MULTIPLE FORMAT DYNAMIC DOCUMENT GENERATION OF PROJECT GUTENBERG TEXTS

by
Brandon Dean Perkins

A Master's paper submitted to the faculty
of the School of Information and Library Science
of the University of North Carolina at Chapel Hill
in partial fulfillment of the requirements
for the degree of Master of Science in
Information Science.

Chapel Hill, North Carolina

April, 2003

Approved by:

_____

Advisor

Brandon Dean Perkins. Multiple Format Dynamic Document Generation of Project Gutenberg Texts. A Master's paper for the M.S. in I.S. degree. April, 2003. 28 pages. Advisor: Gregory B. Newby.

Project Gutenberg consists of over seven thousand eBooks of various and inconsistent file formats. While most books are accessible to all through ASCII format, this not necessarily the preferred format for all users of Project Gutenberg texts. The proposed solution to this problem consists of multiple parts to create a comprehensive package that can be implemented in a production environment. This project demonstrates how converting Project Gutenberg eBooks to XML (eXtensible Markup Language), using a standard DTD (Document Type Definition) or XML Schema, creates the opportunity for all available texts to automatically become available in multiple formats by applying stylesheets to the XML. These formats can include, but are not limited to, HTML (HyperText Markup Language), plain text, or PDF (Portable Document Format). This should provide a framework for future Project Gutenberg collection development.

Headings:

XML (eXtensible Markup Language)

DTD (Document Type Definition)

XSL (Extensible Stylesheet Language)

XSLT (XSL Transformations)

XSL-FO (XSL Formatting Objects)

Digital Libraries

Multiple Format Dynamic Document Generation of Project Gutenberg Texts

Project Gutenberg is an over thirty-year-old attempt to bring information to the general public using computer technologies. Much has changed over the years in computer technology, and the project has come to a crossroads where a large amount of books and other materials have become available, but in many different file formats with no common sense of structure. The other major need of Gutenberg is to make its materials available in the format that is most useful to the consumers of the information. The goal of this project is to demonstrate how formatting Project Gutenberg texts in XML (eXtensible Markup Language) using a common DTD (Document Type Definition) will allow for easy dynamic document generation into multiple formats. Once in this common XML format, any future document formats can easily be applied such that the demands of users can easily be met with a minimum amount of development time.

This paper begins by introducing the fundamentals of Project Gutenberg, XML, DTDs, TEI Lite, XSLT, and XSL stylesheets for TEI XML. The project then goes on to describe how to create and configure an environment capable of taking the family of XML standards and making them generate the output we are looking for. The next step described is the process of actually converting Gutenberg plain text documents into XML instances that conform to the TEI Lite DTD. Following that, the XSL stylesheets for TEI XML need to be modified to work in this environment. And finally, the combination of open source tools, XML Gutenberg documents, and XSLT stylesheets are all used together to dynamically generate HTML, PDF, and plain text versions of the eBooks that have been marked up.

Introduction to Project Gutenberg

The goal of Project Gutenberg is to make all types of information available to the world at large, using electronic resources. The most common type of information is books themselves, but any

other document of significance to the public has the potential for inclusion once criteria are met. The idea is fundamentally simple, to provide easy access to interested parties who want the ability to search, read, and quote historical documents that may or may not be readily available through other avenues.

The ramifications of all of this include:

1. The Project Gutenberg Etexts should cost so little that no one will really care how much they cost. They should be a general size that fits on the standard media of the time.
2. The Project Gutenberg Etexts should so easily used that no one should ever have to care about how to use, read, quote and search them.

(Project Gutenberg, 1992)

The first ramification is much less of an issue today as it was even five years ago. The documents are expensive in people hours. The time necessary to scan, proofread, and make these texts available requires a large amount of time and effort on the part of countless volunteers. As for the size constraint, this too is not as much of an issue as it once was as processing, storage, and network resources are more plentiful and will continue to be in the future. With processor costs dropping, storage size increasing, and broadband and large bandwidth becoming more the norm than the exception, this will make this part of Project Gutenberg accessibility trivial.

The second ramification is the main issue being addressed in this project. Project Gutenberg Ebooks are not nearly as easy to use as hoped. Although accessible through being typically made available through ASCII encoded text files, this is not necessarily the easiest way to read and search the vast Project Gutenberg collection.

The Official Project Gutenberg Web site may be found at http://www.gutenberg.net/. The main eBook distribution site is hosted by ibiblio, the Public's Library and Digital Archive located at the University of North Carolina at Chapel Hill, at: http://www.ibiblio.org/gutenberg/.

Introduction to XML

XML is an acronym for eXtensible Markup Language. It is an over five year old specification developed by the W3C (World Wide Web Consortium). XML is a pared-down version of SGML (Standard Generalized Markup Language), designed especially for publishing documents on the Web and for managing information internal to an enterprise or organization. It allows designers to create their own customized tags, or markup, enabling the definition, transmission, validation, transformation, and interpretation of data between applications and between organizations.

Project Gutenberg texts lend themselves perfectly for markup in XML. Many DTDs and XML Schemas have been developed throughout the years, specifically geared towards marking up books, plays, poetry, and more. Much the same way that ASCII text is an encoding that makes these texts readily available, XML is the first real markup language that can make these texts available to the general population and actually has support for many more character encodings than plain text can typically provide. SGML, being the direct ancestor with many more years of maturity, would also be a very likely candidate markup language for Gutenberg texts. However, SGML's complexity combined with XML's fundamental underpinnings as a Web-based language leads markup preference away from SGML and towards XML.

The XML file itself can be considered the "middle layer" of the total XML-based document management system. Underlying the XML is the DTD or XML schema that describes the structure, or grammar, of the XML document. Besides just being a framework for the XML document, it also allows for the document to be validated to make sure that it conforms to the rules set in the syntax. Having this validation will assure proper document rendering and generation in the transformation stage of the system. The "top layer" is what the end user or application is presented with. The stylesheets used in the transformation exist to create a new document based on the XML document itself. The resulting document could be an alternate XML

file or any other type of file format imaginable, such as HTML, XHTML, PDF, RTF, Braille, text, etc.

This three-layer system separates the concerns of logic, content, and style. Logic is the concern of application developers. Content is the concern of the creative individuals that produce the works that end users are concerned with. In the case of this project, the Gutenberg texts, written by some of the finest authors in history, are the content providers, and we are simply marking up their works into XML. And style is the concern of Web designers and those responsible for passing this data to another node. As long as the "contracts" (agreements between groups on how the data looks from one stage to the next) do not change, these individual concerns can be developed autonomously. This is what is meant when XML emphasizes the separation of content and presentation such that all parties can work as efficiently as possible.

Introduction to DTD

DTD is an acronym for Document Type Definition. A DTD describes what elements (often referred to as tags or markup) and attributes (element modifiers) are used to describe content in an SGML, XML or HTML document, the allowed order of each element, and which elements can appear within other elements. For example, in a DTD a developer could create a BOOK element that can contain CHAPTER elements, but CHAPTER elements cannot contain BOOK elements. In some DTD and Schema aware editors, when content authors are inputting XML data, they can place elements only where the DTD allows. This ensures that all the documentation is well-formed and valid. Applications can then use the XML document's DTD to properly read and validate a document's contents. Changes in the format of the document can be easily made by modifying the DTD.

At this point, a couple of things should be pointed out. So far, any reference to the document that defines the structure of an XML document has been referred to as the "DTD or Schema". The reason for this is that a DTD is not technically a Schema in the XML sense, and therefore they

cannot be lumped together. There are quite a number of Schemas available for XML; some of the more popular ones include XSDL (XML Schema), SOX (Schema for Object-Oriented XML), and XDR (XML-Data Reduced schema). All of these are valid schema languages and they all have tools available for editing, reading, and validating the XML based on them.

The second point that has to be made is that this project had a choice regarding what XML grammar language to use. Although each of the schemas mentioned have a much richer vocabulary for describing structure and content than DTD, all of them are much newer languages and do not have the wide-spread adoption that DTDs currently have. In the near future, it is expected that XML Schema from the W3C will surpass DTDs as the schema of choice for XML documents, but until then, the support for DTD is simply overwhelming.

Introduction to TEI Lite

This project does not attempt to create a new DTD to have Gutenberg texts conform to. A University of North Carolina at Chapel Hill School of Information and Library Science Master's Paper by Cynthia L. Blue explored the issue of creating an XML DTD for Project Gutenberg (Blue, 2001). Ultimately, the project found that the alternatives available at the time were not good choices as the structure for Project Gutenberg documents and consequently created a DTD from scratch. Not many changes have occurred in the last two years as to the likely schemas that could be used to markup Gutenberg texts. While still taking this into account, it was the opinion in this project to choose another likely candidate that is supported and continues to be developed. With these criteria in mind, the TEI Lite DTD was chosen as the grammar to use in this project. The choice of TEI Lite does not indicate that it is necessarily the best choice for production use, however it has a comprehensiveness that lends itself well to the entire XML through presentation process. Even if another DTD is chosen, the project still exposes the underpinnings of creating a dynamic multiple format document generation system for Project Gutenberg.

TEI is an acronym for Text Encoding Initiative. The guidelines "emphasize the interchange of textual information, but other forms of information such as images and sound are also addressed. The Guidelines are equally applicable in the creation of new resources and in the interchange of existing ones" (Text Encoding Initiative, 2002). TEI Lite is a subset of the full TEI. The Website for the TEI Lite specification can be found at: http://www.tei-c.org/Lite/. The whole TEI is very large and overkill for most tasks, including the markup of Project Gutenberg. To get the best results out of the TEI, it needs to be customized to suit the requirements of the particular task. Customization requires a lot of knowledge of the entire TEI, if only in order to know what is unnecessary for a particular task. TEI Lite is a specific customization designed for the majority of TEI markup requirements. This subset has been proven to be very popular. Both the XML and SGML DTDs can be found at: http://www.tei-c.org/Lite/DTD/. TEI Lite contains all of the structural components necessary to create most any type of book, play, poetry, or other document available as a Gutenberg eText.

Introduction to XSLT

XSLT is an acronym for eXtensible Style Language Transformation and is the language used in XSL (eXtensible Style Language, a specification for separating style from content when creating HTML or XML documents) stylesheets to transform XML documents into other XML documents. An XSL engine parses the XML document and follows the transformation rules in the XSL stylesheet, then it outputs a new XML document, XML-document fragment, HTML, or any other format that the XSLT was programmed to output. The XSLT Recommendation was written and developed by the XSL Working Group and became ratified by the W3C in 1999. XSLT is purely concerned with presentation without regard for the content of the XML document itself. Typically, the DTD and XSLT are initially developed in conjunction so that structural elements are directly mapped to presentational components. This separation creates a plug-in type architecture, where any XML document that conforms to the DTD rules can be dropped in and get consistent document formatting through XSLT.

XSLT is actually written in XML itself and provides an easy mechanism for transforming to any other markup language. And through the use of XSL-FO, PDF (Portable Document Format) files can be created when using an XSLT engine that uses these formatting objects.

Introduction to XSL stylesheets for TEI XML

Because the TEI DTD is so mature and widely used, it makes sense that development has already been done to create stylesheets to transform the XML data structured against TEI. To this end, there is the XSL stylesheets for TEI XML project, which can be found at: http://www.tei-c.org/Stylesheets/teixsl.html. The XSL stylesheets for TEI XML are "a set of XSLT specifications to transform TEI XML documents to HTML, and to XSL Formatting Objects. It concentrates on TEI Lite" (Text Encoding Initiative, 2002). The stylesheets to produce the HTML and PDF are very modularized with a main top-level wrapper file that includes all of the sub-level module files that have the real processing directives. These modules include instructions for Parameterization, Basic TEI structure, Front matter, Divisions, Figures, Paragraph-level elements, Tables, Cross-referencing, Lists, Plays, Making frames, Making table-formatted pages, and making slides.

The TEI XSL stylesheets only include HTML and PDF output. Formats such as text, Braille, and RTF have not been developed. As text especially was of great concern for this project, a new TEI XSL stylesheet needed to be written from scratch to perform the operations necessary. The stylesheet that was written is more of a proof of concept than an exhaustive stylesheet to take all of the elements, attributes, and parameters addressed in the TEI XSL stylesheets that cover the majority of TEI Lite.

Environment for Dynamically Generating Documents

Investigating the server-side XML transformation options available was an interesting task. There are not a lot of options available, but the options that are around are robust and proven to work. The solutions seem to fall into three categories. The first is using a server-side scripting language such as PHP, ASP, or Perl to perform the transformations. This requires a great deal of custom

code and great understanding of both the family of XML specifications and the language being coded in. The second is using custom server-side Java code, commonly known as a Java servlet. This is a very proven method of development, however this case requires a lot of knowledge and time to use the APIs to develop an application that can fully extend the functionality of XSLT. The third option is using an entirely integrated system intended specifically for document storage and processing. One of the products in this third category is Cocoon, developed within the Apache XML Project (part of The Apache Software Foundation). After much consideration and testing, it was found that Cocoon provided the best environment for dynamic XML document transformations.

The advantages of using Cocoon are numerous. First, it is an active Open Source project, meaning that support is readily available, bug fixes are quickly applied, and new features are added that meet user and developer demands. Second, it builds upon an entire stack of other Open Source projects, but can optionally be used within a proprietary environment. By default Cocoon leverages various tools from the Apache Jakarta Project, Xalan as the XSLT processor that fully supports the W3C specs, and Xerces as the XML parser just to name a few. It requires a Servlet 2.2 compliant servlet engine to operate in, and luckily the Jakarta project develops Tomcat, which is the servlet container that is used in the official Reference Implementation for the Java Servlet and JavaServer Pages technologies. The one final technology needed to make the whole package run is a Java 1.2 or later compatible virtual machine that must be present for servlet usage of Apache Cocoon.

Following are the steps used to create the servlet engine environment needed for the Gutenberg XML document transformations:

1. Find a machine that is capable of using a Java 2 Runtime environment (J2RE). Java is a high-level programming language developed by Sun Microsystems. Sun itself develops and distributes runtime environments for four platforms, Microsoft Windows, Linux,

Solaris on the SPARC architecture, and Solaris on the x86 architecture. Other vendors (such as Apple, Hewlett-Packard, and IBM) develop runtime environments for different and some of the same platforms. In all cases, a Java 2 Runtime environment is the only key for the remainder of the servlet stack. In the case of this project, development was done on a Sun Ultra 80 machine running Solaris 8 as the operating system.

2. Download the appropriate J2RE for the platform. At the time of the project, the latest stable J2RE available was 1.4.1_02 found at: http://java.sun.com/j2se/1.4.1/download.html. I chose to download the Solaris SPARC 32-bit self-extracting file named "j2sdk-1_4_1_02-solaris-sparc.sh".
3. Install the J2RE on the system. First, I created a directory that would contain all the necessary parts for this project, and I named it "Gutenberg". Second, I changed into that directory to extract the J2RE. The self-extracting archive will extract the files into the current directory that the user is in. Next, we simply needed to run the "sh" command, followed by the full path and filename of the archive. In this case, I ran the following command: sh ../cocoon/j2sdk-1_4_1_02-solaris-sparc.sh. Once the script starts, we need to accept the terms of the Binary Code License Agreement. After that, the archive begins unpacking, checksumming, and extracting the J2RE. After this, the installation is done, the new J2RE is available in the "j2sdk1.4.1_02" directory, and we can move on to the next step.
4. Download the Servlet 2.2 compliant servlet engine. This piece must be present in order to support servlet operation and the dynamic request handling we need. The choice here was to use the Tomcat servlet container developed within the Jakarta project of the Apache Software Foundation. At the time of this project, Jakarta Tomcat version 4.1.18 was the latest release available from: http://jakarta.apache.org/tomcat/. The tar/gzip package that was downloaded is not platform specific and is named: tomcat-4.1.18.tar.gz.
5. Install the Servlet Engine on the system. This is a fairly easy task. Still from within the "Gutenberg" directory, simply gunzip and run a tar extraction on the downloaded archive.

The command used in this project was: gunzip -cd ../cocoon/tomcat-4.1.18.tar.gz | tar -xvf -. This command creates the "jakarta-tomcat-4.1.18" directory that contains the entire servlet engine.

6. Configure the Servlet Engine to run on the system. This means setting the "JAVA_HOME" environment variable in the "jakarta-tomcat-4.1.18/bin/catalina.sh" script. On line 38 of the script, the following two lines should be added:

   JAVA_HOME=/home/brandon/Gutenberg/j2sdk1.4.1_02

   export JAVA_HOME

   This gives Tomcat startup and shutdown scripts the information it needs for invoking the Java virtual machine.

7. Test to see if Tomcat is running. Still, from within the "Gutenberg" directory, the following command was typed: "jakarta-tomcat-4.1.18/bin/startup.sh". And the resulting output was as follows:

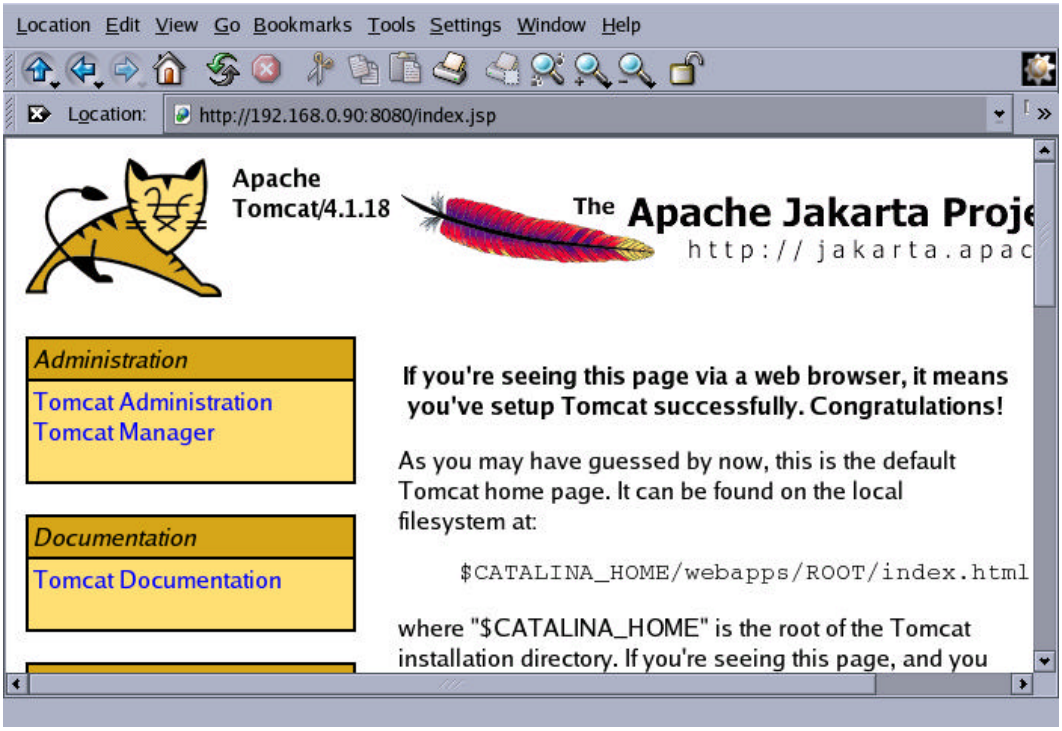
   Using CATALINA_BASE: /home/brandon/Gutenberg/jakarta-tomcat-4.1.18

   Using CATALINA_HOME: /home/brandon/Gutenberg/jakarta-tomcat-4.1.18

   Using CATALINA_TMPDIR: /home/brandon/Gutenberg/jakarta-tomcat-4.1.18/temp

   Using JAVA_HOME: /home/brandon/Gutenberg/j2sdk1.4.1_02

   This would indicate that the Tomcat servlet engine is running, but the best way to test this is to point a Web browser at the server on port 8080 to verify that it is running correctly:



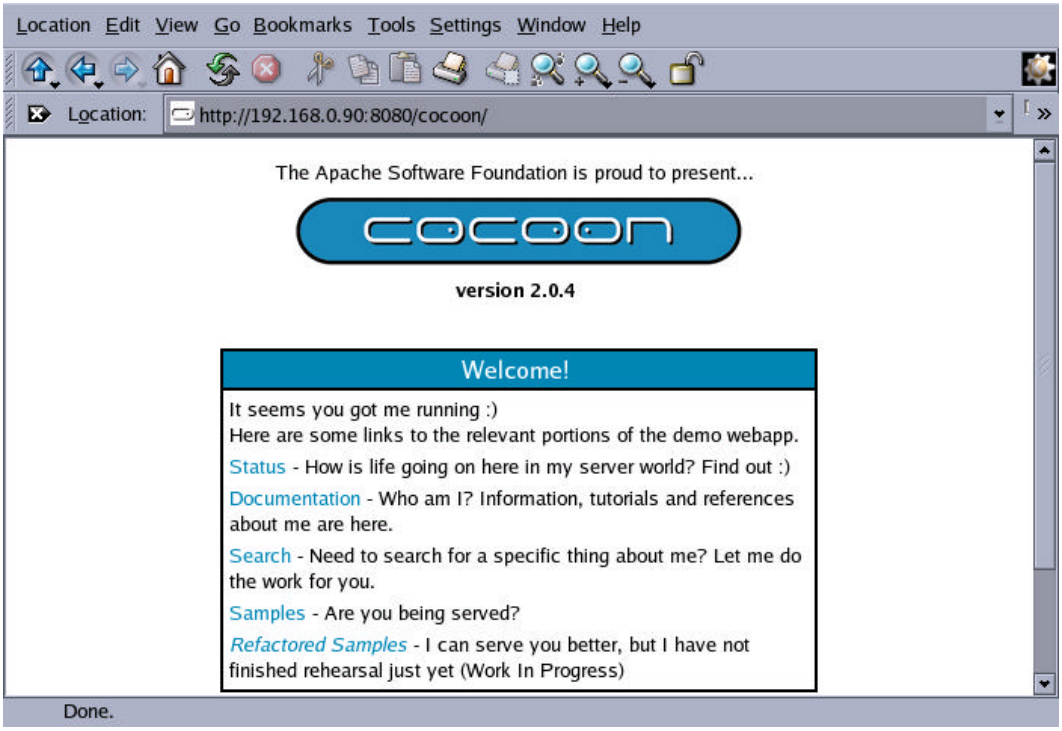
At this point it appears that Tomcat has been installed and setup successfully.

8. Download Cocoon. The third, and final piece of the transformation stack is the servlet itself, which is run inside of Tomcat. Stable versions of Cocoon can be downloaded from: http://xml.apache.org/cocoon/dist/. At the time of this project, the latest version of Cocoon was 2.0.4. Cocoon includes all the packages required to run out of the box with included Xerces, Xalan and FOP packages. This means that we do not need to download anything else to start. Also, due to some incompatibilities between JDK 1.3 and JDK 1.4, we need have to choose between a binary version targeted for JDK 1.2/1.3 and a version specially targeted for JDK 1.4. As we chose a 1.4 JVM, we need to download the 2.0.4 for 1.4 package, which is named "cocoon-2.0.4-vm14-bin.tar.gz".
9. Install the Cocoon Servlet on the system. The fist step is simply to extract the Cocoon archive. This was done using the following command: gunzip -cd ../cocoon/cocoon-2.0.4-vm14-bin.tar.gz | tar -xvf -. This extraction process creates a new directory within the "Gutenberg" directory called "cocoon-2.0.4". There are a lot of samples and documentation in this package, but there is only one file that is truly needed to get the servlet running. Within Tomcat, this is just a matter of copying the Cocoon war file into a

specific directory and then the servlet engine will take care of installing Cocoon when restarted. So, the next step is to copy the cocoon/build/cocoon/cocoon.war file into the tomcat/webapps directory. The specific command, from the "Gutenberg" directory, used in this case was: .cp cocoon-2.0.4/cocoon.war jakarta-tomcat-4.1.18/webapps/.

10. Restart Tomcat to make Cocoon run. This is a simple two-step process to perform. First, the shutdown script is run: .jakarta-tomcat-4.1.18/bin/shutdown.sh. And then the startup script is run: jakarta-tomcat-4.1.18/bin/startup.sh. The testing done in step seven should be performed to make sure Tomcat is up and running again.
11. Test to see if Cocoon is running. To do this, we point a Web browser at the server on port 8080 and the cocoon directory (http://192.168.0.90:8080/cocoon/ in this case) to verify that it is running

    correctly:



    At this point it appears that Cocoon has been installed and setup correctly. Further tests for basic functionality can be performed by following the links from this main page.

At this point, the server/servlet environment has been setup to allow for dynamic multiple-format document generation of XML marked-up Gutenberg texts.

Converting Project Gutenberg Texts to XML

The task of converting Project Gutenberg Texts to XML could easily be researched in another Master's Project, or a distributed effort to migrate the thousands of eBooks into a consistent XML format. Some of this work has already begun with the PGXML (Project Gutenberg XML) project, which is a movement to use XML as the base format for Project Gutenberg eTexts. This project is looking to create a Gutenberg specific DTD based on an existing XML DTD. It is also working to create software that makes it easy to convert and create Gutenberg texts into a standard XML format.

But as these tools and DTD do not exist yet, it was a requirement of this project to be able to create sample XML files from Gutenberg text files.

To do this, I created a very simple Perl script that parses a plain text ASCII Gutenberg document, and outputs a basic XML document conforming to the TEI Lite DTD. This script, called "pgtxt2xml.pl", can be found in Appendix A. Some further hand editing was required on all of the text documents as the data is not consistent in the ASCII file and requires human analysis, or significantly more code, to find the data that goes into certain fields, such as author and date. This procedure does remove the majority of the work required to markup a Gutenberg text as hand editing is time consuming and error-prone. The basic format of the XML that conforms to the TEI Lite DTD can be found in Appendix B.

Modifying and Creating XSLT stylesheets for TEI Lite

As mentioned earlier, there has been a great amount of work on creating a set of XSL stylesheets based on the TEI Lite DTD. These stylesheets provide the framework for generating HTML and PDF output of Gutenberg texts. XSL stylesheets for TEI XML can be downloaded from the TEI

site at: http://www.tei-c.org.uk/Stylesheets/teixsl.html. To make these stylesheets available to the Cocoon servlet, we need to perform the following steps:

1. Download the stylesheets. Luckily both the HTML and PDF stylesheets are packaged in a zip file for easy download. They are available at: http://www.tei-c.org.uk/Stylesheets/teixsl-html.zip (for HTML) and http://www.tei-c.org.uk/Stylesheets/teixsl-fo.zip (for PDF).
2. Create a Gutenberg directory within the Cocoon mount area. The name can be anything, but in this case we will again use the directory name "Gutenberg".
3. Setup the new "Gutenberg" directory. To do this, we need to change into this directory and create three new directories, one for each format we will be generating. In this case, I chose to create the directories "teixsl-fo", "teixsl-html", "teixsl-txt". Then extract each of the XSL zip files into the corresponding directory created using unzip as: unzip <filename>.
4. Customize the parameters stylesheets with site-specific data. These files are called "teixsl-fo/tei-param.xsl" (for PDF) and "teixsl-html/teihtml-param.xsl" (for HTML). These files simply need to have text strings, email addresses, URLs, and images modified to incorporate the preferences, look, and feel of the Gutenberg site.
5. Modify the main HTML XSL stylesheet for use with Xalan. The file "teixsl-html/teihtml-main.xsl" needs to be modified to work with the Xalan transformation engine. The reason is that you we will get nothing on standard output by default, as Xalan uses the EXSLT extensions to XSLT to generate the top-level file by name. By adding the following three lines at line 208, we allow Xalan to produce HTML on-the-fly:

   ```
   <standard-out>
       <xsl:apply-templates/>
   </standard-out>
   ```
6. Create the plain text XSLT stylesheet. The XSLT stylesheet used in this project is not exhaustive, but does provide the framework for creating a more complete stylesheet. This

file, called "teitxt.xsl" should be placed in the "teixsl-txt" directory. This stylesheet can be found in Appendix C.

With all of these files place in their appropriate locations, we can now move on to actually generate HTML, PDF, and plain text versions of Gutenberg texts dynamically.

Creating Multiple Formats of Project Gutenberg Texts

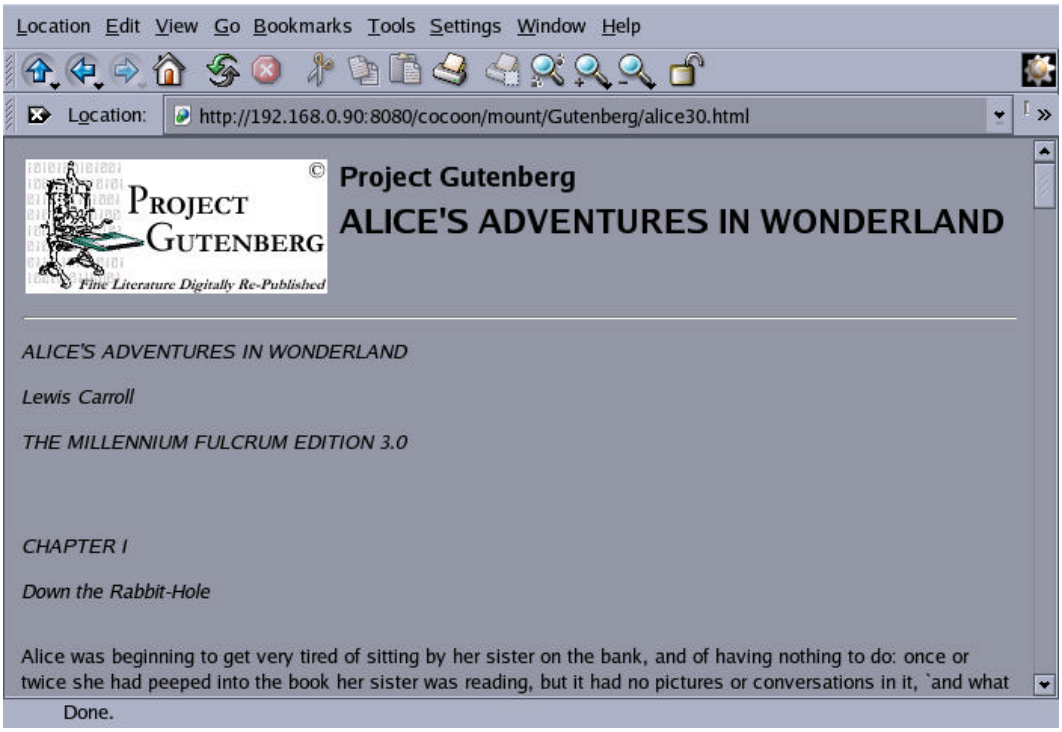
This is what the project is all about. Taking all the various technologies and bringing them together to actually create a human readable file based on one common file to all formats. There are only a few more steps remaining to have the final product:

1. Create the sitemap.xmap file. This file contains the processing instructions that should be performed in this directory. This file sits in the "Gutenberg" directory and specifies what should be done based on the URLs. In our case, the system will accept the file name in the URL to determine the XML file to use and what transformation to perform. For example, if we have a request for "example.html", Cocoon will use the file "example.xml" and use the transformation rules given for an HTML transformation. Same with "example.pdf" using the PDF transformation, and "example.txt" using the Plain Text transformation. The sitemap used in this project can be seen in Appendix D. The Cocoon documentation explains the use of this file in detail.
2. Copy the simple XML to HTML file into the "Gutenberg" directory for debugging purposes. The command to do this is: cp ../../stylesheets/simple-xml2html.xsl .. Besides being used for debugging, this also just provides a generally good presentation of an XML document.
3. Copy the Project Gutenberg XML eBooks into the "Gutenberg" directory. For this project. I used Alice's Adventures in Wonderland by Lewis Carroll, A Christmas Carol by Charles Dickens, and The Hunchback of Notre Dame by Victor Hugo. The XML file names were

"alice30.xml", "carol12.xml", and "hback10.xml" respectively. These names conform to their Project Gutenberg plain text file names.

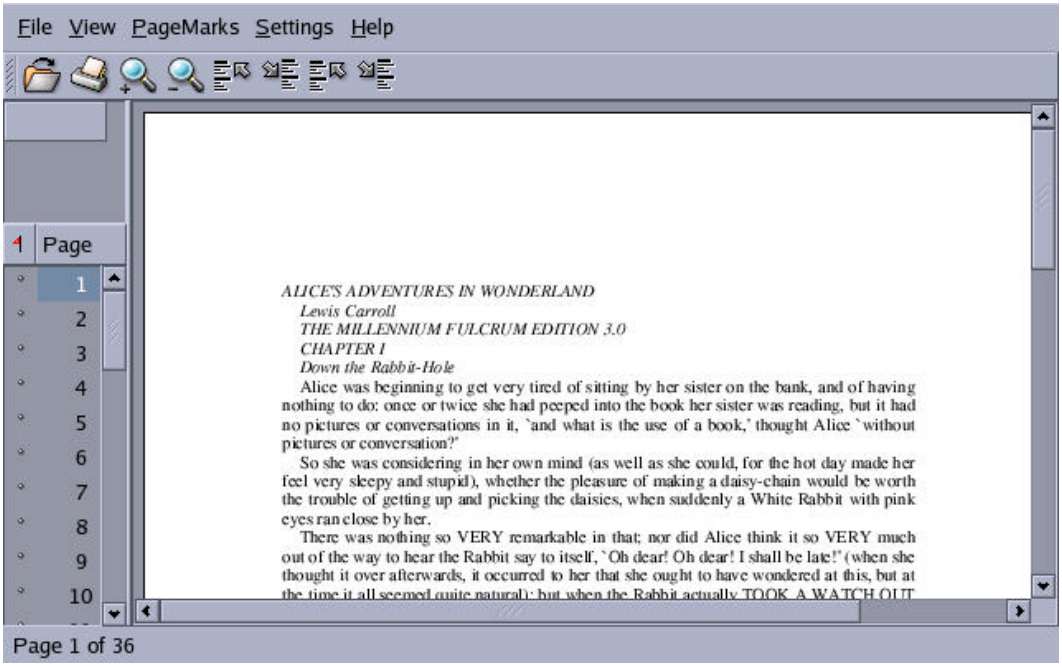
4. Test the HTML rendering of an eBook. To test the HTML version of Alice's Adventures in Wonderland we point a Web browser to the following URL:

   http://192.168.0.90:8080/cocoon/mount/Gutenberg/alice30.html:



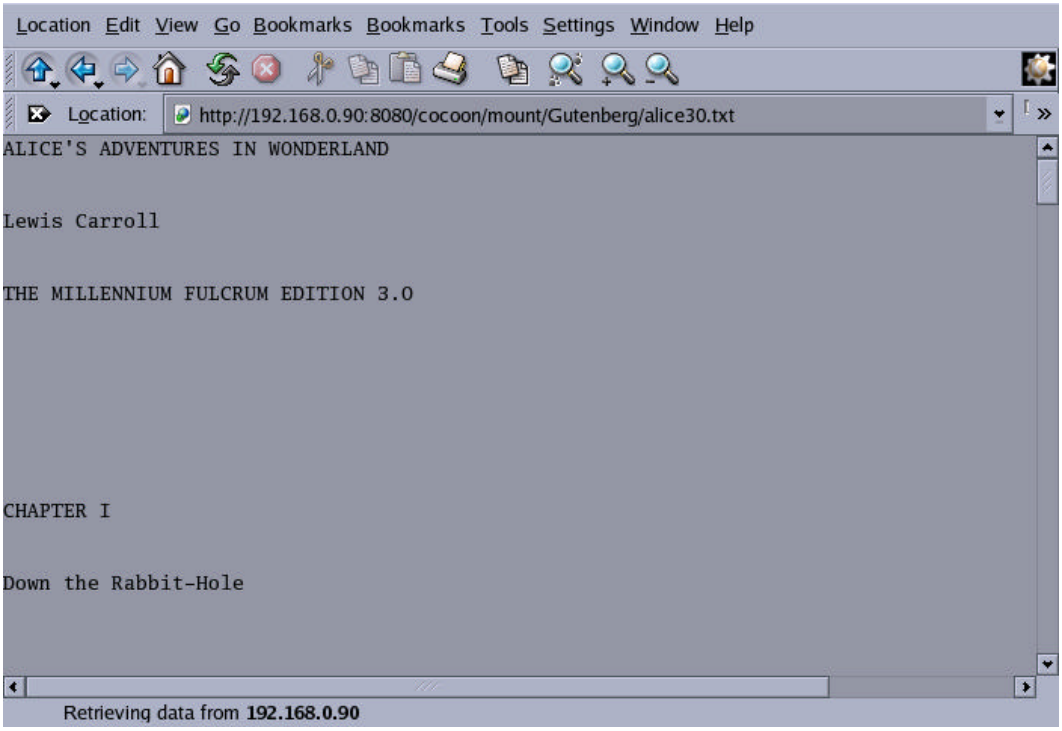
5. Test the PDF rendering of an eBook. To test the PDF version of Alice's Adventures in Wonderland we point a Web browser to the following URL:

http://192.168.0.90:8080/cocoon/mount/Gutenberg/alice30.pdf:



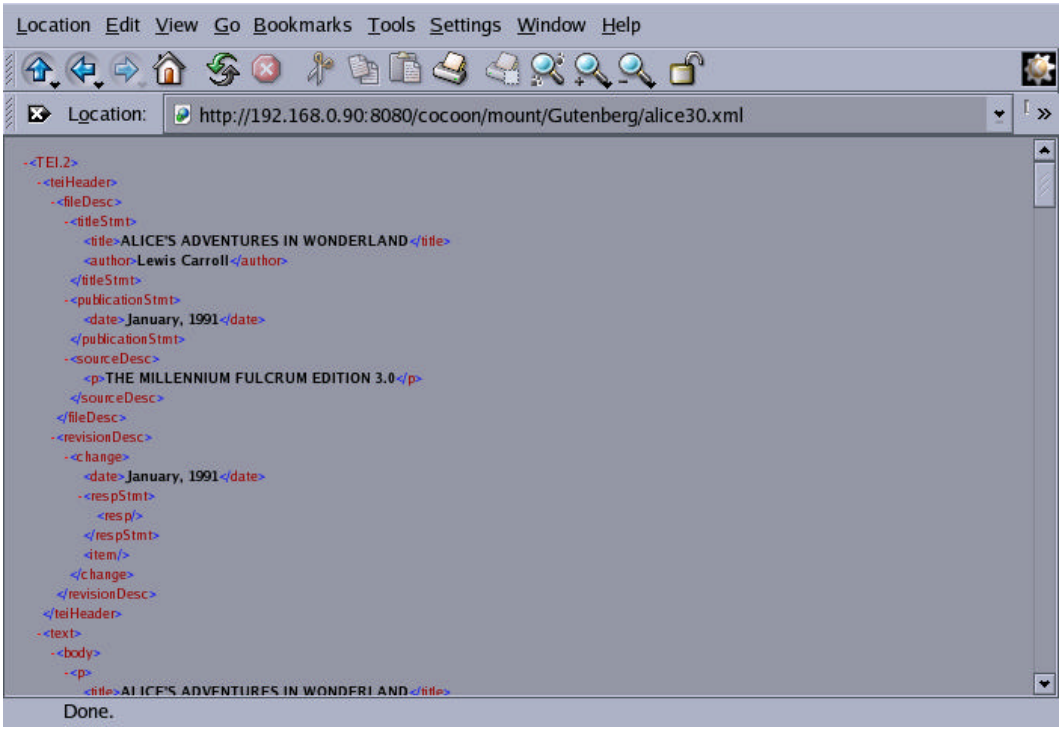
6. Test the plain text rendering of an eBook. To test the plain text version of Alice's Adventures in Wonderland we point a Web browser to the following URL:

http://192.168.0.90:8080/cocoon/mount/Gutenberg/alice30.txt:



7. Test the XML rendering of an eBook. To test the XML version of Alice's Adventures in Wonderland we point a Web browser to the following URL:

   http://192.168.0.90:8080/cocoon/mount/Gutenberg/alice30.xml:



In all, we can see all three file formats generated from a single XML source document. This provides evidence that this procedure can be implemented to create a full production system.

Conclusion

Making dynamic presentations of Gutenberg eBooks is not only doable, but a very valuable use of volunteer resources. An eText only needs to be marked-up once, and neither the submitters nor the proofreaders need to concern themselves with presentations or formats. It should also be pointed out, that this configuration does not limit one presentation per format. For example, a text stylesheet could have multiple column widths; a PDF stylesheet could have a choice between single column, multiple columns, and various page sizes based on locale or printing requirements; and an HTML stylesheet could have options for different colors and fonts. But in

each case, the underlying Project Gutenberg XML does not need to be modified. Stylesheets can be added, removed, or modified, and these changes will be reflected immediately once a user attempts to load a particular URL. The possibilities here are endless; the real challenge is the marking-up of the thousands of eTexts.

Project Gutenberg texts also contain a header, which consists primarily of the document's trademark statement. It is meant to explicitly declare the redistribution rights individuals and organizations have in using these eBooks for any purpose. One of the main advantages to the system described in this project is that this header can be imported into the final output without actually being included in the XML file itself. In this way, one header can apply to many documents, and a minor change in the header can be made once and applied to all documents in much the same way that a modification to a stylesheet needs only to be made once to appear on all documents.

This project researched the possibility of making this system available on ibiblio.org. Unfortunately, there were a few obstacles. I was unable to get a clear answer on the version of Tomcat being used, but the administrators did say they could set up a context for Project Gutenberg. In the ibiblio environment, they proxy the Tomcat traffic through an Apache HTTP server. Second, they are using IBM's J2SDK for Linux 1.3. Although not necessarily a problem, significantly more testing of this system has been performed using Sun's J2RE for Linux 1.4. And the final concern is that the Cocoon Servlet is not installed, and the administrators suggest using the installed Xalan and Xerces libraries to parse and transform XML from within Java. Although not necessarily a problem, it is the opinion of this project that a well-supported, open-source servlet solution is preferable to a large amount of custom Java code.

Besides the solution presented in this presentation, other options could include using XML and XSLT library modules that have been included in the PHP and Perl on ibiblio. Regardless of the transformation technology used, it is imperative that Project Gutenberg begins the process of

migrating all documents to a standard XML format, and hopefully pushes to have all new submissions put into XML in the first place.

The implementation of all of these various technologies to provide a valuable resource to the general public goes to show how standards-based and open-source solutions are capable of performing their intended tasks. And beyond just the ability to do the job, it also is cost effective and has broad-based support from the Internet development community. More development will need to be done, but this project should provide the groundwork for future development work. But hopefully, this is a glimpse into the future of Project Gutenberg.

References

Blue, Cynthia. (2001, April). An XML DTD for Project Gutenberg. [On-Line]. Available: http://ils.unc.edu/MSpapers/2656.pdf.

Project Gutenberg. (1992, August). HISTORY AND PHILOSOPHY OF PROJECT GUTENBERG. [On-Line]. Available: http://promo.net/pg/history.html.

Text Encoding Initiative. (2002, May). TEI U5: Encoding for Interchange: an introduction to the TEI. [On-Line]. Available: http://www.tei-c.org/Lite/.

Text Encoding Initiative. (2002, December). XSL stylesheets for TEI XML. [On-Line]. Available: http://www.tei-c.org/Stylesheets/teixsl.html.

Appendix A - pgtxt2xml.pl

```
#!/usr/bin/env perl

$infile="carol12.txt";
$outfile=$infile;
$outfile=~s/\.txt$/\.xml/;

unless(open(INFILE,"$infile")) {
    die "Could not open file $infile!\n";
}

if (-f $outfile) {
    unlink("$outfile");
}

unless(open(OUTFILE,">$outfile")) {
    die "Could not open file $outfile!\n";
}

print OUTFILE "<?xml version = \"1.0\" encoding = \"UTF-8\"?>\n";
print OUTFILE "<TEI.2>\n";
print OUTFILE "\t<teiHeader>\n";
print OUTFILE "\t\t<fileDesc>\n";
print OUTFILE "\t\t\t<titleStmt>\n";
print OUTFILE "\t\t\t\t<title/>\n";
print OUTFILE "\t\t\t\t<author/>\n";
print OUTFILE "\t\t\t</titleStmt>\n";
print OUTFILE "\t\t\t<publicationStmt>\n";
print OUTFILE "\t\t\t\t<date/>\n";
print OUTFILE "\t\t\t</publicationStmt>\n";
print OUTFILE "\t\t\t<sourceDesc>\n";
print OUTFILE "\t\t\t\t<p/>\n";
print OUTFILE "\t\t\t</sourceDesc>\n";
print OUTFILE "\t\t</fileDesc>\n";
print OUTFILE "\t</teiHeader>\n";
print OUTFILE "\t<text>\n";
print OUTFILE "\t\t<body>\n";
print OUTFILE "\t\t\t<p>\n";

while(<INFILE>) {
    chomp $_;
    $_=~s/\r//g;
    $_=~s/^$/<\/p><p>/;
    if($_ =~ m/^\s{3}/) {
        $_=~s/^\s+/<title>/;
        $_=~s/$/<\/title>/;
    }
    print OUTFILE "$_\n";
}

print OUTFILE "\t\t\t</p>\n";
print OUTFILE "\t\t</body>\n";
print OUTFILE "\t</text>\n";
print OUTFILE "</TEI.2>\n";

close(OUTFILE);
close(INFILE);

system("tidy -xml  --write-back yes -indent -quiet $outfile");
```

Appendix B - Example Basic XML Data

```
<?xml version="1.0" encoding="utf-8"?>
<TEI.2>
  <teiHeader>
    <fileDesc>
      <titleStmt>
        <title>Title</title>
        <author>Author</author>
      </titleStmt>
      <publicationStmt>
        <date />
      </publicationStmt>
      <sourceDesc>
        <p />
      </sourceDesc>
    </fileDesc>
    <revisionDesc>
      <change>
        <date>Date</date>
      </change>
    </revisionDesc>
  </teiHeader>
  <text>
    <body>
      <p><title>Title</title></p>
      <p>Text</p>
    </body>
  </text>
</TEI.2>
```

Appendix C - Plain Text XSLT Stylesheet

```
<?xml version = "1.0" encoding = "UTF-8"?>
<xsl:stylesheet version = "1.0" xmlns:xsl = "http://www.w3.org/1999/XSL/Transform">
        <xsl:param name = "maxwidth" select = "102"/>
        <!--xsl:param name = "maxwidth" select = "68"/-->
        <!--xsl:param name = "maxwidth" select = "34"/-->

        <xsl:output method = "text"/>
        <xsl:template match = "/">
                <xsl:apply-templates select = "/TEI.2/text/body/p"/>
        </xsl:template>
        <xsl:template match = "p">
                <xsl:apply-templates select = "title"/>
                <xsl:call-template name = "wrap-multiline">
                        <xsl:with-param   name   =   "text"   select   =   "normalize-
space(text())"/>
                </xsl:call-template>
                <xsl:text>&#xA;&#xA;</xsl:text>
        </xsl:template>
        <xsl:template match = "title">
                <xsl:call-template name = "wrap-multiline">
                        <xsl:with-param   name   =   "text"   select   =   "normalize-
space(text())"/>
                </xsl:call-template>
                <xsl:text>&#xA;</xsl:text>
        </xsl:template>
        <xsl:template name = "wrap-multiline">
                <xsl:param name = "text"/>
                <xsl:param name = "text" select = "normalize-space($text)"/>
                <xsl:choose>
                        <xsl:when test = "contains($text,'&#10;')">
                                <xsl:call-template name = "justify">
                                        <xsl:with-param   name   =   "width"   select   =
"$maxwidth"/>
                                        <xsl:with-param name = "txt" select = "substring-
before($text,'&#10;')"/>
                                </xsl:call-template>
                                <xsl:text>&#10;</xsl:text>
                                <xsl:call-template name = "wrap-multiline">
                                        <xsl:with-param   name   =   "text"   select   =
"substring-after($text,'&#10;')"/>
                                </xsl:call-template>
                        </xsl:when>
                        <xsl:when test = "$text">
                                <xsl:call-template name = "justify">
                                        <xsl:with-param   name   =   "width"   select   =
"$maxwidth"/>
                                         <xsl:with-param name = "txt" select = "$text"/>
                                </xsl:call-template>
                        </xsl:when>
                </xsl:choose>
        </xsl:template>
        <xsl:template name = "justify">
                <xsl:param name = "txt"/>
                <xsl:param name = "width"/>
                <xsl:choose>
                        <xsl:when test = "$width &lt; string-length($txt)">
                                <xsl:variable name = "real-width">
                                        <xsl:call-template name = "tune-width">
                                                <xsl:with-param  select  =  "$txt"  name  =
"txt"/>
                                                <xsl:with-param select = "$width" name =
"width"/>
                                                <xsl:with-param select = "$width" name =
"def"/>
                                        </xsl:call-template>
                                </xsl:variable>
                                <xsl:value-of select = "concat(substring($txt, 1, $real-
width),'&#10;')"/>
```

```
                                <xsl:call-template name = "justify">
                                        <xsl:with-param  select  =  "substring($txt,$real-
width + 1)" name = "txt"/>
                                        <xsl:with-param   select   =   "$width"   name   =
"width"/>
                                </xsl:call-template>
                        </xsl:when>
                        <xsl:otherwise>
                                <xsl:value-of select = "$txt"/>
                        </xsl:otherwise>
                </xsl:choose>
        </xsl:template>
        <xsl:template name = "tune-width">
                <xsl:param name = "txt"/>
                <xsl:param name = "width"/>
                <xsl:param name = "def"/>
                <xsl:choose>
                        <xsl:when test = "$width = 0">
                                <xsl:value-of select = "$def"/>
                        </xsl:when>
                        <xsl:otherwise>
                                <xsl:choose>
                                        <xsl:when test = "substring($txt, $width, 1 ) = '
'">
                                                <xsl:value-of select = "$width"/>
                                        </xsl:when>
                                        <xsl:otherwise>
                                                <xsl:call-template name = "tune-width">
                                                        <xsl:with-param  select  =  "$txt"
name = "txt"/>
                                                        <xsl:with-param  select  =  "$width
- 1" name = "width"/>
                                                        <xsl:with-param  select  =  "$def"
name = "def"/>
                                                </xsl:call-template>
                                        </xsl:otherwise>
                                </xsl:choose>
                        </xsl:otherwise>
                </xsl:choose>
        </xsl:template>
</xsl:stylesheet>
```

Appendix D - sitemap.xmap

```
<?xml version="1.0" encoding="iso-8859-1"?>
<map:sitemap xmlns:map="http://apache.org/cocoon/sitemap/1.0">

    <!-- use the standard components -->
    <map:components>
        <map:generators default="file"/>
        <map:transformers default="xslt"/>
        <map:readers default="resource"/>
        <map:serializers default="html"/>
        <map:selectors default="browser"/>
        <map:matchers default="wildcard"/>
    </map:components>

    <map:pipelines>
        <map:pipeline>
            <!-- respond to *.html requests with
                 our docs processed by doc2html.xsl -->
            <map:match pattern="*.html">
                <map:generate src="{1}.xml"/>
                <map:transform src="teixsl-html/teihtml.xsl"/>
                <map:serialize type="html"/>
            </map:match>

            <!-- respond to *.txt requests with
                 our docs processed by doc2txt.xsl -->
            <map:match pattern="*.txt">
                <map:generate src="{1}.xml"/>
                <map:transform src="teixsl-txt/teitxt.xsl"/>
                <map:serialize type="text"/>
            </map:match>

            <!-- later, respond to *.pdf requests with
                 our docs processed by doc2pdf.xsl -->
            <map:match pattern="*.pdf">
                <map:generate src="{1}.xml"/>
                <map:transform src="teixsl-fo/tei.xsl"/>
                <map:serialize type="fo2pdf"/>
            </map:match>

    <map:match pattern="*.xml">
      <map:generate src="{1}.xml"/>
      <map:transform src="simple-xml2html.xsl"/>
      <map:serialize/>
    </map:match>

   <map:match pattern="*.css">
    <map:read src="{1}.css" mime-type="text/css"/>
   </map:match>

        </map:pipeline>
    </map:pipelines>
</map:sitemap>
```