

# Producing documentation and reusing information in XML, Part 3: Creating multi-target XML documents

Use single-source documents for multiple audiences and output formats

Skill Level: Intermediate

[William von Hagen](#)

Systems Administrator, Writer  
WordSmiths

07 Jul 2009

XML is an optimal format for writing documentation that you can use with many different documentation software packages and production environments. In this third article in the series, discover how to create single-source documents that can produce output in a variety of different output formats.

## What is single-source documentation?

### Other articles in this series

- [Part 1: Document publishing using XML: Create, format, and publish documents using XML standards and open source tools](#)
- [Part 2: Reuse information in XML documentation: Reduce work, increase usability, and enforce consistency with document reuse](#)

As [Part 1](#) in this series showed, using XML (and its conceptual predecessor, Standard Generalized Markup Language, or SGML) for documentation enables you to separate the *content* of a document—the actual words and data it contains—from

its *presentation*—the way in which the content is displayed in a specific output format or on a specific output device. This separation enables you to focus on the content of that document rather than how the information will eventually be used or presented. [Part 2](#) discussed various mechanisms that enable you to organize XML documentation into smaller units of information (popularly known as *chunking*), to create larger documents or documents for different audiences by threading together different chunks of information or the same chunks of information in different sequences.

### Frequently used acronyms

- DITA: Darwin Information Typing Architecture
- DTD: Document type definition
- HTML: Hypertext Markup Language
- PDF: Portable Document Format
- URL: Uniform Resource Locator
- XML: Extensible Markup Language
- XSL: Extensible Stylesheet Language

A natural outgrowth of XML as a software-independent documentation environment that facilitates information reuse is the need to customize that information so that its content differs based on the specific audience or output format. This reuse is commonly known as *single-source documentation*, because a single set of input files can satisfy the requirements of multiple audiences or output formats. Some single-source requirements are handled automatically by the tools that produce output in different formats. For example, generating PDF output for a DocBook XML document that contains a link to external resources (using the `<ulink>` element) embeds both a hyperlink to that information and its actual URL in that output, while generating HTML output from the same XML document simply embeds a link in that HTML output.

Transforming a single element in different ways for different output formats is a step in the right direction for single-source documentation, but it doesn't enable customization of document content beyond its presentation requirements. Being able to customize the actual content of a document based on its target output format is a fairly common requirement for modern documentation. Luckily, this is easily handled by a combination of preprocessing and taking advantage of flexible aspects of the design of documentation formats such as DocBook XML.

## DocBook elements and attributes

XML elements use name-value pairs called *attributes* to provide additional

information about an instance of an element. Typically, attributes uniquely identify instances of the same element in an XML document or identify specialized behavior for instances of an element. Attributes are contained within the scope of an element and take the form:

```
<attribute_name="value">
```

As an example, the most common attribute for XML elements is the `id` attribute, which uniquely identifies a given instance of an XML element, as in the following example:

```
<section id="introduction">
```

In XML documentation, `id` attributes support cross-referencing through elements such as the `<xref>` and `<link>` elements. Here are examples of references to the `<section>` element used in the previous example:

```
<xref linkend="introduction"/>  
<link linkend="introduction">introduction to this document</link>
```

An example of an attribute that identifies specialized behavior for an element is the `role` attribute of the `<emphasis>` element:

```
<emphasis role="bold">example</emphasis>
```

Content enclosed within an `<emphasis>` element is typically rendered in italics in presentation formats such as PDF or HTML. Specifying the `role="bold"` attribute changes the rendering of this element from italics to bold.

## Alternative approaches to document customization

Historically, markup languages that were oriented toward document production supported some degree of customization of the set of elements they supported. Creating custom elements that can be used in specialized ways within a given tool was a feature of the original Generalized Markup Language, or GML, the predecessor of SGML (see [Resources](#) for more information). As the acronyms suggest, a primary difference between GML and SGML was the standardization of the set of elements that were present in a given document type definition.

Even though SGML and, later, XML document types were designed to use a standard, predefined set of elements, it is still possible to add custom element definitions by referencing external resources that define those elements. However,

adding custom element definitions to an existing DTD or schema is typically a bad idea for various reasons, most importantly:

- Documents that use non-standard elements no longer comply with standard document types such as DocBook or DITA.
- Extending a DTD or schema limits your opportunities for document exchange with other institutions or companies that use the default DTDs or schemas. Even if you also exchange the definitions for your non-standard elements, more customization work will be required for successful document interchange.
- Tools that are designed to work with standard document types will have to be extended to support the new elements. Although doing so is relatively easy with open source tools, this extension might be impossible with proprietary, closed source tools.

This article discusses using a preprocessing step to eliminate conditionalized text that does not apply to a specific audience or output format. Some graphical XML documentation tools provide an equivalent solution by enabling you to set variables used during the output-generation process. This article focuses on the more generic approach of preprocessing that can be used by a wide range of open source XML documentation tools rather than on the mechanisms supported by specific documentation tools.

## Common attributes for DocBook elements

As you might expect, different XML elements have different attributes based on the type and use of a specific element. However, XML documentation schemas and DTDs such as the DocBook document type that is the focus of this series of articles also define a common set of attributes (see [Resources](#) for more information) that can be used on any XML element and that are typically used to identify presentation- or target-specific information for a given element. Common examples of these attributes include:

- **arch:** Designed to identify the computer system or processor architecture to which the content within a given instance of an element applies
- **audience:** Designed to identify the audience to which the content within a given instance of an element applies
- **condition:** Designed for local, application-specific content customization
- **os:** Designed to identify the computer operating system to which the content within a given instance of an element applies

- **revision:** Designed to identify a specific software or document revision to which the content within a given instance of an element applies
- **vendor:** Designed to identify the hardware or software vendor to which the content within a given instance of an element applies

Defining a set of valid values for any of these attributes, then implementing a preprocessor that discards elements that have other values for that attribute provides an easy way to *conditionalize* your documents. The attributes you decide to use for customization are entirely up to you, but the choice typically depends on which of these attributes best applies to the type of content you want to conditionalize and the reason for that customization. This article uses the generic `condition` attribute as an example of general conditionalization that is specific to a given output/presentation format.

The next two sections explain how to use attributes such as these to conditionally identify specific elements or content fragments within an element.

## Conditionally including entire elements

The common attributes discussed in the [previous section](#) provide an easy way to identify the portions of your documents that you want to associate with specific audiences or output formats. This section uses the `condition` attribute as an example, but you can use any of the common elements discussed in the previous sections as long as they are not already used at your site.

A common type of conditionalized content is text that appears in a document when it is formatted for use in different presentation formats. For example, documents that are designed for use online often include links to the next section at the end of each section to simplify navigation for the reader. These links can look like this:

```
<para>
To proceed to the next section of this tutorial, click
<link linkend="link-to-next-section">here</link>.
</para>
```

Although useful in online documents, this information is both redundant and confusing when the same document is formatted as an Adobe® PostScript® or PDF document. To identify this portion of the document as intended only for use when the document is formatted for online presentation, you can add the `condition="online"` attribute to the `<para>` element, as in the following example:

```
<para condition="online">
To proceed to the next section of this tutorial, click
```

```
<link linkend="link-to-next-section">here</link>.  
</para>
```

## Adding preprocessing to the document-formatting process

It would be nice if the simple addition of the attributes discussed in the [previous section](#) to a given element would do the right thing for your documentation tools, but that is rarely the case. To take advantage of this sort of conditionalization, you need to write a small XSL script that discards that portion of your document when you are formatting it for other presentation formats. [Listing 1](#) shows an example of an XSL script that discards content elements that have a `condition` attribute with a value other than `print`.

### Listing 1. Preprocessing script for print output

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"  
                xmlns:xi="http://www.w3.org/2003/XInclude">  
<xsl:output method="xml" omit-xml-declaration="yes" indent="yes"/>  
<xsl:preserve-space elements="*" />  
  
<xsl:template match="*|@*">  
  <xsl:variable name="_element">  
    <xsl:value-of select="name()" />  
  </xsl:variable>  
  <xsl:variable name="_condition">  
    <xsl:value-of select="@condition" />  
  </xsl:variable>  
  <xsl:choose>  
    <xsl:when test="$_condition = ''">  
      <xsl:copy>  
        <xsl:apply-templates select="@* | * | text() | comment()" />  
      </xsl:copy>  
    </xsl:when>  
    <xsl:when test="$_condition = 'print'">  
      <xsl:copy>  
        <xsl:apply-templates select="@*|node()" />  
      </xsl:copy>  
    </xsl:when>  
    <xsl:otherwise>  
      <xsl:message> Skipping <xsl:value-of select="$_element"/></xsl:message>  
    </xsl:otherwise>  
  </xsl:choose>  
</xsl:template>  
  
</xsl:stylesheet>
```

**Note:** This script was written for readability, not elegance. You can certainly simplify it.

Continuing with the preceding example, this script in [Listing 1](#) discards elements whose `condition` attribute identifies them as being targeted for other output formats. This targeting removes the content shown in the preceding "online-only" example, because its `condition` attribute had the value `online`. The first portion of the script processes each element and assigns the name of the element and any

value for a `condition` attribute to associated variables. The remainder of the script conditionally outputs content based on the value of the `condition` attribute. If no `condition` attribute is present or the attribute has a value of `print`, the containing element is copied to the output of the script. If the `condition` attribute has any other value, the associated element is suppressed in the output, and processing proceeds.

To easily integrate a script such as this one into your document-production process, add it to the portion of a Makefile that produces PDF or PostScript documentation. To do so, you use this script to process your input document, redirecting its output into a file, which you then format appropriately.

[Listing 2](#) shows the same XSL script, customized to discard content elements that have a `condition` attribute with a value other than `online`. Adding this script to a Makefile target for online documentation enable you to include elements such as the one in [previous code examples](#) as being online only.

## Listing 2. Preprocessing script for online output

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
               xmlns:xi="http://www.w3.org/2003/XInclude">
<xsl:output method="xml" omit-xml-declaration="yes" indent="yes"/>
<xsl:preserve-space elements="*" />

<xsl:template match="*|@*">
  <xsl:variable name="_element">
    <xsl:value-of select="name()" />
  </xsl:variable>
  <xsl:variable name="_condition">
    <xsl:value-of select="@condition" />
  </xsl:variable>
  <xsl:choose>
    <xsl:when test="$_condition = ''">
      <xsl:copy>
        <xsl:apply-templates select="* | * | text() | comment()" />
      </xsl:copy>
    </xsl:when>
    <xsl:when test="$_condition = 'online'">
      <xsl:copy>
        <xsl:apply-templates select="@*|node()" />
      </xsl:copy>
    </xsl:when>
    <xsl:otherwise>
      <xsl:message> Skipping <xsl:value-of select="$_element"/></xsl:message>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>

</xsl:stylesheet>
```

A comparison of the XSL code in [Listing 1](#) and [Listing 2](#) shows that they only differ by a single word, which makes them good candidates for simplification through the support for parameters (`stringparam`) provided in XSL and therefore in `xsltproc`. I leave this as an exercise, because the goal of these examples was clarity, not a crash course in writing elegant XSL.

## Conditionally including content fragments

The examples in the [preceding section](#) are well suited to include or exclude entire elements based on an attribute value. However, most conditionalized XML documentation also requires conditionalizing portions of the text in another element. Most commonly, you might need to customize specific content within a paragraph. This is easily done in two ways:

- Conditionalize elements that can appear within a paragraph
- Use the `<phrase>` element to conditionalize portions of the text within a paragraph

To conditionalize elements that can appear within a paragraph, add the attribute that you are using for conditionalization and an appropriate value to the appropriate element within a paragraph. For example, the document fragment shown in [Listing 3](#) provides a conditionalized `<mediaobject>` element within a paragraph (formatted for readability).

### Listing 3. A conditionalized mediaobject element

```
<para>
  To create a new file, click the <emphasis
  role="bold">Add</emphasis> icon
  (<mediaobject>
    <imageobject audience="online">
      <imagedata fileref="add.gif"/>
    </imageobject>
    <imageobject audience="print">
      <imagedata fileref="../../images/add.gif" />
    </imageobject>
  </mediaobject>)
  beside the <emphasis role="bold">New File</emphasis> menu entry.
</para>
```

Although primarily used as an example here, conditionalizing a `<mediaobject>` element enables you to specify different paths to locate the graphic in different presentation formats. In this case, online documents find the graphic in the centralized location for graphics on a given Web server or virtual Web server, while print documents locate the graphic in a specific directory for formatting purposes.

Using the `<phrase>` element to identify sections of conditionalized text is somewhat more interesting. With the `<phrase>` element, you can identify a specific range or span of text that is smaller than a paragraph and is perfectly suited to document conditionalization. [Listing 4](#) (formatted for readability) shows an example of conditionalizing a single word to make documentation more specific to a given presentation format.

## Listing 4. Conditionalizing a single word

```
<p>
  See the
    <link linkend="target-id">
      introduction to this
        <phrase audience="online">portion</phrase>
        <phrase audience="print">chapter</phrase>
      of the documentation
    </link>
  for more information.
</p>
```

## Conclusion

The power and flexibility of XML, sets of existing standards, and a rich set of tools for working with and converting XML documents provide a powerful environment for creating and maintaining documentation. The attributes and techniques discussed in this article make it easy to create conditionalized documentation that can contain different content targeted toward specific audiences, computer systems, or presentation formats. If you add a simple preprocessing stage or set variables for use in your documentation-production process, you can create and maintain single-source documentation that produces specialized output. This functionality provides an eminently useful and flexible solution to the traditional documentation problem of simplifying documentation development and maintenance while still best serving the requirements of specific sets of users of that documentation.

## Downloads

Description	Name	Size	Download method
Sample XSL script for print output	conditional-print.zip	1KB	<a href="#">HTTP</a>
Sample XSL script for online output	conditional-online.zip	1KB	<a href="#">HTTP</a>

[Information about download methods](#)

# Resources

## Learn

- [Producing documentation and reusing information in XML, Part 1: Document publishing using XML](#) (William von Hagen, developerWorks, March 2009): Easily assemble an XML publishing system on any UNIX® or Linux™ system. Combine free, open source packages with XML for efficient re-use of information and produce documents in multiple formats from one set of source documents.
- [Producing documentation and reusing information in XML, Part 2: Reuse information in XML documentation](#) (William von Hagen, developerWorks, March 2009): Structure XML documents and fragments for reuse. With XInclude, include external document sections, and with XPointer, include small document fragments into a larger document in Part 2 of this three-part series.
- [DocBook: The Definitive Guide](#): Find complete information about DocBook markup in SGML and XML. The complete text of this book is available online, but it's well worth buying a copy if you do serious publishing work with DocBook.
- [DocBook XSL: The Complete Guide](#): Read definitive information about DocBook style sheets and publishing. The complete text of this book is available online, but consider buying a copy if you do serious publishing work with DocBook.
- [Common attributes](#): See a list of the common set of DocBook XML elements.
- [GML](#): Read Wikipedia's entry on the Generalized Markup Language.
- [IBM XML certification](#): Find out how you can become an IBM-Certified Developer in XML and related technologies.
- [XML technical library](#): See the developerWorks XML Zone for a wide range of technical articles and tips, tutorials, standards, and IBM Redbooks.
- [developerWorks technical events and webcasts](#): Stay current with technology in these sessions.
- [Technology bookstore](#): Browse for books on XML, DocBook, and other technical topics.
- [developerWorks podcasts](#): Listen to interesting interviews and discussions for software developers.

## Get products and technologies

- [DocBook DTD](#): Download the latest version and get started.
- [DocBook XSL Stylesheets](#): Download the latest version. (Go to **DocBook Project site > file releases** and select **docbook-xsl**.)

- [nXML mode](#): If you plan to edit XML documents in the Emacs text editor (and you should), download the latest version of XML mode for Emacs.
- [IBM product evaluation versions](#): Download or [explore the online trials in the IBM SOA Sandbox](#) and get your hands on application development tools and middleware products from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.

## Discuss

- [XML zone discussion forums](#): Participate in any of several XML-related discussions.
- [developerWorks XML zone: Share your thoughts](#): After you read this article, post your comments and thoughts in this forum. The XML zone editors moderate the forum and welcome your input.
- [developerWorks blogs](#): Check out these blogs and get involved in the [developerWorks community](#).

## About the author

William von Hagen

William von Hagen has been a writer and UNIX systems administrator for more than 20 years and a Linux advocate since 1993. Bill is the author or co-author of books on subjects such as Ubuntu Linux, Xen Virtualization, the GNU Compiler Collection (GCC), SUSE Linux, Mac OS X, Linux file systems, and SGML. He has also written numerous articles for Linux and Mac OS X publications and Web sites.

## Trademarks

IBM, the IBM logo, ibm.com, DB2, developerWorks, Lotus, Rational, Tivoli, and WebSphere are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. These and other IBM trademarked terms are marked on their first occurrence in this information with the appropriate symbol (® or ™), indicating US registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. See the current list of [IBM trademarks](#).

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.