasing is best implemented by applying them to `phrase` elements.

For example, say you have the following content:

```
<section>
  <title>Introduction</title>
  <para>
    In this book, you will learn many things.
  </para>
</section>
```

The phrase "In this book" is appropriate if you are only producing a PDF document, but you may want to use the words "online help" instead of "book" for your HTML Help system. As you may not wish to make the entire `para` or `section` element conditional, you can add the inline `phrase` element to limit the scope of the condition. The conditional content will then be marked as follows:

```
<section>
  <title>Introduction</title>
  <para>
    In this
    <phrase
      condition="print-only">book</phrase><phrase
      condition="online-only">online help</phrase>,
    you will learn many things.
  </para>
</section>
```

The formatting of the content is a little tricky: you do not want to introduce any unwanted whitespace between the conditional phrase and the comma that follows it. Of course, you could alter the scope of the conditional content as follows to achieve the same result:

```
<section>
  <title>Introduction</title>
  <para>
    In this <phrase condition="print-only">book,</phrase>
    <phrase condition="online-only">online help,</phrase>
    you will learn many things.
  </para>
</section>
```

Placing the comma inside the conditional phrase makes the formatting a little easier, but all this is really only a concern when formatting content manually in a text editor. XML editors will manage this for you.

The scope of a condition is not limited to phrases; you can make any element conditional. However, it would obviously be a bad idea to try to make a `book` element conditional, so DocMaker will ignore the `condition` attribute on a `book` element. In this example, a whole `section` is conditional:

```
<chapter>
```

```
  <title>Introduction</title>

  <section>
    <title>Who Should Read This?</title>
    <para>
      Anyone using the application should read this
      <phrase condition="print-only">book.</phrase>
      <phrase condition="online-only">online help.</phrase>
    </para>
  </section>

  <section condition="online-only">
    <title>Navigating Through the Online Help</title>
    <para>
      Use the links at the bottom of each page to navigate
      through the online help. You can also use the back
      button of your web browser to return to the previous
      page.
    </para>
  </section>

</chapter>
```

💡     Avoiding Conditional Content

You can make life easier on yourself by avoiding phrases like "In this book"; if the phrase is not there, you will not have to make it conditional!

### 8.4.3   Conditional Content and Cross-References

Some care is required when using conditional content. It is possible to create a cross-reference to conditional content, but you must ensure that the content containing the cross-reference is subject to the same condition as the content being referenced or the cross-reference will be invalid. Cross-references are resolved before the conditions are applied, so this type of invalid reference may not be detected.

If a title contains conditional phrases, the cross-reference must be configured as though all phrases were present. Here is an example:

```
<chapter>
  <title>Introduction</title>

  <section>
    <title>
      About This
      <phrase condition="print-only">Book</phrase>
      <phrase condition="online-only">Online Help</phrase>
    </title>
    <para>
      Everything you need to know about this application is
      contained in these pages!
    </para>
  </section>

  <section>
    <title>Other Resources</title>
    <para>
      If the statement in
      <olink targetptr="section:About This Book Online Help" />
      was misleading, consider taking a training course to
      find out more about this application.
    </para>
```

```
    </section>

</chapter>
```

Note how the value of the `targetptr` attribute of the `olink` element contains the text from both conditional phrases. When the output documents are generated, the titles and the generated text for the cross-references will appear with the correct conditions applied. The requirement to include the text from all the phrases is just need for the initial resolution of the cross-reference. It is also unlikely that you will ever need to make a cross-reference to an element with conditional phrases in its title, so this limitation is not significant.

You will need to take care to use the correct whitespace between the phrases when setting the value of the `targetptr` attribute. Here is a slight variation on the above example showing this:

```
<chapter>
  <title>Introduction</title>

  <section>
    <title>
      About This
      <phrase
        condition="print-only">Book</phrase><phrase
        condition="online-only">Online Help</phrase>
    </title>
    <para>
      Everything you need to know about this application is
      contained in these pages!
    </para>
  </section>

  <section>
    <title>Other Resources</title>
    <para>
      If the statement in
      <olink targetptr="section:About This BookOnline Help" />
      was misleading, consider taking a training course to
      find out more about this application.
    </para>
  </section>

</chapter>
```

Note that there is now no space between the words "Book" and "Online" in the `targetptr` value as there is no whitespace between the phrases in the title. Any whitespace present between words or phrases, even new-line characters, is reduced to a single space character in the `targetptr` value, but no extra whitespace is inserted.

# Chapter 9

## Editing DocBook XML Documents

## 9.1 Introduction

DocBook XML defines a set of tags that can be used to structure a document and clearly identify its content. The document is stored in plain text format, so any text editor can be used to edit the content. However, there are other editing options and several issues to consider when editing. These are described below.

## 9.2 Choosing an Editor

### 9.2.1 Text Editors vs. XML Editors

You can use any text editor or XML editor to edit DocBook XML documents. Most document authors are familiar with text editors but maybe not with XML editors. Those who have experience writing HTML or XML documents with text editors may prefer to continue to use their favorite text editor. Many text editors also support syntax highlighting, automatic indentation, and other features that are XML-aware. These can help to make editing more productive.

XML editors are designed to recognize the structure of the XML document being edited and allow it to be manipulated more directly than with text editors. Many support the same plain text editing functionality of text editors, but most include a structured "tree view" and other interfaces. A few are even more specialized: they recognize DocBook XML as a special document format and support WYSIWYG document editing.

> **ⓘ WYSIWYG Editing**
>
> WYSIWYG stands for "What you see is what you get," a term used to describe editors that allow the user to edit a document on the computer screen in the same visual format that it will have when it

is printed. This is typical of most modern word processor applications. However, as DocBook XML does not contain formatting information, such an editor displays the document as it might appear by applying various formatting styles to the elements. The editor could be customized to apply styles that are like those that the document rendering system will use, but as the rendering system may apply different styles and produce many different output formats, the term WYSIWYG is probably better interpreted as, "It might look a little bit like this."

For DocBook XML editing, a WYSIWYG editor allows the author to get a quick impression of the structure and content of a document in a more natural representation, but there is a danger that it could also encourage too much focus on formatting. DocBook XML is not about formatting. There may be a tendency to use elements because of what they look like on the screen and not because they correctly identify the structure or content. For example, using `emphasis` for menu items because it is shown in an italic font instead of using `guimenuitem`, because the author does not like that the latter appears in a boldface font. You must remember that the appearance is that imposed by the editor, not the main rendering system. If the in-house style calls for menu items to appear in an italic font, then they will appear that way in the final output regardless of what style the WYSIWYG editor applied during editing. When faced with a concern over the formatting of a document, ask yourself this, "What would this look like in an audio book?" The answer should give you the correct perspective: do not concern yourself with formatting. If the structure and content are correct, you have done your job well; DocMaker is then responsible for all of the formatting.

Text editors have been around a long time and debate about which editor is "the best" has given rise to some infamous "religious" wars (the most famous—*vi* vs. *emacs*—has raged for nearly thirty years). If *Microsoft Notepad* is the first thing that springs to mind when you think of a text editor, then a text editor may not be for you. *Notepad* is poorly suited to editing DocBook XML documents because it lacks the features that make using a text editor for XML practical and productive. If you use a more advanced text editor, and know how to use it to the fullest, it may well be the ideal environment for you when editing DocBook XML documents.

XML editors are probably the best choice if you more familiar with WYSIWYG environments and may have limited experience editing XML documents. There is still a learning curve involved as you try to come to grips with the document structure, but the fairly recognizable interface and the familiar appearance of the content will help you to get started. With experience, you will realize that the main impediment to productive editing of structured documents is a reliance on the mouse; the sooner you learn how to move around a document and manipulate the content using the keyboard the better. Although the document may look like any you have seen in a word processor, you are restricted by DocBook XML's structure and content

rules when editing, so you will spend more time focusing on such aspects, and a lot more if you rely on the slow process of accessing menus and selecting options using the mouse.

## 9.2.2 Basic Requirements

Regardless of the editor you choose, there are a few basic requirements:

- The editor must allow you to choose the character encoding used when saving a document and allow the document to be edited using that character encoding. An editor that only supports ASCII text, i.e., text that includes only unaccented letters, numerals, and basic punctuation, is not sufficient.

- If the editor is an XML editor, it must not reformat the content of document elements that represent verbatim environments, i.e., elements, such as `programlisting`, that contain significant white-space characters.

Any editor that does not support these requirements cannot be used.

## 9.2.3 Selected Editors

Below is a selection of some editors that can be used to edit DocBook XML documents. Some are better than others, but most are being improved regularly and you should re-evaluate them occasionally.

XML Editors

**XMLmind XML Editor (XXE)**

XXE is a good, Java-based, XML editor that is targeted at DocBook XML and XHTML editing. Of those evaluated, it has by far the best WYSIWYG support for DocBook XML editing and is continually improving. Its features include table editing, context-sensitive element selection, customizable menus, a macro language, WYSIWYG and tree views, spell-checking, automatic conversion of characters not in the declared character set into character entities, attribute editing, image insertion, and many more. It also assists in the development of documents that are split across several files. It reformats content consistently and correctly, so you can use it to do the reformatting for you and then switch to a text editor when making minor changes. The standard edition is free of charge. Make sure you use at least version 2.7. While Cúram does not support any particular editor for DocBook XML, XXE is strongly recommended.

**Eclipse XML Editor**

The XML editor built into recent versions of *Eclipse* (and its derivatives) is quite capable. While it does not support WYSIWYG editing, it does provide context-sensitive tag completion, automatic formatting, and validation features that make it a useful tool. For authors who use this environment regularly, this editor may be convenient for many

tasks.

Text Editors

**Vim**

Standing for "vi-improved", *Vim* is a powerful text editor that supports syntax-highlighting, XML tag completion, tag formatting, folding, and a myriad of other general-purpose features that make it suitable for use as a DocBook XML editor. The learning curve is steep, but the graphical interface will come as a nice surprise to those who think of *vi* as a command-line editor. It comes with full support for Unicode 4.0 and all the popular character encodings, so it safe to use for XML editing.

**TextPad**

You can think of *TextPad* as, " *Vim* for people who can't remember keyboard short-cuts." It has a good range of features and, when configured properly, is a good choice for authors who may have experience using editors similar to the less capable *Notepad*.

## 9.3 Using an Editor

### 9.3.1 Overview

All the editors described above, and many others, are suitable for use when editing DocBook XML files. However, there are a number of issues that arise in their daily use that must be understood. Unless you are careful, you can make life considerably harder for yourself and other authors. Most of the issues relate to document encoding and formatting and how these affect the interactions with the source code management (SCM) system.

### 9.3.2 Encoding

The encoding of a document is critically important. The encoding defines the character set that the document uses; knowledge of the encoding allows other systems to correctly interpret the character data that makes up a DocBook XML file. As the file is in XML, the standard XML processing instruction at the top of the file can be used to declare the encoding used by the file. If you use an XML editor, it will recognize this encoding declaration and interpret the file in that manner. Similarly, XML editors can either save the data in the declared encoding, or can be configured to save all files using a specific encoding, changing the encoding declaration in the file to the appropriate value in the process. However, few text editors recognize the declared encoding of an XML file and will not necessarily respect that encoding when saving a file. It possible for an author to use a text editor to open a file that uses one encoding and then save it using an encoding different to the one declared in the file. The data in the file is now corrupt, though the degree of corruption may vary depending on the characters used. It is important, therefore, that the author is aware of the encoding that should be

used at all times.

If document files use many different encodings and text editors make managing the encoding a bit hit-and-miss (or tedious, at best), it makes sense to simplify the process. By using a single encoding for all files, the process becomes less error-prone. As all Cúram documents are written in English (US or GB variants), the Western European language encoding defined in the ISO-8859-1 ("ISO Latin 1") standard is a good compromise. This character set defines the encoding of nearly all of the characters (including the commonly used "ú") that you will need for Cúram documents. It is also one of the most widely used encodings and almost every text editor that can handle multiple encodings can handle this character set.

You might wonder why the equally well supported and more versatile UTF-8 encoding is not a better solution. The answer is simple: try typing an em-dash, right or left double-quotes, or any Chinese characters using your keyboard. Unless you have a non-English keyboard setup, or know the codes for these characters, you will not be able to do it. While the Chinese characters might not be very relevant to you, the other punctuation characters may be. Microsoft though them important enough that it embraced the ISO Latin 1 character set and extended it to include the left and right variants of the single and double quotation marks, the em-dash, the en-dash, the Euro symbol, the trademark symbol, the bullet mark, and several others. This character encoding is dubbed "Windows Latin 1", "Windows ANSI", "windows-1252", or "CP1252", though that list of names is not exhaustive. The problem is that this encoding does not have the same level of cross-platform support as ISO Latin 1. As it is difficult to type most of the extended characters on the keyboard, it is of little extra benefit in any event.

ISO Latin 1 is a subset of Windows Latin 1, so it is safe to allow it to be interpreted as Windows Latin 1, though care must be taken not to enter any of the extended characters when editing. DocBook XML documents should use the `quote` element instead of quotation marks, use the itemizedlist where bullet marks are needed, and use the product element when a trademark symbol is required, this leaves only the fairly common em-dash and Euro symbol, and the less common en-dash to be handled. The Euro symbol is supported by ISO-8859-15 ("ISO Latin 9"[1], a small extension to ISO Latin 1), but that is not widely supported and it still does not provide the em-dash or en-dash characters.

The solution to all this lies in the fact that a program that manipulates the XML will interpret the character data using the declared encoding, but process it internally using a full Unicode encoding. Therefore, you can use XML "character entities" when entering characters that are not in the ISO Latin 1 character set. For example, the Unicode code for the em-dash is 8212, so, if you enter `&#8212;` in your document, it will be interpreted as an em-dash and, as all the characters in the character entity itself are in ISO Latin 1, it is safe to use this method. As life is too short to learn what code to use for what character, DocBook XML defines named entities for many commonly used characters. Instead of entering `&#8212;`, you can use `&mdash;` instead. Similarly, you can also use `&ndash;` and `&euro;` and

many others. The codes used are not those defined by the Windows Latin 1 encoding, they are the Unicode (UTF-16) codes. You can even enter those Chinese characters in this manner. It is perfectly valid to write an XML document in Chinese that uses the ISO Latin 1 encoding as long as you use the character entities. The character data is all in the ISO Latin 1 encoding, but the information conveyed using that character set is actually in Chinese.

> ### XXE and Character Encodings
>
> One of the nicer features of *XXE* is the help it provides when entering characters that do not appear on your keyboard. It provides a list of Unicode characters and allows you to choose any and insert it into the document. If the character is defined in the ISO Latin 1 encoding, it will insert it literally. If the character is not defined by the encoding, it will insert a character entity instead. If the DocBook XML DTD defines a named character entity for that character code, *XXE* will automatically insert the entity name instead of the code.
>
> Unless you look at the actual XML file, you will be oblivious to all this; *XXE* displays the actual character glyph on the screen and not the character entity, so you can forget about character sets and encodings and just keep writing.

If ISO Latin 1 is accepted as the standard encoding for all documents, all the documents are in English, and few characters outside of this character set are required, the compromise is a good one: it is well supported, easy to enter at the keyboard, cross-platform, and is reasonably immune to mistakes that arise when it is confused with Windows Latin 1.

### 9.3.3 Formatting

One of the benefits of using a plain text format like DocBook XML, rather than a proprietary binary format, is the easy of integration with SCM systems. CVS is used to manage Cúram documents and its facilities for automatic merging can be utilized for DocBook XML files. Unlike the former *FrameMaker* documents, it is now possible to edit an single file from a book without locking all the other files and it is even practical for more than one person to edit the same file concurrently. The former is possible because there is no data about page numbers, paragraph numbering, cross-references, etc. in DocBook XML that causes dependencies between files in a book; all this information is calculated during rendering. The latter is now practical because CVS can automatically merge the changes from two authors by comparing the content of the files. Using change histories, it is also possible to select two versions of a file and compare them using a suitable tool to identify the changes that were made between those versions.

Most of this newly enabled functionality depends on the way CVS merges two versions of a file: it cannot merge two files if each contains changes made to the same lines of the original version (the "common ancestor"). Most of the time, these conflicts will be unavoidable and relatively easy to resolve. However, if an author's editor reformats the XML file (and most XML editors do this), it will likely change every line in the file and may

cause a merge conflict if the file is also being edited by someone else, or, more likely, will render the change history unusable. The author may only have edited a single line, but by using an editor that reformatted the file, CVS will consider every line changed.

Several ways to avoid this problem are:

- Use an editor that does not reformat the file (almost any text editor fits this bill, but few XML editors preserve the formatting).

- Standardize on an XML editor that can be configured to format the file in the same manner for all authors (you can still use a text editor for changes).

- Only support XML editors that can be configured to format files in an identical manner (no two editors allow such control).

- Reformat all the files to a standard format before submission using a separate tool.

For now, only the second option is supported. Bot text and XML editors are desirable, so any text editor can be used, *XXE* is the recommended XML editor, and the commonly used *Eclipse* XML editor can be used as it does not force you to reformat files. The last option is likely to be introduced at some point in the future and will include the added benefit that authors using text editors are not forced to deal with poorly formatted content created by other authors using text editors!

## 9.4  Summary

The choice and use of editors for DocBook XML documents can be summarized as follows:

- DocBook XML is an open, plain text format allowing many different editors to be used.

- To simplify the handling of documents, the character set should use the ISO Latin 1 (ISO-8859-1) encoding. This must be declared at the top of each file in the XML processing instruction and editor must be configured to use this encoding when loading and saving files. Only a few commonly used characters are not included in this character set; these can be used by inserting named character entities.

- Many XML editors have been evaluated for use. *XXE* is recommended for its comprehensive DocBook XML support and good WYSIWYG presentation. The *Eclipse* XML editor can be used as long as the document is not reformatted. No other XML editor is recommended; they are either inferior to these choices, or they cannot be used in the same environment as the others because of incompatible reformatting of documents.

- Any text editor that supports the ISO Latin 1 encoding can be used. Text

editors that support Windows Latin 1 (*Notepad* shows this as just "ANSI"), but not ISO Latin 1, can be used if only characters common to both character sets are used (i.e., characters in the range 128–159, including "smart" quotes, dashes and the euro symbol among others). These characters should be entered using character entities with either the Unicode character code or the equivalent DocBook XML name. You will find a description of these codes in Chapter 7, *Character Entities*.

## Notes

[1]Note that there is no such thing as "ISO Latin 15" (at the time of writing). The ISO-8859 series includes several non-Latin character sets, hence the numbers can be a little confusing. Even more confusing is that ISO-8859-1 and ISO 8859-1 (without the extra hyphen) are not the same. The former is actually an "Internet enabled" variation on the latter; it defines all the control codes. The latter is more correctly named ISO/IEC 8859-1.

# Appendix A

## DocMaker Tools

## A.1 Introduction

DocMaker has a number of tools that may be useful in other projects. In particular, it includes an Ant task for performing XSLT transformations and facilities for profiling the performance of stylesheets and Ant build scripts. These tools can be borrowed from DocMaker and reused in other projects.

## A.2 Transform Ant Task

### A.2.1 Features

DocMaker's Transform Ant task has a number of features that differentiate it from Ant's own Style/XSLT task. The Transform task was developed to solve some of the problems inherent in the use of Apache Xalan under Java 1.4 and later. The Java Runtime Environment (JRE) 1.4 and later includes an old version of Apache Xalan in its core and it can be difficult to use a newer version of Xalan instead. The Transform task ensures that any version of Xalan can be used and that the version included in the JRE will be ignored.

Some of the main features of the Transform task are:

- Special, optimized classloaders to make it simple to use any version of Apache Xalan regardless of the version of Xalan included in the JRE.

- All classpath configuration can be performed within the same Ant script, so no JVM options need to be set and no JRE installations need to be modified.

- XML inclusions are recognized and inclusions specified in the input XML file can be performed optionally.

- Dependency checking support also detects and checks the timestamps on XML files included by the main input XML file.

- Performance profiling of the XSLT stylesheets can be enabled and the results analyzed in a spreadsheet application.

- An option to compile the stylesheet and use the Apache Xalan XSLTC processor rather than the default interpretive processor. This may improve the speed of the transformation.

## A.2.2  Task Configuration

The Transform task forms an implicit Ant `fileset` for the selection of one or more input XML files to be processed. The base directory of the `file-set` is the same as the project's base directory unless overridden using the `basedir` attribute. The task can contain `include` and `exclude` child elements to select the input files. The task is similar to Ant's core Style/ XSLT task, but dispenses with some redundant features.

| Attribute | Description | Required |
|---|---|---|
| basedir | Specifies the base directory used to locate the input XML files. The default is the project's base directory. | No |
| destdir | The directory in which the output XML files will be placed. If not specified, the project's base directory will be used. If neither the `destdir` nor the `basedir` attributes are specified, then the output XML files will be created in the same directory as the input XML files and unless a different `extension` is specified, the transformation will fail. If the destination directory does not exist, it will be created. | No |
| in | The name of a single input file to be processed. The name will be resolved relative to the project's base directory if it represents a relative path. This must be used in combination with the `out` attribute. Extended dependency checking is not support when using `in` and `out`. | No, unless `out` is specified. |
| out | The name of a single output file to be created when the single input file is processed. The name will be resolved relative to the project's base directory if it represents a relative | No, unless `in` is specified. |

| Attribute | Description | Required |
|---|---|---|
| | path. The `destdir` attribute must not be used. If the directories in the path to the output file do not exist, they will be created. | |
| extension | The file extension to use for the output XML files. If not specified, the extension of the input file will be used. | No |
| style | The XSLT stylesheet file to be used. If the path is not absolute, it will be resolved relative to the project's base directory. | Yes |
| classpathref | A reference to a path-like structure (usually a `path` element) that defines the classpath required for Apache Xalan and Apache Xerces to override any version included in the JRE. The JAXP/TRAX API (`javax.xml.*`, etc. classes) need not be included if these classes are present in the JRE, as these will not be overridden. If stylesheets use extension functions or elements, the appropriate classes should also be included in this classpath. If not specified, the system classpath will be used. | No |
| xmlcatalogref | A reference to the XML catalog used to resolve references to a document type definition (DTD) from the input XML files. See the documentation on the Ant `XMLCatalog` type to learn how to configure an XML catalog. | No |
| dependdir | The base directory to use when locating dependency files used for dependency checking of XML inclusions. For each input XML file relative to the `basedir`, there may be a dependency file in the same relative location below the `dependdir` that contains the dependency information. | No |
| dependextension | The extension used for the dependency files in the `dependdir`. The | No |

| Attribute | Description | Required |
|---|---|---|
| | file names should be the same as the names of the input XML files. By default, the extension of the input XML file will be assumed. | |
| xinclude | Indicates if XML inclusions in the input XML files should be processed. This is `false` by default. Set to `true` if XML inclusions are required. | No |
| force | Forces the processing of the input XML files and disables all dependency checking. This is `false` by default. | No |
| profile | Enables performance profiling of the stylesheet by specifying the name of a CSV (comma-separated values) file that should be created with the profile data after the processing is complete. If not specified, no profiling will be performed. This is `false` by default. Set to `true` to enable profiling. Note that profiling will not work if the XSLTC processor is enabled using the `compile` attribute. | No |
| compile | Enables the use of the compiling processor XSLTC instead of the Xalan interpretive processor. The XSLTC processor must be available on the classpath or this option will be ignored. This is `false` by default. Set to `true` if compiled stylesheets should be enabled. If enabled, stylesheet profiling will not be possible, even if XSLTC is not found on the classpath and the interpretive Xalan processor is used. | No |
| locationinfo | Enables the creation of node location information when source files are parsed. This is `false` by default. Set to `true` if node location information should be enabled. If enabled, stylesheet may use Xalan extension functions to access the system ID, line number, and column number of a node. This requires considerable extra resources to support, so only | No |

| Attribute | Description | Required |
|---|---|---|
| | enable it when necessary. | |
| scanincludeddir-ectories | If any child `include` elements match directories, indicates if all files in these directories should be processed. This is `true` by default. | No |

Table A.1 Transform Task Attributes

As well as nested `include` and `exclude` elements, the Transform task also supports nested `param` and `outputproperty` elements in the same way as Ant's Style/XSLT task. See the documentation of that task for more details. Other nested elements of the Style/XSLT task are not supported.

The following example illustrates how the Transform task is defined.

```
<path id="cp.task">
  <fileset dir="${docmaker.lib.dir}">
    <include name="jde-commons-ext.jar" />
    <include name="docmaker.jar" />
  </fileset>
</path>

<taskdef name="transform" classpathref="cp.task"
  classname="curam.docmaker.tools.anttasks.Transform" />
```

Example A.1 Defining the Transform Task

The classpath includes two JAR files located in the `lib` directory of the DocMaker installation. The JAXP and TRAX APIs must also be provided, but these will probably be accessible via the classpath of Ant or the JRE, so they rarely need to be specified explicitly. These APIs are usually defined in the `xml-apis.jar` file that is supplied with Apache Xalan, Xerces or Ant, or in the core JRE.

The next example shows how an XML catalog and the Xalan classpath can be configured for use with the task.

```
<path id="cp.transform">
  <fileset dir="${xalan.dir}">
    <include name="xalan.jar" />
    <include name="xercesImpl.jar" />
  </fileset>
</path>

<xmlcatalog id="cat.docbook">
  <dtd publicId="-//OASIS//DTD DocBook XML V4.2//EN"
      location="file:///${dtd.dir}/docbookx.dtd"/>
</xmlcatalog>
```

Example A.2 Configuring the Transform Task

The classpath should include the JAR files necessary to run the version of Xalan that you require. A compatible version of Xerces should also be included. Some versions of Xalan may require additional JAR files; the above will work for Xalan 2.6.0, but Xalan 2.7.0 also requires the serial-

`izer.jar` file that is shipped with it. If there are multiple instances of the Transform task in an Ant script, and each uses the same classpath reference, they will all share a single classloader to access Xalan classes. This has significant performance benefits. If you use different classpath references for each instance of the task, the classloaders will not be shared. Therefore, where possible, try to define a single classpath that will be used by all instances of the Transform task.

The XML catalog defines the location of the DTD file corresponding to the public ID of DocBook XML 4.2. Note that a bug in Ant 1.6 means that the "file:///" prefix is required before the location of the DTD file. The XML catalog ensures that a local copy of the DTD will be used for validation if an input XML file defines a `DOCTYPE` with the same public ID. If an XML catalog were not used, a copy of the DTD would be downloaded from the location specified by the system ID of the `DOCTYPE` (probably somewhere on the Internet) and download could impact performance adversely.

The transform task can be used as shown below.

```
<transform destdir="${html.dir}" style="my.xsl"
           extension=".html" basedir="${in.dir}"
           xinclude="true" classpathref="cp.transform"
           xmlcatalogref="cat.docbook">
  <include name="**/book.xml" />
</transform>
```

Example A.3 Using the Transform Task

This will use the stylesheet `my.xsl` to transform all `book.xml` files located below the directory indicated by the `in.dir` property into `book.html` files in corresponding directories below the directory indicated by the `html.dir` property. If the input XML files include other XML files, the inclusions will be resolved before the transformation is performed.

## A.2.3  Dependency Checking

Dependency checking is based on the modification time-stamps of the files and operates according to the following rules:

- If the `force` attribute is set to `true`, no dependency checking is performed and all input XML files are processed.

- If the XSLT stylesheet file is newer than the output XML file, the corresponding input XML file is processed again.

- If the input XML file is newer than the output XML file, it is processed again.

- If any of the files listed in an input XML file's corresponding dependency file are newer than the output XML file, the input XML file is processed again.

The dependency file is a simple file that lists the names of files that the input XML file depends on. Each input XML file can have a corresponding

dependency file that is resolved using the `dependextension` (if any) and the same relative path to the `dependdir` (if any) as the input XML file has to the `basedir`. The dependency file lists the dependent files one per line using paths that are relative to the corresponding input XML file. DocMaker includes an XSLT stylesheet than can transform an XML file into a dependency file by identifying the XML inclusions in the input file. However, if included XML files have further XML inclusions, they are not detected. If XML inclusions are not used, the simple dependency checking rules will probably suffice, but the following example demonstrates how to use the extended dependency checking.

```
<target name="dependencies">
  <transform destdir="${dep.dir}" style="depend.xsl"
             extension=".d">
    <outputproperty name="method" value="text"/>
    <include name="**/*.xml"/>
  </transform>
</target>

<target name="transform" depends="dependencies">
  <transform destdir="${out.dir}" style="my.xsl"
             dependdir="${dep.dir}" dependextension=".d"
             xinclude="true">
    <include name="**/*.xml"/>
  </transform>
</target>
```

Example A.4 Dependency Checking of XML Inclusions

The `dependencies` target does not enable XML inclusions and uses an XSLT stylesheet to transform input XML files into dependency files with the extension ".d" located in another directory. The `depend.xsl` stylesheet can be copied from DocMaker's `lib` directory; it will create a text file that lists the values of the `href` attributes of any `xi:include` elements it finds in the input file. You are free to generate dependency files using any other method, but this is the way that DocMaker does it.

The `transform` target depends on the `dependencies` target. It enables XML inclusions and processes the same input XML files using a different stylesheet. It indicates the location of the dependency files to ensure that, even if an input XML file has not changed, a change to a file that it includes will be detected and trigger the processing again. When editing DocBook XML content, this is useful if only a chapter file is edited, as the change will be detected via the inclusion in the book file and the book will be rendered again.

## A.2.4 Performance Profiling

Profiling the execution of XSLT stylesheets can be very useful if performance is a problem. The Transform task can generate a profile in a file in CSV (comma-separated values) format that shows the number of invocations of each template in the stylesheet, the total time spent executing each template and other statistics. This file can be opened in any common spreadsheet application to sort the data and analyze the problem areas. Enabling

profiling is simple and is shown below.

```
<path id="cp.transform">
  <fileset dir="${docmaker.lib.dir}">
    <include name="jde-commons-ext.jar" />
    <include name="profile.jar" />
  </fileset>
  <fileset dir="${xalan.dir}">
    <include name="xalan.jar" />
    <include name="xercesImpl.jar" />
  </fileset>
</path>

<target name="transform-html">
  <transform destdir="${html.dir}" style="my.xsl"
             extension=".html" basedir="${in.dir}"
             xinclude="true" classpathref="cp.transform"
             xmlcatalogref="cat.docbook"
             profile="html-profile.csv">
    <include name="**/book.xml" />
  </transform>
</target>
```

Example A.5 Profiling XSLT Stylesheets

Profiling requires the addition of two JAR file to the Transform task's classpath (the classpath used for transformations, not for the task definition). Both JAR files can be found in the lib directory of the DocMaker installation. To compensate for a bug in Apache Xalan, the profile.jar file should come before the xalan.jar file in the definition of the classpath as shown above.

Profiling is enabled by adding the profile attribute to the transform task and setting it to the location of the CSV file that should be created. The profile will include data gathered from the transformations of every input XML file and will be saved when the task has completed. The extension ".csv" is convenient, as you are likely to have a spreadsheet application installed that is associated with this extension.

## A.3  Ant Script Profiling

The same type of CSV profiles that are generated by the Transform task to profile XSLT stylesheets can also be generated to profile the execution of entire Ant scripts. An Ant build listener class is included with DocMaker that will generate a profile in a CSV file if the listener is defined when invoking Ant. Two JAR files, profile-ant.jar and jde-commons-ext.jar, are required when profiling Ant scripts. They should be on the system classpath supplied to Ant. This can be achieved in two ways: copy these files to the lib directory of the Ant installation, or include then in the CLASSPATH environment variable. The latter method is less invasive. To enable the profiling, use Ant's -listener command-line option to include curam.docmaker.tools.profile.ProfilingAntListener as a build listener. After the Ant build has completed, the profile will be written to a file named ant-profile.csv.

## A.4  Accurate Profiling

The accuracy of the profile data depends on the accuracy of the timers used. The profiling framework in the JDE Commons (used by DocMaker) includes timers called *stop-watches*. There are a number of different stop-watch implementations that use different timing mechanisms depending the version and vendor of the JRE used; the most accurate one that can be supported by the JRE will be selected automatically at run-time. Regardless of the accuracy of the stop-watch used, all of the profiles report times in nanoseconds.

For accurate profiling, a JRE 1.5 (or later) should be used, but profiling under the Sun Microsystems 1.4.2 (or later) JRE is also quite accurate. Both of these JREs provide sub-microsecond accuracy and can accurately measure the elapsed time of very fast operations, such as simple templates in a stylesheet. Timing facilities in earlier JREs or in the IBM 1.4.2 JRE are not as accurate and have only millisecond accuracy or worse. This is not accurate enough for profiling XSLT stylesheets, but may suffice when profiling the more lengthy tasks of an Ant script.

The `jde-commons-ext.jar` file supplied with DocMaker is a special build of the JDE Commons that includes all of the stop-watch implementations, ensuring that the best possible implementation will be selected for you automatically. While the `jde-commons.jar` file supplied with the Cúram JDE could be used when profiling, it may have been created with a JRE that could not compile all of the stop-watches and may not be able to support accurate profiling under *any* JRE.

# Appendix B

## Converting FrameMaker Documents

## B.1 Introduction

Originally, Cúram documentation was written using *Adobe FrameMaker*. DocMaker provides a number of facilities to aid the conversion of documents in *FrameMaker* format to the DocBook XML format. While much of the conversion process is automatic, manual intervention is necessary and the quality of the output depends on a number of factors that will be described in detail. Be comforted by the fact that you only have to do this once!

## B.2 Prerequisites

Before converting documents, you must have both DocMaker and *FrameMaker* installed on your computer. *FrameMaker* must be configured to use the "Structured FrameMaker" interface. This can be selected at installation time, or, if already installed, by selecting *File→Preferences→General...* and changing the *Product Interface* to "Structured FrameMaker" before pressing the *Set* button. *FrameMaker* will need to be restarted for this change to take effect.

## B.3 The Conversion Process

Cúram documents are organized as books with separate files for the front matter, table of contents, and the main text. The text may be organized with one or more chapters or appendices per file. Books organized in a different manner cannot be converted.

Only the main text is converted, the front matter is not processed and is discarded. Most of the discarded text is just boilerplate or generated information that DocMaker will create afresh. Information such as non-standard legal notices and the printing history must be recreated by hand after the con-

version process.

The conversion process is performed in several phases:

1. Each file in the book must first be checked and prepared for conversion. If a document is in good order, this process is relatively quick and painless. However, documents that do not use the Cúram template correctly may take some time to prepare.

2. The conversion facilities provided by *FrameMaker* are used to add structure to the documents. A *FrameMaker* document called a *conversion table* is used to identify known sequences of paragraph formats and map them to DocBook-like[1] elements. If the paragraph formats were used in unconventional ways, there will be problems identifying the documents' structure.

3. A simple *FrameMaker structure application* is used to control the creation of XML files from the structured documents. The structure application defines the export rules for any images included in the document.

4. A temporary `book.xml` file is created to group and order the XML files created by *FrameMaker*. As *FrameMaker* cannot provide a meaningful conversion of its book file, this file is created by DocMaker and then adjusted by hand as needed.

5. DocMaker takes the XML files produced by *FrameMaker*, and the book file that groups them, and converts it to DocBook XML format. This conversion process performs some integrity checks, resolves cross-references, and restructures the *FrameMaker* output into a valid DocBook document.

6. Once the DocBook is produced, you must perform some manual steps to finish the conversion process. This includes the renaming of image files and the completion of the main `book.xml` file.

## B.4  Conversion Issues

### B.4.1  Introduction

Before converting *FrameMaker* documents to DocBook format, they must be checked, and probably edited, to ensure the success of the conversion process. If the documents are well prepared, the conversion process will be smoother and the DocBook output will require less modification.

### B.4.2  The Cúram Template and Formats

Only documents that use the standard Cúram *FrameMaker* template can be converted. The conversion process relies on the identification of the standard paragraph and character formats defined in the template to determine the

structure and meaning of the text in the document. If these formats are used incorrectly, ambiguously, or in ways that deviate from their prescribed usage, the conversion process may fail.

Manual overrides of the predefined formats will not be preserved. For example, if the character format "Emphasis" is used, it will be converted to the DocBook `emphasis` element, but if an override such as the direct application of an italic font was used, it will be lost. There is no way the conversion process can determine why italics was used; it may well not have been used for emphasis, so such an assumption would be invalid.

All manual overrides will be flagged in the output DocBook document. They can be identified within a *FrameMaker* by placing the cursor in the paragraph or character sequence being checked and looking at the format name in the status bar in the lower left of the *FrameMaker* window. The paragraph format name and, if applicable, the character format name will be displayed. If either format name has an asterisk character appended, this indicates a manual override. On some occasions (typically when a character format is applied to all of the text in a paragraph), it may be impossible to eliminate the flagging of a manual override on a paragraph format. These can be ignored.

To remove manual overrides on character sequences, select the text and set the character format to "Default ¶ Font" to clear the overrides before applying the correct character formats.

### B.4.3  Cross-References

The conversion process attempts to identify and preserve all cross-references in the document, however, there are some limitations.

Chapter cross-references may be broken due to an anomaly in the use of the Cúram template whereby a cross-reference marker is created on a *master* page instead of a *body* page. These will need to be fixed manually before conversion.

Only some of the standard cross-reference formats (and even less of the non-standard ones) are supported. One problem is the use of cross-references that target another file. Many can be resolved automatically, but there are restrictions. The table below describes the supported cross-reference formats and when they should be used. It is likely that most of the cross-references in your document are compliant, but some may need to be changed. Some formats have several names (some are just misspellings); the supported alternatives are listed.

| Format Name | Description |
| --- | --- |
| Chapter Reference | A reference to a "ChapterNumber" paragraph. It should be the only format used for references to chapters. Any type of reference to a "ChapterTitle" paragraph is not supported. |

| Format Name | Description |
|---|---|
| | The alternative names "ChapterReference", "Chapter Num", and "Chapter Ref" are supported. |
| Chapter & Section Reference | A reference to a "Section" or "Subsection" paragraph within a chapter. This is the preferred format for use when sections are located in a different chapter or file. |
| | No alternatives are supported. |
| Appendix Reference | A reference to an "Appendix Number" paragraph. It should be the only format used for references to other appendices. |
| | The alternative "Appendix Ref" is supported. |
| Appendix & Section Reference | A reference to a "Section" or "Subsection" paragraph within an appendix. This format was not included in the standard template, however, for a cross-reference to a section in an appendix from a location outside that appendix, this format should be applied or the cross-reference may not be resolved. You can import it from the conversion template described below. An ordinary section reference may work, but the probability is lower. |
| | The alternative "Appendix Section Reference" is supported. |
| Section Reference | A reference to a "Section" or "Subsection" paragraph within the same chapter or appendix. If this is used to reference a section outside of the current chapter or appendix, it is less likely to be resolved correctly. |
| | Many variations on the format name are acceptable: "Heading & Section", "Section & Heading", "SectionAndTitleReference", "Subsection Reference", and "SubSection Reference". |
| Subsubection Reference | The "Subsubsection Reference" format is usually defined differently from the "Section Reference", as this section level is not numbered, however, it will be treated like a standard section reference during conversion and is acceptable once the reference is within the same file. Again, when the reference is to another file, the chance of it working |

| Format Name | Description |
|---|---|
| | becomes slimmer. |
| | Long, made-up words seem to be hard to spell, so the following variations have been identified and are acceptable: "Subsubection Reference" and "Subsubsection". |
| Table Reference | A reference to a "TableTitle" paragraph within the same chapter or appendix. References to tables, even in the same chapter or appendix, cannot always be resolved due to limitations in the conversion facilities provided by *FrameMaker*. |
| | No alternatives are supported. |
| Figure Reference | A reference to a "FigureTitle" paragraph within the same chapter or appendix. References to figures in other chapters or appendices may not be resolved. |
| | No alternatives are supported. |
| Listing Reference | A reference to a "ListingStart" paragraph within the same chapter or appendix. References to listings in other chapters or appendices may not be resolved. |
| | No alternatives are supported. |

Table B.1 Cross-Reference Formats

None of the cross-reference formats that include page numbers are supported. These should be located and changed to one of the supported formats.

References within the same file are always resolved accurately for all but tables. References to other files (and local tables) are resolved by matching the title text stored in the reference to the title text of possible targets. If there are several targets with the same title text (e.g., "Introduction", etc.), the references cannot be resolved. In this case, a warning is issued and the reference will have to be fixed by hand later. The special formats "Chapter & Section Reference" and "Appendix & Section Reference" improve the odds by narrowing the search for titles to the target chapter or appendix respectively.

When a cross-reference cannot be resolved due to ambiguities, DocMaker will replace the reference with a marker that can be easily identified. The format of an unresolved cross-reference marker is `__dm_xref_type__target@@@`, where `type` is the type of reference, for example, `table`, or `chapter_section`, and `target` is some information about the target such as the title text or paragraph number. Simply find these markers as fix them when the conversion process is complete.

### B.4.4  Vocabulary Limitations

The template used for Cúram documents does not have the rich *vocabulary* of formats (or tags) that are available in DocBook. The resulting DocBook output will, therefore, need to be edited to "upgrade" the document to use the new vocabulary.

The "GUIElement" character format is used for all buttons, menus, labels, etc., but DocBook distinguishes between these objects. The conversion process applies the `interface` element to all of them, but you should then modify your document and apply the `guibutton`, `guimenu`, `guilabel`, etc. elements. For menus, this can include the use of the `menuchoice` element that allows the effective definition of complex menu choices that include menus, sub-menus, and menu items.

The "CodeElement" character format is also widely used for many different objects such as class names, literal text, command names, source code fragments, etc. These are all marked as `literal` elements in the converted output, but, again, `classname`, `command`, `sgmltag`, etc. should be used instead and must be applied manually.

### B.4.5  Other Issues

*FrameMaker* documents include a dictionary of words that are allowed in the document. When converted, these words are lost. The DocBook document will need to be spell-checked and the allowable words added to the wordlist. As it is possible for the document dictionary to contain erroneous words that were added by accident, this presents an opportunity to review the spelling within all the documents.

Quotation marks are used directly in *FrameMaker* documents, but the `quote` element is used in DocBook instead. As an author types a quotation mark in *FrameMaker*, it is converted to a "smart quote" by the application. These marks can cause problems when converting to different encodings and are very difficult to maintain when editing a DocBook document in a basic editor. The conversion process, however, cannot replace these quotation marks with the `quote` element with any confidence, so the exercise should be performed manually after conversion. Where the double quotation marks appear within source code listings, they should be "straight double quotes"[2] to comply with the syntax of that language and should not be marked with the `quote` element. If the template changes the style of the quotation mark used when rendering the `quote` element, you do not want it to affect any source code examples.

Character, such as the "smart" quotation marks mentioned above, that are not within the ISO "Latin 1" character set, appear as numeric character entities. For example, an em-dash appears as `&#8212;`. These can be difficult to decipher, so you can convert these into named character entities by editing the file using a suitable XML editor such as *XXE* and then saving it. In the case of the em-dash, it will now appear as `&mdash;`. This procedure is

not mandatory, and while it makes the file more readable, the biggest culprits are the "smart" quotes that should be replaced with a `quote` element instead of character entities.

DocBook has no inherent concept of formatting beyond the expectation that whitespace in the content of certain elements (such as `programlisting` elements) should be preserved during rendering. This means that there is no way to specify an indented paragraph, something commonly used in Cúram documents by applying the "Indent" paragraph format. DocMaker will indent paragraphs as appropriate when rendering DocBook documents, but there is no one element in DocBook that can be applied to all indented paragraphs without the potential for identifying content incorrectly. For example, paragraphs within the same `listitem` element are indented to the same level by DocMaker, something that has to be manually arranged using different formats in *FrameMaker*. However, isolated indented paragraphs may be intended as short code samples, ad hoc lists, user input, etc. and these require the application of very different DocBook elements. These can only be resolved manually.

## B.5  Preparing a Book

💡 FrameMaker's Find Tool

*FrameMaker* allows more than just text to be located with its find tool. You can also search for cross-references by format name, any cross-reference regardless of format, unresolved cross-references, paragraph and character tags, anchored frames (figures), character formatting, etc. This can be very useful when preparing a document. The preparation is the most painful part of this process, so learning this shortcut can save you many hours of drudgery!

You can, for example, find cross-references and then change the format of those that refer to sections in other files. Also, you can search for characters with a particular format, say in "Times New Roman Italic" font and check that the "Emphasis" character format was used.

You can find out more about these features in the online help. See the "Searching for any item" topic in the "Word Processing" section accessible from the contents page.

The first step in the conversion process is preparation. Your *FrameMaker* documents need to be modified and corrected before the conversion process will work. Follow these steps:

1. For each file in the book containing a chapter or appendix, fix any erroneous chapter cross-reference markers by following these steps:

    a. Open the file in *FrameMaker*.

    b. Select *View→Text Symbols* from the menu to ensure that you can see the markers used for cross-references. (If this option is already

enabled, you can skip this step.)

    c.    Select *View→Master Pages* from the menu to access the master page used by the document.

    d.    If the master page contains a paragraph using the "ChapterNumber" format, or the "Appendix Number" format if this is an appendix, look for a symbol that looks a "T" at the end of the paragraph. If one is present, select it and cut it to the clipboard. This can be trickier than it sounds if you try to use the mouse. The easiest way to do this is to place the cursor at the end of the paragraph using the *End* key, hold down the *Shift* key and press the *Left Arrow* key once to select the marker, then press *Ctrl-X* to cut it.

    e.    Select *View→Body Pages* from the menu to return to the document text. Paste the cross-reference marker back at the end of the "ChapterNumber" or "Appendix Number" paragraph (but only if one is not already there). Doing this will prevent unresolved cross-references, but if there are two markers, they are unavoidable. Save the file and close it.

When you have completed these modifications, update the book and fix any broken chapter cross-references that are reported.

You should also ensure that only the supported cross-reference formats are used in your document. You can import the supported formats from the conversion template described below. If you want to commit your changes back to CVS before beginning the "destructive" phase of the conversion process, however, be sure to import only the cross-reference formats, and not any of the other formatting components (just turn-off most of the check-boxes in the dialog box).

2.    Remove all blank lines (other than in sample program listings) and page breaks from the text. These can be found more easily if text symbols are made visible. The use of blank lines for vertical spacing is redundant as such spacing is achieved by correctly using the paragraph formats. If it looks wrong, do not worry, DocBook is not about the *look*, it is about the *content*; formatting appropriate to the content will be applied by DocMaker during rendering and not by the author. Unless the blank line is meaningful content (as it may be in a program listing), it should be removed from the document.

3.    Reformat sample program listings so that only space characters are used for indentation, not tab characters. It may also be necessary to shorten lines in program listings, but this can be left until later when DocMaker performs its rule-checking.

It is also likely that program listings are not using the correct formatting if the code was copied and pasted from another application. Syntax-highlighting is a typical symptom. Check that all lines in the listing use the "ListingBody" paragraph format.

4. Many formatting overrides (changes made to text after applying the template's paragraph and character formats) can be fixed by re-applying the formats *en masse* for a particular file as follows:

   a. With the document file open, select *File→Import→Formats...* from the main menu.

   b. In the dialog box, select *Current* from the list of documents in the *Import from Document* list.

   c. Check *all* the check-boxes in both the *Import and Update* and *While Updating, Remove* sections and press the *Import* button. *FrameMaker* will now re-apply the formats in the document's own template to the tagged paragraphs and characters.

5. Check that the appropriate paragraph formats have been applied to the paragraphs in the document. For example, indented paragraphs should have used the "Indent" or "IndentDouble" format, and not the "Body" format with a manually-applied indentation.

6. Check that character formats are correct. Text that appears in any font other than the default paragraph font for the paragraph that contains it, should use one of the defined character formats and not use a manually-applied format override.

7. Frequently, figures are not correctly inserted into Cúram documents. Figures require the application of two paragraph formats to sequential paragraphs. The first paragraph should use the "Figure" format and contain an *anchored frame* (with text symbols visible, you should see an upside-down "T" symbol indicating the presence of the *anchor*). The second paragraph should contain the text of the figure title and use the "FigureTitle" format.

   To correct any deviation from this prescribed format, follow these steps:

   a. If the anchor for the anchored frame is not in a paragraph of its own (it may be at the end of the previous paragraph), relocate it to an empty "Figure" paragraph immediately before the "FigureTitle" paragraph.

   b. If the paragraph above the "FigureTitle" paragraph does not use the "Figure" format, change it. (Sometimes, it can be difficult to place the cursor within this paragraph due to the presence of the anchored frame. The easiest way to do it is to place the cursor on the "FigureTitle" paragraph and press the *Up Arrow* key once.)

   c. If the anchored frame was not configured correctly, the figure may be now be located in a seemingly arbitrary position on the page. You can correct this (though it is not required by the conversion process) if it makes it difficult to check the formatting of nearby paragraphs or figures: right-click on the anchored frame (you will

need to do this on, or very near, the frame boundary to avoid selecting the enclosed figure) and select *Anchored Frame...* from the pop-up menu. In the dialog box, ensure that the *Anchoring Position* is set to "At Insertion Point" and that the *Distance above Baseline* is set to "0". Click the *Edit Frame* button to confirm any changes.

Before completing the final preparation stages, you may want to save your document files and commit any changes to the SCM system. Up to this point, your changes have improved the formatting of the document, after this point the changes may be destructive and it is unlikely that you will want to preserve them.

*FrameMaker* can get confused about cross-references during the conversion process because the `.book` files store the full path to the `.fm` files. This means that when a book is structured and a new book file is created, the cross-reference that refer to other files still refer to the old, unstructured file in the original location. However, if the referenced file does not exist, then *FrameMaker* looks in the same folder as the `.book` file for the `.fm` file and correctly resolves the reference. Therefore, the conversion process requires that the original files be moved around to "trick" *FrameMaker* into correctly handling cross-references. This explains why you will begin to feel like you are playing musical chairs with the folder structure!

If you are lucky (or unusually diligent), you will have found and corrected all of the manual overrides in the document. The next steps will remove further overrides and prepare the document for conversion by changing the document template. Any information about the document content conveyed by overrides to the defined paragraph and character formats, will be lost. For example, if the importance of some content was implied by overriding the font of a "Body" paragraph to italic, this override will be discarded. Character overrides independent of a text set to a character format, will still be preserved for now.

1.  Close all documents in *FrameMaker*.

2.  You must now create a working folder to hold your original book's files. Using your file manager, go to the folder containing your book's files and create a new sub-folder named `fminput`. *Move* (do not copy) all of your book's files (the `.book` file and all the `.fm` files) to this folder.

3.  In *FrameMaker*, open the file named `fm2fmdb_template.fm` located in the `lib` folder of your DocMaker installation.

4.  Open the `.book` file for your document (now in the `fminput` folder).

5.  In the book window, select all of the files in your book that contain chapters or appendices (these are usually all of the files after the generated table of contents file).

6.  Select *File→Import→Formats...* from the main menu.

7. In the dialog box, select `fm2fmdb_template.fm` from the list of files in the *Import from Document* list.

8. Check *all* the check-boxes in both the *Import and Update* and *While Updating, Remove* sections and press the *Import* button. *FrameMaker* will now apply the "Cúram DocBook Conversion Template" template to all of the files.

9. With the same files selected in the book, now select *File→Import→Element Definitions...* from the main menu.

10. In the dialog box, again select `fm2fmdb_template.fm` from the list of files in the *Import from Document* list and press the *Import* button. Element definitions will be imported. You can dismiss the log window.

11. If you had any document files open while the formats were applied, they will need to be saved. You can go to the book window, right-click on the selected book files, and choose *Save* from the pop-up menu. This will save the changes made to these files that you had open when you imported the new template. Files that were not open will have the changes written to them automatically.

12. With the book as the current document, from the main menu select *Edit→Update Book...* and click the *Update* button in the dialog box. When the update has completed, save the book.

The preparation of your document is now complete. It may not yet be perfect, but the conversion process will detect any further problems and you can make the necessary changes to your document as needed.

## B.6 Structuring a Book

The next step is to convert the files in your book into *structured FrameMaker* documents.

A *conversion table* document supplied with DocMaker instructs *FrameMaker* to automatically identify typical sequences of paragraph and character formats and mark them up as document elements. For example, a "BulletedList" paragraph optionally followed by some "Indent" paragraphs are identified as DocBook `para` elements within a single `listitem` element. Consecutive occurrences of this pattern are identified as an `itemizedlist` element. Because the conversion process depends on the identification of these patterns, it is vital that the standard patterns are respected when preparing the document.

Structuring the document can be an iterative process: if the structure is not identified correctly, you will have to fix your document and try again. Here are the steps that you must follow:

1. Using your file manager, go to the folder containing the `fminput` folder and create a new sub-folder named `fmoutput`. This will be

used to contain the newly created structured *FrameMaker* document files.

2.  In *FrameMaker*, close all open documents and then open the file named `fm2fmdb_ct.fm` located in the `lib` folder of your Doc-Maker installation folder.

3.  Open the main `.book` file for your document (from the `fminput` folder) and select *File→Utilities→Structure Current Book...* from the main menu.

4.  In the dialog box, select `fm2fmdb_ct.fm` from the *Conversion Table Document* list. Set the *Output Directory* field to the `fmoutput` folder you created in the first step and check the *Allow Existing Files to Be Overwritten* check-box. Press the *Add Structure* button to create the newly structured document files. Each file in the book will be automatically opened, structured, and then saved. The newly structured book file will be opened for you.

5.  To check that the structuring process was successful, you must open each structured document file that was created in the `fmoutput` folder and examine it using the *Structure View* (you can access them via the book file that should be open). Some of the visual formatting of the content will be lost, but this is not important, only the structure is important now. Only the files containing chapters and appendices need to be checked. Backup files can be ignored.

    Open each file and click on the *Structure View* button above the right-hand scroll bar. If the root element of the structure view is `NoName`, the structuring has failed. Locate the point of failure in the document and determine the cause. It is most likely because the paragraph formats were used incorrectly. If you need to fix the formatting, open the *original* file, make your changes, save it, and then start the structuring process again.

6.  Apart from the `NoName` root element problem, there may be other, less critical, structural errors. You must locate these and fix them in the original document before running the conversion process again. Here are some of the errors:

    *   The structure view will highlight elements (created from paragraph and character formats) that contain manual formatting overrides by adding an `override` attribute with the value `yes`. If you have difficulty removing an override, select the text, reset the format to the default, and then apply the correct format.

    *   Text that was formatted without applying any character format will be contained in an `ILLEGAL_FORMATTING` element. In the original document, either remove the formatting, or apply the appropriate character format.

    *   Cross-references that were inserted using a cross-reference format

not in the standard template will be contained in an `ILLEG-AL_CROSS_REF_FORMAT` element. You should use one of the standard formats instead.

- Some standard cross-reference formats are not supported. These are indicated by an `UNSUPPORTED_CROSS_REF_FORMAT` element. Change these cross-references to use one of the supported formats.

- Paragraph, character and table formats that are not in the standard template will be marked by an `ILLEGAL_PARAGRAPH_FORMAT`, `ILLEGAL_TABLE_FORMAT`, or `ILLEG-AL_CHARACTER_FORMAT` element respectively. Reformat these using one of the standard formats.

- *FrameMaker* uses various types of *markers* to support cross-reference targets, hypertext links, conditional text, etc. Only the "Cross-Ref", "TopicAlias", "ImageAltText", "ImageLongDescText", and "TableSummary" markers are supported by DocMaker. All other markers are converted to an `UNSUP-PORTED_MARKER_TYPE` element. These do not have to be fixed, but warnings will be generated during later processing.

You can use the find tool in *FrameMaker* to locate many of these problems by searching for elements by name. The element name need only be a part of the full element name, so searching for "ILLEGAL" will find all of the elements whose names contain that phrase.

7. With the structured `.book` file open, right-click on each file containing a chapter or appendix, select *Numbering...* from the pop-up menu and set the numbering as follows:

- The file containing the first chapter should be set to restart all chapter and paragraph numbering from "1" and use a numeric format.

- Subsequent chapter files should be set to continue all chapter and paragraph numbering from the previous file. You can set all these files simultaneously by selecting them all in the book's file list before opening the numbering dialog box.

- The file containing the first appendix should be set, like the first chapter, to restart chapter and paragraph numbering from "1" and use a numeric format. Note that appendices normally use an alphabetic format, but this must be changed to numeric to allow cross-reference resolution.

- Subsequent appendix files should be set the same way as the subsequent chapter files. Again, a numeric, not alphabetic, format must be used.

- The page numbering is not important and can be left as is.

The exact settings are dependent on the way your book is structured, so

more details cannot be provided here. The numbering is important later when DocMaker must resolve cross-references made across file boundaries.

8. Now, to prevent *FrameMaker* from messing up the cross-references, rename the `fminput` folder (the folder containing the working copy of your unstructured book) to `fminput.hide`. This will prevent cross-references in the new structured documents from referring to the unstructured documents from which they were created. If you have any files in the `fminput` folder open in *FrameMaker*, you should close them before renaming the folder. If you experience problems trying to the rename the folder, you may have to exit from *FrameMaker* and try again.

9. Select *Edit→Update Book...* from the main menu and click the *Update* button in the dialog box. Save the book when the update process has completed.

When you have identified and fixed all of the problems, and restructured the documents, you can proceed to the next step: converting the files to XML. If you have missed any formatting errors, they will be caught later by DocMaker and you will have to fix the files and reprocess them.

> **Note**
>
> If you need to re-run the structuring process, you should rename `fminput.hide` back to `fminput` first. Then, before updating the newly structured book, you should rename `fminput` to `fminput.hide`. This procedure is a little awkward, but if you do not do it, the cross-references will most likely be wrong but DocMaker will not be able to correct them.

## B.7 Converting to XML

The next task is to convert the structured FrameMaker documents to XML format. A *structure application* document is used to control *FrameMaker* when it is writing the XML output and converting embedded graphics.

1. Using your file manager, go to the folder containing your original book files and create a new sub-folder named `xmloutput`. This will be used to hold the saved XML files and graphic files that will later be converted to DocBook format.

2. Go to the `fmoutput` folder containing your newly structured document files and remove any backup files created during the structuring process.

3. In *FrameMaker*, open the file named `fm2fmdb_app.fm` located in the `lib` folder of your DocMaker installation.

4. Change the value of the `DOCMAKER_HOME` variable to the path to your

DocMaker installation folder by selecting *Special→Variable...* from the menu. In the dialog box, select the `DOCMAKER_HOME` variable from the *Variables* list. Press the *Edit Definition...* button. In the next dialog box, change the path in the *Definition* field to the path to your Doc-Maker installation. Note, you must type two back-slash characters for each back-slash used in the path, for example, use `C:\\DocMaker` and not `C:\DocMaker`. Press the *Done* button in each dialog box to apply the new variable value and save the file. You will only need to do this once unless the location of your DocMaker installation changes or it is reinstalled.

5. From the main menu, select *File→Structure Tools→Read Application Definitions*. You will not need to do this again unless you restart *FrameMaker*.

6. From the main menu, select *File→Utilities→Convert Documents to Structured Format...*.

7. In the dialog box, set the *Application* to "CúramDocMaker". In the *Input FrameMaker Files* section, set the *Directory* field to the location of this book's `fmoutput` folder and set the *Suffix* field to "fm". In the *Output Structured Documents* section, set the *Directory* field to the location of the `xmloutput` folder you created in the first step and set the *Suffix* field to "xml". Check the *Allow Existing Files to Be Overwritten* check-box and press the *Convert* button. Check the log window for any errors and correct them. Errors reported against files that do not contain chapters or appendices can be ignored.

You should now see XML files in your `xmloutput` folder. These are in an intermediate format that will soon be converted to correct DocBook XML.

## B.8 Creating the `book.xml` File

*FrameMaker* produces no useful conversion of the main book file, so the DocBook XML equivalent must now be created. This file, `book.xml`, contains references to all the other files in the book and allows DocMaker to process the book as a whole. DocMaker can create this file automatically, but you will still need to check it manually and possibly correct the ordering of the content.

1. Open a command prompt, and change the current folder to the folder containing your book files and the `xmloutput` folder.

2. At the command prompt, run the following command:

```
C:\MyBook> docmaker prepare-fm
```

This will create a `book.xml` file in the `xmloutput` folder. The book will contain a reference to every XML file (in alphabetical order) cre-

ated by *FrameMaker*. Files that match the name patterns `*TOC.xml`, `*PrintHistory.xml`, `*Cover.xml`, `*IX.xml` (but not `*Unix.xml`), and `*.backup.xml` will be excluded automatically. If you have a file containing content that matches one of these patterns, you must add it to the `book.xml` file manually.

3. Open the `xmloutput/book.xml` file in an editor and locate the line defining the title of the book:

```
<book>
  <title>Unknown</title>
  ...
</book>
```

Change the text within the `title` element to the correct title of your book.

4. Below the book title, you will see *XInclude references* for each main content file in your book. Reorder the references if necessary. If a file was not included (perhaps one of the file names has the same ending as the files that are automatically excluded), add a new reference in the appropriate place within the `book` element copying the format of those present.

Once the `book.xml` file is ready, you can now convert your documents to correct DocBook XML format. If you encounter errors, you will have to fix them in the original files, but you will not have to recreate the `book.xml` file.

## B.9  Converting to DocBook XML

*FrameMaker* cannot convert your documents into correct DocBook XML, though it does come close, so DocMaker will now take over and complete the conversion process. DocMaker will also restructure your document and place one chapter or appendix in each output file. The files are named using the *normalized title* (see Section 6.8, *Cross-References* for details) of the chapter or appendix and a prefix. For chapters, the prefix is "c_" and for appendices, "a_". Any graphic files will be copied and placed in an `images` sub-folder of your document folder.

1. Using your file manager, go to the folder containing your book files and create a new sub-folder named `dboutput`. This will be used to hold the final DocBook XML files and the `images` sub-folder containing all the graphic files.

2. Open a command prompt, and change the current folder to the folder containing the `dboutput`, and `xmloutput` folders.

3. At the command prompt, run the following command:

```
C:\MyBook> docmaker convert-fm
```

DocMaker will now convert the `xmloutput/book.xml` file to Doc-Book XML format and, as this file includes all the others, all of your documentation will be processed. Each chapter and appendix will be placed in a separate file and a new `book.xml` file will be created that references them. Graphic files (identified by DocMaker as those files that do not have a `*.xml` extension) will be copied to the `dbout-put/images` folder.

The chapter and appendix files are named using the title of the chapter or appendix. Some titles containing characters such as "ú" may cause problems during builds. You can fix these by renaming the file and then renaming the references to the file in `book.xml`.

If your document contains a code sample that uses an XML `CDATA` block, it is likely that *FrameMaker* will not generate valid XML. If you get an error caused by this, edit the XML file and change the block's closing "]]>" sequence to "]]&gt;".

If an error message appears reporting that a cross-reference could not be resolved, it is nearly always caused by the use of the wrong cross-reference format. For example, a cross-reference to a figure that is formatted using the "Table Reference" format will cause such a problem.

The conversion process is now complete. The new files in `dboutput` are in valid DocBook XML format. However, a few clean-up tasks remain.

## B.10  Processing Graphics

While *FrameMaker* exports all the diagrams and images in your document to individual files, it does not embed adequate information in the XML documents for DocMaker to create proper references to the graphic files. You will need to rename the graphic files and check the references to these files created by DocMaker. *FrameMaker* was also instructed to output graphic files only in the GIF (Graphics Interchange Format) and WMF (Windows Mediafile) formats. The former is used for all images and the latter for diagrams drawn within *FrameMaker*. The WMF files must be converted to SVG format to be compatible with DocMaker.

Converting WMF to SVG

The *OpenOffice.org Draw* application can do this type of conversion. Open the WMF file, select the graphic, and then use the *File→Export...* menu option to export the file to SVG format.

SVG Containing Bitmaps

If a WMF format image file contains both a bitmap (e.g., a screenshot) and drawing elements (e.g., arrows and text), then DocMaker may not be able to process the SVG file created during conversion. It is best to convert these files to GIF image files.

The names of the image files match those of the original *FrameMaker* file in which they appeared, though with the addition of a "sequence number" and a different extension. For example, the fifth image that was in the file `UserGuide.fm` would be output to a file named `UserGuide-5.gif`. The sequence numbers begin anew with each file in the book; books that contained only one main content file will result in image files with unique sequence numbers; books that contained many files will result in image files with sequence numbers that repeat as each file is processed. DocMaker can preserve the sequence numbers, but cannot determine the proper file names, so these must be fixed by hand.

DocMaker must invent file names for its references to the image files because the original file names are unavailable. The names that DocMaker uses are composed from the chapter or appendix name and the image sequence number. If the book contained a single main content file, it is just a matter of finding the `imagedata` elements in the DocBook XML files and renaming the image file with the same sequence number to match the value of the `fileref` attribute. For example, the value of the `fileref` attribute might be `images/concepts-34.gif` (meaning the "Concepts" chapter and the thirty-fourth image that appeared in the file containing that chapter), so you should locate the image file with sequence number `34` and rename it to `concepts-34.gif`. If the book contained more than one main content file, you may have more than one image file with the same sequence number and you must ensure both that the sequence numbers match and that the original *FrameMaker* file, whose name is used in the name of the image file, contained the chapter whose name is used in the `fileref` attribute reference.

This process is relatively simple, but there is a possible cause of confusion. If a chapter was split over several files with sections in these file containing images, then the reference in the DocBook XML file may be ambiguous. In this case, the reference will use the chapter name and sequence number, but as there were several files in the chapter, the sequence number in the image files will have been repeated. For example, `images/development-2.gif` could refer to the second image in *any* of the original files that contributed content to the "Development" chapter; you may see this reference repeated more than once if more than one file contained two or more images. To resolve this, you will need to identify the section that contains the reference and then match that to the image file named after the file that contained that section; the sequence numbers will then match as before. However, you will need to assign new numbers and change the value of the `fileref` attributes so that all these references are uniquely named.

> **Note**
>
> When renaming files, there is no need to keep the sequence numbers; these are just a convention used in the conversion process. You can rename the files to anything you like (you could base the name on the title of the figure), as long as you change the value of the `fileref` attribute to match. You can also organize the image file in sub-folders of the `images` folder and, again, update the reference

in the `fileref` attribute.

The following steps must be followed for each file in the `dboutput/images` folder.

1.  If the file extension is `.wmf` convert the file to SVG format with an extension of `.svg` (see above).

    You can keep the WMF format file if you wish to use it as the *master* copy of the graphic, but if you change it, you must update the SVG file.

2.  View the image file and then locate the matching figure in your original *FrameMaker* document. The exported graphic files have the same name as the original document file and an additional sequence number. This should make it easier to locate. Once located, note the figure title.

3.  Open the DocBook XML file in `dboutput` for the chapter or appendix that contains the figure and locate the `figure` element that contains the reference to the graphic file. The file name will be different, so use the figure title that you noted in the previous step to identify the correct `figure` element.

    For figures without titles, you will have to locate the corresponding `informalfigure` element. As there is no title, use the sequence number and the context as a guide.

4.  Within the `figure` element you will see an `imagedata` element containing a reference to an image file. Note the value of this `fileref` attribute. If the extension on the file name is `.gif`, but the graphic file is in SVG format, change the extension to `.svg` and change the value of the `format` attribute to `SVG`.

5.  Rename the graphic file to match the name in the `fileref` attribute. Note, however, that if the original document file contained only sections and no chapter or appendix title, that you may need to assign a new sequence number as described above.

## B.11  Final Tasks

Your new DocBook document is now only missing one thing. The legal notice and printing history has to be added. A suitable starting point is the standard notice supplied in DocMaker's sample book templates.

1.  Locate the DocMaker installation folder and open the file `templates/book/compound/book.xml`.

2.  Copy the entire `bookinfo` element.

3.  Open the `dboutput/book.xml` file containing your new book and paste the `bookinfo` element between the book's title and the XInclude references.

4.  Update the copyright and print history and add any extra legal notices

if required before saving the file.

You have now completed the conversion process. The contents of the `dboutput` folder can be relocated to an appropriate place in your documentation hierarchy and placed under SCM control. You should run DocMaker's spell-checking and rules-checking targets to identify any other problems that may need attention. Generating the PDF documentation and examine it; here are some of the more common issues that may need to be resolved:

- Footnote references in the text (usually denoted by a superscript number in the original FrameMaker file) that appear inside inline mark-up may cause validation errors. This is because the `footnote` element is invalid inside elements such as `emphasis` and `literal`. During the conversion process, these footnotes are moved from within the element to just after it if they appear at the end of the mark-up. However, footnotes that are not at the end of the mark-up cannot be safely moved without knowledge of the context and must be moved manually.

- Tables that have no titles can cause problems when DocMaker generates ID values for cross-references. Adding titles to the tables will fix this.

- Words that are valid in the document but not in the spelling dictionary, for example, property names, must be added to a new word list. You can do this by running DocMaker's spell-checker.

- Tables are inserted into *FrameMaker* documents using a special anchor that is usually located in the previous paragraph. When converted, the DocBook `table` element will be embedded in the corresponding `para` element and should be moved outside the paragraph.

- Lists that contain complex content or unusual paragraph sequences may not be identified correctly and may need to be restructured.

- Instructions describing menu selections should be updated to use the `menuchoice` element.

- Quotation marks should be replaced with the `quote` element.

# Notes

[1] At this stage, the structure is not valid DocBook XML, but it will be transformed in the final stage to produce correct output.

[2] A "straight double quote" is ASCII (and ISO "Latin 1") character 34 decimal. It can be typed literally into the document or the character entities `&quot;` or `&#34;` can be used.