
X-Power

Use XML to write papers

Abstract

Having authored a number of papers, I've begun to run into the pain and agony of having to write while fighting with my chosen composition environment (that is, MSWord) and how to format things. For example, getting fonts and paragraph settings configured just the way I like them, standardizing on heading titles, and so on. I realize that I probably don't use the tool (again, MSWord) to a fraction of its complete capability, but I face a difficult choice: learn the tool in order to master it, and consume valuable time I could be writing, or simply live with the shortcomings of my ignorance and keep focused on the technologies I care about.

Fortunately, another solution presented itself in the form of a tool written by DevelopMentor for use in its internal projects. Conceptually, it's a simple idea--capture all content in XML, then render it into the medium desired (Word, PowerPoint, LaTeX, and so on) via tools. I was fortunate enough to be one of the early adopters, and was so enamored of the system that I immediately set out to see if I could replicate success by adopting something similar for my own use in writing papers.

Problem discussion

Almost a year ago, DevelopMentor¹ decided to move to a model whereby content (that is, the interesting stuff we try to produce for students' consumption in class) would be captured in a way different from the norm. Until that point, the model for authoring classes was fairly simple and familiar to many who've been in similar situations: a guy goes into a dark room, works in relative isolation for a while, and emerges with a set of PowerPoint slides ready for consumption by the world at large.

The reasons behind this changeover are legion, but this is neither the time nor the place (nor the author) to discuss them. In short, what emerged from this effort was a process by which an author (me, among others) would produce XML files, which would then be run through a series of steps that would ultimately turn it into a PowerPoint slide deck and corresponding MSWord file. The slide deck would be simply the captured "points" highlighting the key details from the corresponding prose stored in the MSWord document. The MSWord document would also contain space in the document for student notes, so that it could be used as the electronic source for the students' course books. In short, it was an awesome experience to be one of the early adopters of the toolset.

For me, this was going to be my first "real" experience with XML in a production environment--it would be the first time I've used XML "in anger". I'd certainly dabbled in XML, used it as a configuration syntax, cheated on HTML parsing by using an XML parser (and simply requiring my users to offer up XHTML 1.0-compliant HTML), and so on, but never really sat down and used it as the core of a system. This would be exactly that--a system built entirely around a suite of XML technologies.

After playing around with this new system, I realized that while I was certainly authoring in XML, I wasn't really "using" it--the key being that I had no experience in creating such a system. Certainly the technologies involved held great promise for being able to "weave" together the presentation and content layers (developed separately), but I wasn't getting any of that just by using it.

I also began to realize that I was experiencing similar problems to DevelopMentor's on a much smaller scale. For example, under the current website, I present a separate page for each paper, with the abstract from the paper captured on the web page, and links to the PDF of the paper (and an accompanying .jar file containing the code, if any) alongside. I realized that if I ever changed an abstract within a paper, I would need to remember to go back and change the abstract on the website, a clear violation of one of the fundamental precepts of Computer Science:

Thou Shalt Write It Once

--The Once And Only Once Rule

Therefore, and in the spirit of guys everywhere who just can't claim to know a technology until they've replicated something in it², I immediately set off to create my own system by which to take XML as the authoring medium for my white papers, and produce some sort of acceptable output out the other end.

Solution discussion

When discussing this sort of transformation process, the first thing that immediately leaps to mind is the XML Stylesheet Language: Transformations specification, or XSLT for short. Voluminous references and tutorials have been written on XSLT that I freely recognize that I'll not be able to do XSLT justice within a short whitepaper; therefore I'll refer readers to [exml] or [xmlcomp] , and let you read about how wonderful XSLT is from those sources. The long and short of it is, XSLT allows one to transform an input XML document into some sort of output format. What had originally stumped me about adopting this process was the simple question of what I would render to; certainly PDF ultimately boils down to a text-based format³, but I didn't particularly feel like becoming enough of a guru on PDF to be able to render XML directly into PDF.

This led to a search for technologies and existing APIs that would render XML directly into PDF, and one such technology exists: Apache Cocoon, a servlet-based, XSLT-based rendering engine that would take an incoming HTTP request, an input XML file, a transformation XSL file, and produce a PDF (among other options) for response back to the requestor. While a useful approach for dynamically-generated websites, the content of my papers is pretty static, and the time taken to render each paper for each individual request would get fairly obtuse over time. Moreover, I just generally dislike on-demand generated documentation, since I'd rather just let people download (and keep) the original file⁴.

XSLT was originally just one half of XSL (the eXtensible Stylesheet Language), with the actual formatting comprising the other half, and it was a chance conversation with Peter Drayton (<http://www.razorfish.com>), another DevelopMentor instructor, that turned me on to the other half: XSL:FO.

XSL:FO. XSL:FO, or the XSL:Formatting Objects specification, is the "silent partner" in the XSL pair of specifications. While most attention is focused on XSLT for doing transformations from XML to "anything", XSLFO focuses specifically on how to prepare content for display on display-centric devices (like paper, or browsers, and so on). It offers some impressive control over how the display of content should be presented, and as part of that discussion, Peter turned me on to the Apache FOP project, a Formatting-Object-to-PDF library.

Of such conversations are projects born.

Needless to say, upon returning to the hotel room, I fired up the laptop, cruised over to the Apache website, and found FOP. Fifteen minutes later, I was starting to write my first .fo file. I rendered it into a PDF file, and when that was finished, I fired up Acrobat to take a look. Within seconds, I knew I was hooked. This would be the delivery mechanism: I would write the paper in an XML format (whose schema I still hadn't yet decided on), XSLT it into XSLFO, then XSLFO/FOP it into a PDF file for widespread consumption.

Details. Naturally, there are about a thousand details to talk about in this system. For this paper, I'll focus on two central pieces that would allow anybody to replicate it in their own environment if they so desire. Specifically, I'll talk briefly about the schema I came up with and a bit how I render it into FO. Then, I'll talk about how I streamlined the build process to make it almost painless to do the rendering from a simple command-line environment.

Schema. The first step was to create an input format; without this, there'd be nothing to transform from. In this case, because I didn't particularly care about standardization or sharing the format with others, I chose not to build a formal XSD (XML Schema Description) to go along with my input format⁵. For this reason, the best way to describe the format of the paper is to simply show the relevant sections:

```
<whitePaper revision="$Revision: 1.1 $">
```

```
<title>X-Power</title>
<subtitle>Use XML to write papers</subtitle>
<author>
  <name>Ted Neward</name>
  <url>http://www.javageeks.com/~tneward</url>
  <mail>tneward@javageeks.com</mail>
  <date><day>12</day><month>June</month><year>2001</year></date>
</author>
<abstract keywords="XML XSLT XSLFO">
  <p> . . . abstract discussion paragraph . . . </p>
  <p> . . . another paragraph . . . </p>
</abstract>
<h1 topic="Problem Discussion">
  <p> . . . discussion . . . </p>
</h1>
<h1 topic="Solution Discussion">
  <p> . . . discussion . . . </p>
</h1>
<h1 topic="Consequences">
  <p> . . . discussion . . . </p>
</h1>
<summary>
  <p> . . . tie it all up, nice and neat . . . </p>
</summary>
<biblio>
  <book id="exml">
    <title>Essential XML (2nd Edition)</title>
    <author>Don Box and Aaron Skonnard</author>
    <isbn>No clue</isbn>
    <publisher>Addison-Wesley</publisher>
  </book>
  <book id="xmlcomp">
    <title>The XML Companion</title>
    <author>Neil Bradley</author>
    <isbn>No clue</isbn>
    <publisher>No clue</publisher>
  </book>
</biblio>
</whitePaper>
```

White paper, in XML format

The entire paper is wrapped in a `whitePaper` element, representing the paper as a whole. It can have a `revision` attribute, within which I put the CVS keyword string⁶ so that it can be displayed within the Colophon (in a later section of the paper).

The first section within the paper are the header elements, `title`, `subtitle`, and the author information inside the `author` element. The `author` information contains information about the author of the paper, and currently only has support for a single author⁷; if I ever co-author a paper, I'll have to amend this.

What follows is the `abstract` element, containing a brief synopsis of what this paper is trying to talk about without giving the punchline away. This is what will show up as a summary of the paper, and what will be used to "browse" the paper from the website without having to read the paper in its entirety to discover what the paper is about. Notice that the `abstract` element also takes an attribute, `keywords`, that provide hints to search engines⁸ for people searching along those topics.

Next come a series of text sections, referred to simply as `h1` (for Header-One, reminiscent of HTML's

identically-named tag) sections. Each `h1` element contains a `topic` attribute, indicating what this section is about. I typically break my papers up into three sections, presenting a problem, a solution, and the consequences inherited from this solution. Each section (including the `abstract` section previously and the `summary` section that follows) will be the heart of the paper, and as such are intended to "stand out" from one another--in the current rendering, I render the "topic" name as white text inside of a black bar stretching across the width of the page.

There are also a number of in-line elements that can be used to decorate the prose I write; specifically, these break down to paragraphs (`p`), which can also have `topic` attributes, figures (which can have captions), in-line code font for code-related keywords, `codeFrag` blocks showing off a tightly-related collection of code, `quote` blocks that explicitly quote another source, and `output` blocks that represent what users should see on the console when something is executed or I want to highlight what you should see in directory listings. (There are a few other in-line elements, but they're pretty uninteresting as elements go.)

Thus, a sample paragraph block within a section might read like:

```
<p>This led to a search for technologies and existing APIs that would
render XML directly into PDF, and one such technology exists: Apache
Cocoon, a servlet-based, XSLT-based rendering engine that would take an
incoming HTTP request, an input XML file, a transformation XSL file, and
produce a PDF (among other options) for response back to the requestor.
While a useful approach for dynamically-generated websites, the content
of my papers is pretty static, and the time taken to render each paper
for each individual request would get fairly obtuse over time. Moreover,
I just generally dislike on-demand generated documentation, since I'd
rather just let people download (and keep) the original file<footnote>Certainly,
nothing stops a reader from capturing the result and saving it off under
some local name, but users are less likely to save something off if they
believe or know that it is a dynamically-generated process--why capture
something that is constantly changing?</footnote>.</p>
```

Example paragraph with blocked sections

Note that, although it can't be seen within this example, many `codeFrag` elements have character-data sections inside of them (`<![CDATA[. . .]]>`), "escaping" all of the text within the block and therefore removing me from having to convert all of the "<" into "<" to satisfy XML parsers.

Formatting. Once I'd come to a schema of sorts, what was left was to transform the input into valid XSL:FO format. Again, I'm not going to turn this paper into a huge XSL:FO tutorial--there are far better resources on the web to do that. Instead, I'll pick a single element within my schema, the `output` element, and show the transformation rule that generates the FO to produce the presentation I wanted.

In this case, I wanted output to show up the same way it does on many developers' own machines--white text on black background. (Somewhat reminiscent of old mainframe terminals, I believe. We all just want to crawl back to our roots at heart, I guess.) Therefore, the following block of XML content

```
<output>C:\Foo\bar> echo "This is a test"
This is a test
C:\Foo\bar>
</output>
```

<output> section

should render into something like

```
C:\Foo\Bar> echo "This is a test"
This is a test
C:\Foo\bar>
```

when the rendering to presentation is finished.

To make this happen, I set up an XSLT template to match on the "output" elements:

```
<xsl:template match="output">
  <fo:block background-color="black" color="white"
    border-width="0.5pt" border-color="gray" text-align="start"
    padding-start="6pt" padding-end="6pt" white-space-collapse="false"
    space-before="6pt" space-after="6pt" >
    <fo:inline font-family="Courier" font-size="8pt">
      <xsl:if test="@src != ''">
        ###<xsl:value-of select="@src" />###
        <!-- <xinc:include href="{@src}" parse="text"
xmlns:xinc="http://www.w3.org/1999/XML/xinclude" /> -->
      </xsl:if>
      <xsl:if test="not(@src != '')" >
        <xsl:value-of select="." />
      </xsl:if>
    </fo:inline>
  </fo:block>
</xsl:template>
```

<output>-matching XSLT rule

The `fo:block` block tells XSL:FO that I want to start a new "block" of text, which typically means I want to change the attributes on the text to follow; in this case, I want to change the background color to block, the text color to white, and so on. Of key note is the `white-space-collapse` attribute, which tells XSL:FO not to ignore whitespace, so that my line endings and code formatting will be honored and displayed accordingly⁹. The `space-before` and `space-after` attributes tell how much space to put before and after the block, which in this case will give me about a half-line between the start of the output block and at the end of it.

Next comes an `fo:inline` block, which tells XSL:FO that I want to change attributes on this particular block, in this case, to change the font used to render the subsequent text--I want output to be in monospaced font, for which Courier is the best choice, at least on the Win32-based laptop on which I write these things¹⁰.

Within that `fo:inline` block comes an XSLT `if` block, which does as you'd expect--conduct a test that reduces to a boolean expression, and either execute the block, or not. In this case, I test the current node (the output node) to see if it has a "src" attribute on it--the "@src" notation is shorthand in XPath for "attribute::src", which means to select an attribute node named "src" on the current context node. If it does, then I'll want to take the value of that attribute, and pull its contents in directly into the FO file. To do that, I'll use an `XInclude` block, but for now, I have to ignore it--the Xalan XSLT engine doesn't understand `XInclude` yet¹¹. If the "src" attribute isn't present, then I'll just include the text of the output node directly.

The rest of the XSL file follows along the same lines, matching against elements inside the XML file to produce necessary XSL:FO commands. Once the XSLT file was finished, I created a sample paper, ran the Xalan processor over it using this XSL file, and produced a ".fo" file. With that, I took the Apache FOP library, executed it from the command-line, and produced a .PDF file.

Ahhh, success. Total elapsed time, to produce the XSLT file (not counting the writing time for the paper itself), probably about a day.

Building. As I started writing more and more XML, I found that I was wanting to do a lot of experimentation--write something, XSLT it, FOP it, view the rendered PDF. While it was certainly possible to do the ubiquitous "up-arrow, up-arrow, enter" to navigate back through the shell's history of

previous commands to re-run Xalan to do the XSLT-to-XSLFO transformation, followed by a "down-arrow" to re-run FOP to do the XSLFO-to-PDF rendering, that got old really quickly. What I needed was a simple build script to do the transformations from me, so I could just run a single command-line and get everything I wanted in a single step.

At this point, a number of readers will likely be nodding their heads, knowing precisely where I'm going with this: I needed a simple build tool to control the multiple-steps I was going through, and such a simple (or not-so-simple, as time goes on) build utility exists, again within the open-source community: Ant.

Ant provides the ability to capture build commands within an XML-based configuration file, and will do all necessary dependency-ordering to ensure that step A comes before step B if the inputs to step A change. In this case, I needed the XSLT to run before the XSLFO did. Unfortunately, Ant has no built-in support for running Xalan or FOP directly, so I did what any good open-source developer does: I hacked my own.

Specifically, Ant provides the ability for custom "tasks" to be used, where a "task" is simply some Java code that implements a specific interface that Ant consumes. In this case, that interface is called `org.apache.tools.ant.Task`, and it was trivial to write a simple Ant task that reads some configuration out of the Ant buildfile and pass those parameters over to the Xalan entry point:

```
package com.javageeks.ant.xalan;
import org.apache.tools.ant.BuildException;
import org.apache.tools.ant.Task;
public class XalanTask extends Task
{
    private String xslFile;
    private String inFile;
    private String outFile;
    private String params;
    public void setXsl(String value)
    {
        xslFile = value;
    }
    public void setIn(String value)
    {
        inFile = value;
    }
    public void setOut(String value)
    {
        outFile = value;
    }
    public void setParams(String value)
    {
        params = value;
    }
    /**
     * Execute the task
     */
    public void execute()
        throws BuildException
    {
        if (xslFile == null || inFile == null || outFile == null)
        {
            throw new BuildException("xsl, in and out must be set");
        }
        String[] args =
        {
            "-IN", inFile, "-XSL", xslFile, "-OUT", outFile
        }
    }
}
```

```
};
try
{
    System.out.print("Executing Xalan: ");
    for (int i=0; i < args.length; i++)
    {
        System.out.print(args[i] + " ");
    }
    System.out.println("...");
    org.apache.xalan.xslt.Process.main(args);
}
catch (Throwable t)
{
    throw new BuildException(t.toString());
}
}
```

A Xalan Ant Task

A task to run FOP looks similar, and both tasks are available at the paper's webhome for you to download. (They're incredibly trivial, however, and I strongly encourage anybody using Ant to know how to write a custom task--it's far more efficient in the long run.)

Once this is done, building a paper from its .xml file becomes a simple exercise in Ant script-writing:

```
<project name="PapersXSLFO" default="dist" basedir=".">
  <taskdef name="fop" classname="com.javageeks.ant.fop.FOPTask" />
  <taskdef name="xalan" classname="com.javageeks.ant.xalan.XalanTask" />
  <target name="prepare">
  </target>
  <target name="paper">
    <xalan xml="file:/C:/Projects/Tools/PapersXSLFO/whitePaper.xml" in="paper.xml"
out="paper.fo" />
    <fop fo="paper.fo" pdf="PapersXSLFO.pdf" />
  </target>
  <target name="code">
  </target>
  <target name="dist" depends="paper, code">
  </target>
</project>
```

build.xml

The two "taskdef" tags do the mapping necessary between my custom tasks (com.javageeks.ant.fop.FOPTask and com.javageeks.ant.xalan.XalanTask) and the corresponding Ant tags I use ("fop" and "xalan", respectively). Once that's done, I do the usual thing in Ant--set up some tasks, along with dependencies between them, with some dependent actions to run. In this case, I've set up a "prepare" task that does nothing, a "paper" task that runs that runs the XSLT and FOP steps, a "code" task to compile any code that might go along with the paper, and a "dist" task to create the distribution files¹².

At this point, then, it becomes trivial for me to regenerate the paper after modifying the source--I simply run "ant" at the command-line ¹³, and after the drive stops spinning, I have a nicely-formatted .PDF file for people to consume.

Consequences

This system has a number of advantages over writing papers directly in an authoring environment like MSWord:

Dynamism. Quite possibly the best advantage of this approach is its extremely dynamic nature; even as I was writing this paper, I found a few features that I wanted to add into the presentation and content capture mechanisms (for example, I suddenly decided I wanted to have captions on the code fragments, like the one displaying the Xalan Ant task, above), and it took all of about five minutes to add that support. A simple modification to the whitePaper.xsl file, based on some FO settings (and an occasional glance back to some of the other XSL:FO encodings I'd used in other places), and I suddenly had caption support for the <codeFrag> element. (In all honesty, it probably took less time for me to add the caption support than it would have for me to wander through the MSWord docs to figure out how to capture this in some kind of style setting in Word.)

Standardization. Even better, should I ever decide that I'd like to change the formatting around the paper (rather than the content), I only need to change the whitePaper.xsl file. Any changes that have been made to the XSL file will be picked up as soon as I rebuild the various papers, and it requires nothing more on my part than a simple command-line execution. What's more, because it's all integrated in with Ant, I can create a site-wide build script to rebuild all the papers at once, so changing the formatting or fonts or colors (or all of the above) become a trivial undertaking.

On top of that, along those same lines, I can put standardized sections into the papers themselves; for example, each paper now includes a Copyright section (which they did before, because I cut-and-paste one into each one) as well as a Colophon section describing how this paper was produced. As an added bonus, each Colophon includes the revision number of not only the paper content itself, but also the version of whitePaper.xsl used to produce the PDF file, just for my own crude form of quality control.

Flexibility. Hand in hand with standardization, this system allows me to render to multiple formats should I really desire to--for example, instead of rendering to FO, I could render the paper to HTML for easier viewing directly on the website. Once again, because the paper content is captured in XML format, I'd only have to write the XSLT once to transform to HTML, then re-render each paper into HTML. (Once I'd written that XSLT to HTML, however, I'd have two XSLT scripts to keep current with one another, and that essentially implies dual codebases, which is always a pain. This is the bane of transformation systems--keeping each endpoint of the transformation in sync with the other transformations.)

Revision control. Because the entire system is based on simple text-based formats and languages, everything goes under revision control (CVS, in my case), and can be supported wholly by the tools there. I can do "diff"s against the content or the XSL files to determine what was changed from one revision to another, deltas between revisions can be stored (as opposed to a complete copy, which was what was required to store the MSWord binary formats from revision to revision), and if necessary, I can even do remote editing via a simple telnet link if I have to fix a particularly nasty misstatement or XSL bug. Even more importantly, the formatting is kept under revision control--so if I change "look and feel", and later decide I hate it, a rollback followed by a rebuild is all I need to do to throw the new L&F away.

Life on the Bleeding-edge. Getting things to render the way I want them to in FO has been difficult, to say the least. Some of the simplest things for a word processor to do for me (footnotes being the glaring example) have been impossible for me to duplicate within FO. This may be a weakness with XSL:FO as a whole, or just the Apache FOP engine in particular, or just my own fumbblings with the XSL:FO documentation, but it's what forced me to switch away from footnotes to endnotes (such as used in this paper). This is simply indicative of what happens when one starts using technologies right away--not only is there little documentation, but the products may not have all the features you'd want or expect them to have.

Lack of immediate WYSIWYG. One of the principal advantages of an authoring environment like MSWord is the ability to see exactly what will be rendered; in fact, there's no rendering step at all--what you type is what will show up on the page or screen. (For those too young to remember, WYSIWYG, or "What You See Is What You Get", was all the rage as a buzzword about ten years ago, when word processors of the day were essentially character-based user interfaces. Once we moved to GUIs, WYSIWYG became standard and expected.) If I need to see exactly how something will render, I need

to go through a transform-render cycle, similar to the compile-link cycle developers are familiar with, to see what comes out. For developers, this is pretty much familiar ground, but non-developers may find it awkward to realize that what they type doesn't translate directly to what's on the screen. Since this system is geared specifically to developers (that is, me), this isn't much of an issue, but it could be if I ever adopted this system to work for authoring in other environments or for other target audiences.

Centralization. One facet I've not yet started to explore is the notion of centralization of content--creating a centralized content database from which the papers are generated. While the paper topics wouldn't necessarily be applicable for a centralized repository, certainly the bibliography sections would be, with each paper simply pulling out those bibliographic references it cites. That would save me from having to cut&paste (or worse, re-typing) each bibliographic reference for each paper.

Summary

Obviously, I'm happy with what's been produced thus far, particularly since it answers a particular need that I personally ran into--this is the essence of the "developer's itch" that Eric Raymond refers to in his "Cathedral and the Bazaar" paper on Open Source. Writing prose in XML has definitely freed me from having to worry about formatting and/or the intricacies of MSWord, which is a bonus, as far as I'm concerned.

One thing I've just begun to start to do is explore what happens next: Now that I've got reams of XML¹⁴, what do I do next? If I just leave it here, I've simply traded one authoring environment for another one. Two tasks leap immediately to mind: first, the generation of the .html file for the paper itself, and the .html file for all of the papers in general. Look for those to be generated automatically as part of the system sometime in the near future.

Notes

- [1] A company of ultimate coolness and technology for whom I am privileged enough to work and teach.
- [2] This is the same spirit that led C++ programmers to write their own String class over and over again, and Java programmers to write their own HTTP access libraries, and XML guys to write their own parsers, and...
- [3] Have a look at this PDF file in a text editor--PDF was essentially the successor to Adobe's PostScript format, except that Adobe lost control of PostScript and regretted it for years afterward. They've held onto control over PDF quite tightly since it's inception.
- [4] Certainly, nothing stops a reader from capturing the result and saving it off under some local name, but users are less likely to save something off if they believe or know that it is a dynamically-generated process--why capture something that is constantly changing?
- [5] Should either of those assumptions change, then I'd need revisit this decision; XSDs would ensure that all parties understood which version of the input format was being used for which paper, and so on. Certainly this would need to be in place for a "production" system.
- [6] For those unfamiliar with CVS, every time this file is checked in, the new version number is appended to the text of the string, thus placing the actual version number directly in the text
- [7] Come to think of it, the only author writing papers for javageeks.com is me, but I hope to change that sometime in the near future. Besides, it seems a bit vain to make your bio information part of the presentation rendering rather than the content.
- [8] If I ever hook the papers up to a search engine, that is. For now, they are simply echoed back onto the paper's home page on the website.
- [9] Annoyingly, I haven't found a way yet to get it to honor blank lines--it still compresses those down.
- [10] FOP has the ability to add new fonts within it, even TrueType ones--check out the Apache FOP website for details
- [11] On that note, if anyone who's an XML guru happens to know how to accomplish that within Xalan, please drop me a note.
- [12] In other papers, the "code" task would compile the associated sample code, and the "dist" task would then take the compiled code and create a .jar archive of the code for downloading by readers of the paper.
- [13] A script file kicks off the necessary Java commands to run Ant at that point--see the Ant

documentation for details.

[14] I'm slowly converting the "old" MSWord-format papers over to XML, but it's a tedious process.

Bibliography

- **[exml]** *Essential XML (2nd Edition)*, by Don Box and Aaron Skonnard. Addison-Wesley (ISBN: No clue)
- **[xmlcomp]** *The XML Companion*, by Neil Bradley. No clue (ISBN: No clue)

Copyright

This paper, and accompanying source code, is copyright © 2001 by Ted Neward. All rights reserved. Usage for any other purpose than personal or non-commercial education is expressly prohibited without written consent. Code is copywrit under the Lesser GNU Public License (LGPL). For questions or concerns, contact author.

Colophon

This paper was written using a standard text editor, captured in a custom XML format, and rendered to PDF using the Apache Xalan XSLT engine and the Apache FOP-0.17 XSL:FO engine. For information on the process, contact the author at tneward@javageeks.com. Revision \$Revision: 1.1 \$, using whitePaper.xml revision \$Revision: 1.1 \$.