IBM Cúram Social Program Management

Cúram DocMaker Manual

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.

This edition applies to version 6.0.4 of IBM Cúram Social Program Management and all subsequent releases and modifications unless otherwise indicated in new editions.

Table of Contents

Chapter 1 Introduction 1
1.1 What is DocMaker? 1
1.2 What is DocBook? 1
1.3 What's Wrong with the Way Things are Now? 2
1.4 How Does DocBook Solve These Problems? 2
1.5 What is this Manual? 3

Chapter 2 Using DocMaker 4
2.1 Introduction 4
2.2 Folder Structure Patterns 4
2.3 File Naming Patterns 5
2.4 Running DocMaker 7
2.5 Dependency Checking 13

Chapter 3 The Document Hierarchy 14
3.1 Introduction 14
3.2 Hierarchy Elements 14
3.3 Books 15
3.3.1 Book Structure 15
3.3.2 Book Information 15
3.3.3 Book Categories 16
3.3.4 Revision Histories 17
3.3.5 Books with Long Titles 18
3.4 Parts 18
3.5 Chapters 19
3.6 Appendices 20
3.7 Sections 20
3.7.1 Section Structure 20
3.7.2 Nesting Sections 21

Chapter 4 Block Elements 22
4.1 Introduction 22
4.2 Paragraphs 22
4.2.1 Overview 22
4.2.2 para 22
4.2.3 formalpara 23
4.3 Lists 23

4.3.1 Overview 23
4.3.2 itemizedlist 24
4.3.3 orderedlist 25
4.3.4 variablelist 25
4.4 Figures and Examples 26
4.4.1 Overview 26
4.4.2 figure 27
4.4.3 mediaobject 27
4.4.4 Accessibility 28
4.4.5 example 29
4.4.6 programlisting 30
4.5 Tables 32
4.5.1 Overview 32
4.5.2 table 32
4.5.3 colspec 33
4.5.4 Accessibility 34
4.5.5 The Peculiar Content of entry Elements 35
4.5.6 Spanning Columns 37
4.6 Admonitions 39
4.6.1 Overview 39
4.6.2 Titles 40
4.6.3 Choosing an Admonition 40
4.7 Sidebars 41
4.7.1 Overview 41
4.7.2 sidebar 41
4.8 Block Quotes 41
4.8.1 Overview 41
4.8.2 blockquote 42
4.9 Literal Layout 42
4.9.1 Overview 42
4.9.2 literallayout 43

Chapter 5 Inline Elements 44
5.1 Introduction 44
5.2 Using Inline Elements 44
5.3 Basic Inline Elements 45
5.3.1 Overview 45
5.3.2 quote 45
5.3.3 emphasis 46
5.3.4 firstterm 46
5.3.5 envar 46
5.3.6 filename 46
5.3.7 replaceable 46
5.3.8 citetitle 46
5.3.9 ulink 46
5.3.10 email 47
5.3.11 systemitem 48
5.3.12 database 49
5.3.13 foreignphrase 49

5.3.14 command 49
5.3.15 option 50
5.3.16 application 50
5.3.17 productname 50
5.3.18 trademark 50
5.3.19 acronym 51
5.3.20 abbrev 51
5.3.21 medialabel 51
5.3.22 remark 51
5.3.23 superscript 52
5.3.24 subscript 52
5.4 Program Source Code 52
5.4.1 Overview 52
5.4.2 sgmltag 52
5.4.3 classname 54
5.4.4 interfacename 54
5.4.5 methodname 54
5.4.6 type 54
5.4.7 parameter 55
5.4.8 varname 55
5.4.9 constant 55
5.4.10 property 55
5.4.11 literal 55
5.4.12 symbol 55
5.4.13 errorcode 56
5.4.14 errortext 56
5.4.15 errorname 56
5.4.16 errortype 56
5.5 Command-line Environments 56
5.5.1 Overview 56
5.5.2 prompt 56
5.5.3 userinput 56
5.5.4 computeroutput 57
5.5.5 Combining Elements 57
5.6 User Interfaces and User Interaction 57
5.6.1 Overview 57
5.6.2 mousebutton 57
5.6.3 keycap 58
5.6.4 keycombo 58
5.6.5 guibutton 59
5.6.6 guilabel 59
5.6.7 guiicon 59
5.6.8 guimenu 59
5.6.9 guisubmenu 59
5.6.10 guimenuitem 59
5.6.11 menuchoice 60
5.7 Footnotes 60
5.8 Cross-References 61
5.8.1 Introduction 61

5.8.2 Internal Cross-References 62
5.8.3 Context-Sensitivity 64
5.8.4 External Cross-References 65
5.9 Inline Images 66

Chapter 6 Character Entities 68
6.1 Entities 68
6.2 Punctuation Marks and Accented Characters 70
6.3 Numeric and Symbol Entities 70

Chapter 7 Editing DocBook XML Documents 72
7.1 Introduction 72
7.2 Choosing an Editor 72
7.2.1 Text Editors vs. XML Editors 72
7.2.2 Basic Requirements 74
7.2.3 Selected Editors 74
7.3 Using an Editor 75
7.3.1 Overview 75
7.3.2 Encoding 75
7.3.3 Formatting 77
7.4 Summary 78

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

Using DocMaker

2.1 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.

2.2 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.

2.3 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 2.1 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 2.2 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.

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.

2.4 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 2.1 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, DocMaker 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
```

2.5 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 3

The Document Hierarchy

3.1 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.

3.2 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.

3.3 Books

3.3.1 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.

3.3.2 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.

3.3.3 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>
```

3.3.4 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.

3.3.5 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>
```

3.4 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.

3.5 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.

3.6 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>
```

3.7 Sections

3.7.1 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.

3.7.2 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 4

Block Elements

4.1 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.

4.2 Paragraphs

4.2.1 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.

4.2.2 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.

4.2.3 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.

4.3 Lists

4.3.1 Overview

Lists allow information to be arranged in sequence and help to organize information effectively. There are a number of different list types in DocBook: `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.

4.3.2 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.

4.3.3 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.

4.3.4 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.

4.4 Figures and Examples

4.4.1 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.

4.4.2 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 4.1 A Sample Image

4.4.3 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 DocMaker.

`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.

4.4.4 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>
```

4.4.5 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 4.1 An Example of Some Java Code

4.4.6 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 DocBook 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.

4.5 Tables

4.5.1 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.

4.5.2 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 4.1 The Phone List

4.5.3 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 4.2 The Phone List Again

4.5.4 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 4.3 Another Phone List

4.5.5 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 4.2 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 5, *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 4.3 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.

4.5.6 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	021	555-6789

Table 4.4 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 4.5 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.

4.6 Admonitions

4.6.1 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 going any further.

4.6.2 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..

4.6.3 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.

4.7 Sidebars

4.7.1 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.

4.7.2 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.

4.8 Block Quotes

4.8.1 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.

4.8.2 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

4.9 Literal Layout

4.9.1 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.

4.9.2 literallayout

The `literallayout` element structure is like that of a `programlisting`. 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 5

Inline Elements

5.1 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.

5.2 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.

5.3 Basic Inline Elements

5.3.1 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.

5.3.2 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.

5.3.3 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.

5.3.4 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.

5.3.5 envar

Use this for the name of an environment variable.

5.3.6 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.

5.3.7 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`.

5.3.8 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.

5.3.9 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 `citetitle` 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 5.3.11, *systemitem* .

5.3.10 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.

5.3.11 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 5.1 Values of the **`class`** Attribute of **`systemitem`**

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

5.3.12 `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 5.2 Values of the **`class`** Attribute of **`database`**

5.3.13 `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).

5.3.14 `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.

5.3.15 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 5.5, *Command-line Environments*, for more information.

5.3.16 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.

5.3.17 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.

5.3.18 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.

5.3.19 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.

5.3.20 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.

5.3.21 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*.

5.3.22 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.

5.3.23 superscript

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

5.3.24 subscript

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

5.4 Program Source Code

5.4.1 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.

5.4.2 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 ` ` 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 5.3 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.

5.4.3 `classname`

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

5.4.4 `interfacename`

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

5.4.5 `methodname`

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

5.4.6 `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++ `typedef`s. 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>>.

5.4.7 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.

5.4.8 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.

5.4.9 constant

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

5.4.10 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.

5.4.11 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!

5.4.12 symbol

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

5.4.13 errorcode

Use this for the unique number identifying an error.

5.4.14 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 `replaceable` element and the whole message marked with the `errortext` element.

5.4.15 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.

5.4.16 errortype

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

5.5 Command-line Environments

5.5.1 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).

5.5.2 prompt

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

5.5.3 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.

5.5.4 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.

5.5.5 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`

5.6 User Interfaces and User Interaction

5.6.1 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.

5.6.2 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.

5.6.3 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.

5.6.4 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.

5.6.5 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.

5.6.6 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.

5.6.7 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.

5.6.8 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”.

5.6.9 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.

5.6.10 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.

5.6.11 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.

5.7 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.

5.8 Cross-References

5.8.1 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.

5.8.2 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 5.3.9, *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 supported element, the same scheme is used.

5.8.3 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-sensitive 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 section in the content that should be displayed when help for that page is requested.

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 topicname="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 containing 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.

5.8.4 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.

5.9 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 4.4.3, *mediaobject*).

Notes

[1] UML stands for “Unified Modeling Language”.

Chapter 6

Character Entities

6.1 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 7, *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.

6.2 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 6.1 Punctuation Marks and Accented Characters

6.3 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 6.2 Numeric and Symbol Entities

Chapter 7

Editing DocBook XML Documents

7.1 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.

7.2 Choosing an Editor

7.2.1 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.

7.2.2 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.

7.2.3 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*.

7.3 Using an Editor

7.3.1 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.

7.3.2 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.

7.3.3 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!

7.4 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 6, *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.