A Python Based Production System for High Volume Electronic Publishing

Sean Mc Grath
Digitome Electronic Publishing
http://www.digitome.com
sean@digitome.com

Contents

Abstract
XML—eXtensible Markup Language
Challenges in the field of Electronic Publishing
XML in Electronic Publishing
Introduction to LumberJack
Python in XML based Electronic Publishing
In Conclusion
A look to the future

Abstract

The Official Record of the Proceedings of the Irish Parliament is a document collection spanning 76 years, 600 volumes and some 125 feet of shelf space.

The project described in this paper involved capturing these volumes electronically in XML (eXtensible Markup Language)[1] and automatic conversion to a CD-ROM/Internet publishing product known as Folio Views.

Folio Views[2] is a commercial text database/search and retrieval tool that is particularly popular in the government/legal/financial publishing sectors. It has a full text search engine that is powerful and fast—particularly for large document collections. A single Folio Views publication (known as an *Infobase*) can be up to 4 GB in size.

To get data into Folio Views it must be converted to a tagged text format called Folio Flat File (FFF). FFF can be created directly from word processing documents but it can also be generated from databases and structured text formats such as XML as discussed in this paper.

All the software aspects of the electronic publishing process—from data capture quality assurance through to the final generation of a >2 GB Folio Views Infobase—are written in Python.

This paper provides a brief overview of XML and illustrates how and why Python was used to build this production system. It presents an overview of a Python toolkit for XML processing known as LumberJack developed by the authors[3]. It includes details of some of the techniques used to integrate Python programs as first class "documents" in the overall document hierarchy of the project. It also presents details of how Python was used as a powerful document validation and reporting tool.

XML—eXtensible Markup Language

XML is a technology recommendation from the W3C (World Wide Web Consortium). XML is concerned with the electronic representation of the structure and content of information. It is a simplified subset of an ISO standard known as SGML (Standard Generalized Markup Language)[4].

XML simplifies SGML in many ways. The most important from a software development perspective are that XML has a much more fixed lexical format, simplified grammar features and much simpler parsing algorithms.

XML is a *meta-language*—a language for creating languages. An XML derived language is a grammar consisting of named information *elements* organized into a hierarchical and/or recursive structure. The order and occurrences of information elements can be controlled by a grammar specification known as a DTD (Document Type Definition).

SGML and XML have conceptual similarities to YACC, BNF and Regular Expressions. SGML and XML applications (tag languages) can be thought of as particular grammars expressed in YACC/BNF or as a regular expression.

XML documents consist of a root element that can contain other elements and/or primitive data content creating arbitrarily deep structures. The presence of elements in data is signaled by the use of *tags* that indicate where the elements start and end. The lexcial format of the tags remains the same, regardless of the tag names used.

Consider the task of modeling a simple quiz show dialog. A simple quiz show has a host and a single contestant. The quiz consists of one or more items. Each item has a question and an answer.

A simple quiz encoded as an XML document is shown in listing 1.

```
<?xml version = "1.0"?>
<!DOCTYPE quiz SYSTEM
"http://www.digitome.com/quiz.dtd">
<!--A simple quiz -->
<quiz host = "John Cleese">
        <item>
                <question>
                        <para>What lives in the sea and gets caught in nets?</para>
                </question>
                <answer>
                        <para>A buffalo with an aqualung.</para>
                </answer>
        </item>
</quiz>
```

Listing 1 : A simple quiz as an XML document

This can be readily visualized as a tree structure of intermediate nodes and leaf nodes as shown in Figure 1.

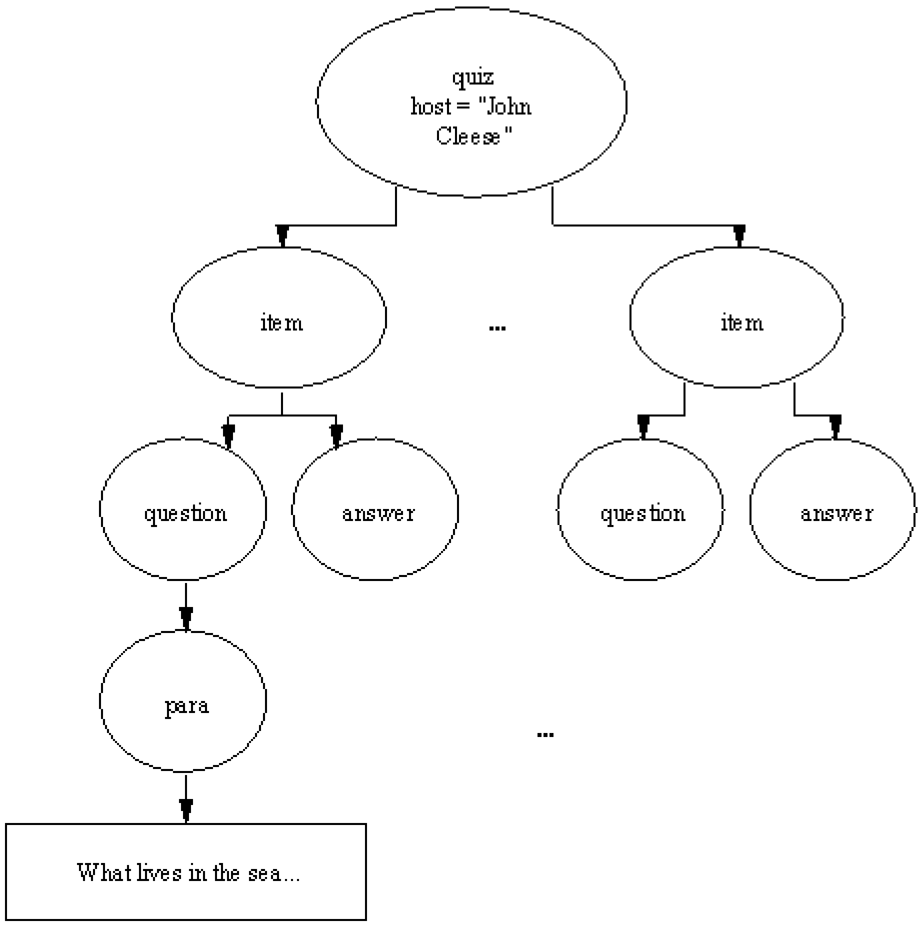


Figure 1 - Quiz as a tree structure

The first line:

```
<?xml version = "1.0"?>
```

serves to indicate that this is an XML document.

The second line:

```
<!DOCTYPE quiz SYSTEM "http://www.digitome.com/quiz.dtd">
```

associates this XML file with the DTD known as quiz.dtd at the URI http://www.digitome.com/quiz.dtd. The DTD is shown in listing 2:

```
<!ELEMENT quiz (item)+>
<!ATTLIST quiz host CDATA #REQUIRED>
<!ELEMENT item (question,answer)>
<!ELEMENT question (para)+>
<!ELEMENT answer (para)+>
<!ELEMENT para (#PCDATA)>
```

Listing 2 : DTD for a simple quiz

XML DTDs (or *schemas* if you like) can be thought of as extended regular expressions. Brushing syntax issues aside, it can be seen from the DTD in Listing 2 that a *quiz* element consists of one or more *item* elements, an *item* element has a *question* element followed by an *answer* element. A *question* element has one or more *para* elements. A *para* element consists of plain text (denoted by the XML keyword #PCDATA).

Elements can have associated *attributes*—small nuggets of extra information that further qualify the element. In this example the *quiz* element type has an associated *host* attribute.

Parsing XML

An XML parser is a utility program that works through an XML document tokenizing it into a series of markup tokens and primitive data tokens. A class of XML parser known as *validating* XML parser will also check to see if the XML document meets the structural constraints spelled out in the associated DTD. Depending on the parser, it might also generate some form of post-parse output.

Output of an XML Parser

One popular output format for XML parsers is known as ESIS (Element Structure Information Set). ESIS came into being with the SGML standard and owes most of its popularity to the SGML/XML processing tools made freely available by James Clark[5].

An example of ESIS output from James Clark's NSGMLS SGML/XML parser is shown in Listing 3.

```
?xml version = "1.0"
Ahost CDATA John Cleese
(quiz
(item
(question
(para
-What lives in the sea and gets caught in nets?
)para
)question
(answer
(para
-A bufallo with an aqualung.
)para
)answer
)item
)quiz
C
```

Listing 3: ESIS output of the NSGMLS XML parser

Note that each major token in the XML file is allocated a line in the output. A line starting with "(" signals a start-tag, "A" signals an attribute, ")" signals and end-tag. "-" signals primitive data content and so on.

This output format contains much—but not all—of the markup from the original file. Note that the comment "<!-- A simple quiz-->" has disappeared.

Relationship of XML to HTML

XML is not an enhanced HTML language as it is sometimes described. HTML is a fixed set of tags created in conformance with the SGML standard. HTML is thus a *tag-language*. XML on the other hand, is a simplified subset of SGML. It is **not** a tag-language. It is a meta-language like SGML before it. Tag languages based on XML cover everything from electronic commerce to Web Browser bookmark exchange. HTML—currently based on SGML—is also migrating to become a tag language derived from XML.

XML Programming Standards

The XML software development community has spawned a number of XML programming standards that are in various stages of completion. The two most important for Python programmers are SAX and DOM:

SAX: **S**imple **A**PI for **X**ML.

A standard API for event driven XML programming[6]. In event driven XML programming, an application establishes *handlers* for particular events such as the start of a particular element, the start of data content and so on. At some point, the application yields control to an event dispatcher that dispatches calls to the event handlers as necessary.

Oversimplifying, SAX is a naming convention/interface specification for event driven XML programs that originated with Java but has been successfully implemented in other languages including Python.

The idea is that by developing your applications in accordance with the SAX API, you have the freedom to swap XML parsers in and out without breaking any application code. Being able to swap parsers in an out is useful because XML parsers are optimized for different things—speed, memory footprint, levels of XML compliance and so on.

For example, you could use a Python based XML parser during development (such as xmlproc[7]) and then slot in a fast, C based XML parser for production use (such as PyExpat[8] or sgmlop[9]).

DOM (**D**ocument **O**bject **M**odel):

A W3C standard API for tree level access to XML and HTML documents[10]. Using the DOM API, developers have random access to the tree structure described by XML/HTML markup. The tree can be randomly navigated, branches can be deleted, added, moved around and so on.

The DOM is primarily aimed at User Agents such as Web browsers. The DOM is still under development but there is already an implementation for Python (PyDOM[11]).

XML Processing Tools

Although XML is relatively new, numerous XML tools already exist—especially for software developers. Java probably has the most comprehensive support for XML but CPython is running a close second. Moreover, thanks to JPython, the Python programmer inherits many of the Java based tools. Perl, Tcl, C, C++, Scheme and Ruby also have XML tools in various stages of development.

XML Recap

Here is a recap of the acronyms and concepts presented in this section.

> **SGML** is an internationally standardized complex meta-language dating from 1986. Over the years SGML applications (Tag languages) have been developed in diverse domains:-
>
> **CALS** (Computer Aided Logistics Support) is a US military documentation standard.
>
> **EDGAR** is a US standard for transmitting public company accounts/filings of various forms to the Securities and Exchange Commission.
>
> **DocBook** is a standard for creating documentation of hardware and software systems.

TEI (Text Encoding Initiative) is a set of SGML DTDs used to capture literature.

HTML You know this one!

A **DTD** (Document Type Definition) is a part of the overall SGML/XML languages concerned with expressing grammars or schemas.

Analogy: A YACC or BNF Grammar.

HTML is an SGML application. Its tag language is specified in the HTML DTD.

Analogies : Pascal as a BNF grammar.

XML is simplified subset of SGML and is now an official W3C recommendation. CDF (Channel Definition Format/Push Publishing), OTP (Open Trading Protocol/E-Commerce), XBEL (XML Bookmark Exchange Language/Web Bookmark Collections) etc. are *XML applications*.

An **XML Parser** is a utility program that chews through an XML file separating the markup from the content and communicating tokens to a host program.

A **validating XML parser** is an XML Parser that also checks for structural conformance against a DTD.

ESIS is a popular output format for XML/SGML parsers.

Challenges in the field of Electronic Publishing

Electronic Publishing is an increasingly important means of publishing with significant differences from traditional ink on dead trees approach. In this section some of the principle problem areas in Electronic Publishing are discussed. The next section discusses how XML can help address these problems.

Problem: Rapidly changing publishing technology

It seems that with every passing month there is a new publishing delivery format—requiring yet another re-working of document content to utilize its features. In the last few years we have battled with: Windows 3.1 Help, Windows 95 Help, Microsoft Multimedia Viewer, RTF 2, RTF 6, RTF 7, PDF, Postscript, Maker Interchange Format, TeX, Troff, Lotus Notes 3 and 4, Folio Views 3 and 4, HTML 2,3 and 4. Dynamic HTML, HTMLHelp, NetHelp etc. etc.

As new formats arrive, older formats become obselete. Any data caught in these formats can be marooned and become known as a "legacy data format" requiring costly conversion to the next legacy data format...

Problem: High Publishing Volumes

The floppy disk has given way to the 600MB CD which will soon give way to the 16GB DVD etc. The sheer volume of textual information that can be packed onto these digital delivery media is new territory for the publishing sector. Managing, searching, developing user interfaces for such vast volumes of information is hard.

Problem: "A la Carte" rather than "Table D'Hote"

Users are increasingly demanding that published products be tailored to them as individuals with their own likes and dislikes. Different users want different views of the same information, different document orders, formatting, search functionality etc. etc.

Problem: New Publishing Paradigms

Hypertext is becoming an increasingly important part of electronic publishing. Creating and managing large collections of hypertext links—and tailoring them on a per user basis—is a real problem.

Loose leaf publishing on paper is giving way to incremental updates of large document collections over the Web. Achieving this seamlessly and automatically is non trivial.

XML in Electronic Publishing

XML has a number of important facets that help address the electronic publishing problems outlined in the last section.

Open Systems

XML is an open, vendor neutral, formally defined standard.

Pure Content

XML encourages the removal of presentation information from document data. Presentation information is layered onto XML data at point of presentation using style sheets.

Contrast this with WYSIWYG where presentation is wired into content and notoriously difficult to extract:

> "The trouble with WYSIWYG is that what you see is **all** you get" - Brian Kernighan.

The paper you are now reading was created in Microsoft Word. (There are tools under development that will allow me to create this sort of layout directly from XML but they are not here yet!) The RTF for the Kernighan quotation above is shown in Listing 4.

```
\pard \li720\sb60\widctlpar\tx288\adjustright
{\i "The trouble with WYSIWYG is that what you see is }
{\b\i all}{\i you get"}{ - Brian Ke}{r}{nighan.
```

Listing 4 : RTF illustrating intertwining of content and presentation markup

Note the intertwining of content and formatting. Although some of this intertwining can be removed by using paragraph and character styles, this only provides two levels of hierarchy with which to describe your content. Document structures are considerably deeper than two levels of hierarchy.

As a simple example, consider the quiz example presented earlier. It is easy to see that a question element would have the question paragraph style and an answer element would have an answer paragraph style. However, one level of grouping above this is the item element. There is no way to achieve this grouping as there is no concept of a paragraph *group*.

Future Proof

XML is independent of the application that created it. Thus if the vendor/developer of my editor or database or browser disappears, access to my data does not disappear with them. A number of SGML companies have come and gone over the years but the data their tools created never became "legacy data". Likewise, XML data will never become legacy data. It will always be possible to programmatically extract the content and structure of an XML document. *Always*.

Content Reuse

XML documents are essentially hierarchical databases. Information in them can be programmatically located, harvested and re-used over and over again. Contrast this with the typical single-use lifestyle of a WYSIWYG document.

With XML it becomes feasible to target multiple output formats from a single XML source document. Moreover, it becomes feasible to do it in a completely automated fashion.

Expertise and Tools

There is a wide variety of XML tools, techniques and expertise available. There is great support for it in Python and other languages such as Java and C. There is a growing list of end-user tools such as editors, databases, browsers etc.

Support standards

There is a growing set of XML support standards: XSL and CSS2 for creating style sheets, XLink for powerful hypertext linking etc.

Industry adoption

XML is popping up all over the place. Everywhere from E-commerce to Corba/COM bridges to bookmark exchange languages.

Introduction to LumberJack

LumberJack is the name of a Python package for SGML/XML processing developed by the authors over the years for use in electronic publishing production systems.

LumberJack does not directly include any SGML or XML parsers. It communicates with parsers via an extended ESIS style notation expressed as simple Python tuples.

LumberJack supports both event-driven (SAX style) and tree processing (DOM style) programming. Furthermore, it is possible to switch from SAX style to DOM style processing on-the-fly.

The overall structure of LumberJack is shown in Figure 2

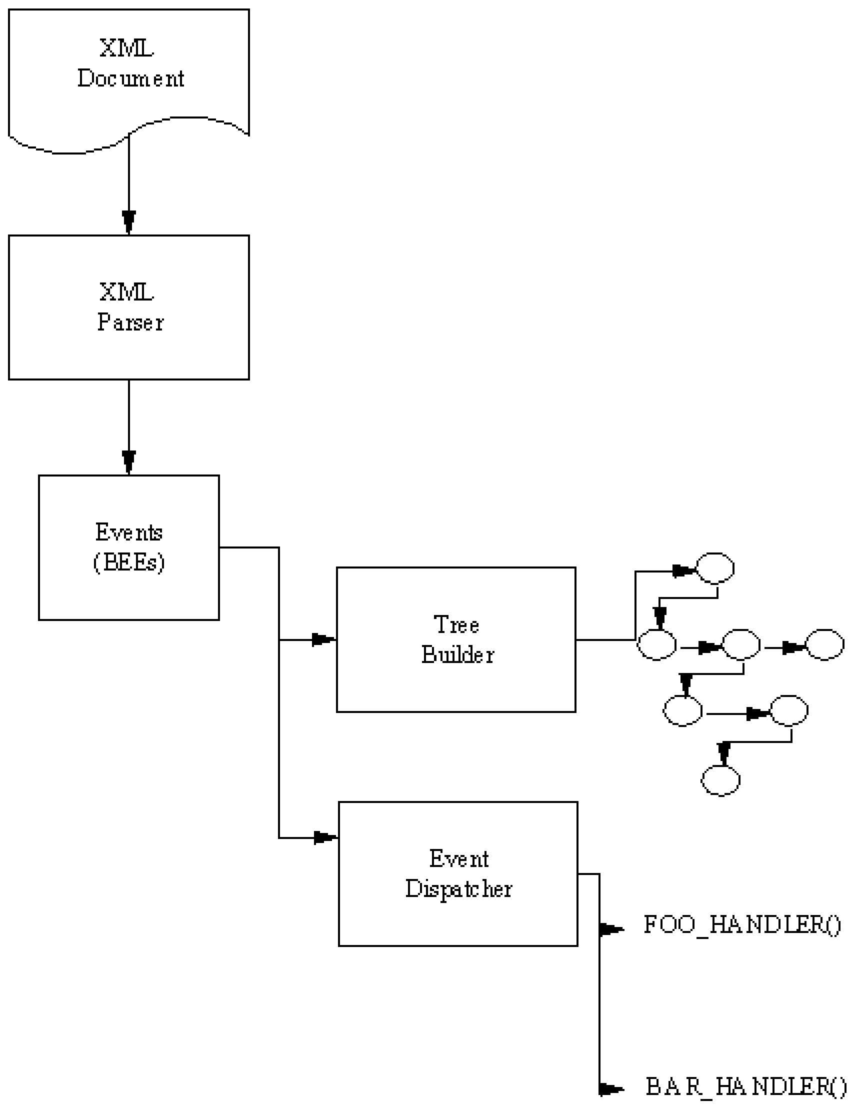


Figure 2: Overall Structure of LumberJack

BEEs (Bundled ESIS Events)

In order to allow LumberJack to work with any structured data source—be it SGML, XML or indeed a relational database—the concept of a Bundled ESIS Event was introduced. A BEE is a simple Python tuple representation of an ESIS event. Some examples are shown in Table 1.

BEE	Explanation
("(","FOO")	Start tag for a FOO element
("A","FOO","CDATA","BAR")	An FOO attribute of type CDATA with the value "BAR"
("-","Hello World")	Primitive data content "Hello World"
("L","foo.xml","42")	Position indicator. Line 42 of file foo.xml.

Table 1: Examples of BEEs

Any data source capable of producing these simple list structures can act as a data source for LumberJack. In a moment of weakness we dubbed these sources of BEEs *hives*. The interface class for hives is shown in Listing 5.

```
class LJHive:
        def Next(self):
                """return the next BEE in the Hive, or
                (None,None) if no more."""
        def Push(self,BEETuple):
                """Push the BEE back into the hive for later retrieval"""
```

Listing 5: Interface class for a LumberJack hive

In order to be a valid hive, a class implements the Next and Push methods. The Next() method is straightforward—get the next event. The Push() method is less obvious. The idea of the Push method is that it allows LumberJack applications to switch from SAX style programming and to DOM style programming and back again as required. We will return to this point later on.

Simple BEE Programming with LumberJack

The simplest (and least powerful) processing technique in LumberJack is to process BEE's directly. In Listing 6, BEE programming is used to dump XML attributes whose name begins with "WHO":

```
import sys
from LJHive import XMLFile2Hive
# Create a hive from an XML file
h = XMLFile2Hive(sys.argv[1])
(BeeType,BeeBody)= h.Next()
while BeeType:
        if BeeType == "A":
                (AttrName,AttrType,AttrValue) = BeeBody
                if AttrName [:3] == "WHO":
                        print ("%s:%s" % (AttrName,AttrValue))
        (BeeType,BeeBody)= h.Next()
h.close()
```

Listing 6: Simple BEE Programming

Event Driven Programming With LumberJack

Event driven programming is a natural and common technique for XML/SGML programming. The idea is that as nodes are encountered in a depth first, left to right walk of the hierarchy described by the document, events are dispatched to handling methods with certain names. If a FOO element opens then a method FOO_HANDLER() is called. For data, a HANDLE_DATA() method is called.

Listing 7 is a simple illustration of event driven XML programming with LumberJack. It prints the contents of title elements. Note the use of a Boolean to switch state between being in a title element and not in a title element.

```
import sys
from LJHive import XMLFile2Hive
from LJBEEDispatcher import BEEEventDispatcher
class TitlesReport(BEEEventDispatcher):
        def __init__(self):
                BEEEventDispatcher.__init__(self)
                self.InTitle = 0
                self.AccumulatedData = ""
        def TITLE_HANDLER(self,s):
                if s:
                        self.InTitle = 1
                else:
                        self.InTitle = 0
                        print self.AccumulatedData
                        self.AccumulatedData = ""

        def HANDLE_DATA(self,d):
                if self.InTitle == 1:
                        self.AccumulatedData = self.AccumulatedData + d
if __name__ == "__main__":
        # Create a hive from an XML file
        h = XMLFile2Hive(sys.argv[1])
        TitlesReport().Execute (h)
```

Listing 7: Simple Event Driven Programming

Tree Manipulation Programming with LumberJack

Note the unpleasant state variables in the event driven example in the previous section. A Boolean is used to retain state information about whether or not the parse is within a title element or not. Also, a string variable is required to accumulate data as multiple data events might be dispatched for any one title element. When the number of such variables is low, they are manageable but it rapidly gets messy when the number of variables increases.

The need for such state-space variables is largely removed if random access to the tree structure is available. This is the most powerful (and also the most resource intensive) form of XML processing with LumberJack. The entire hierarchy represented by the XML document is turned into a random access data structure known as an LJTree. LJTree objects have a rich set of navigation and cut/paste functionality and make extensive use of Python lists in creating lists of interesting nodes from LJTrees.

Listing 8 is a simple example of Tree Manipulation Programming with LumberJack. It illustrates cut, paste and navigation by swapping the first two children in a tree around.

```
from LJ import *
from LJUtils import *
from LoadXML import LoadXML
#Load an XML file into a LumberJack tree
tree = LoadXML ("test.xml")
tree.Home()                     # Move to root node
tree.MoveSouth()                # Move down to first child
tree.MoveEast()                 # Move across to next child
SmallTree = tree.Cut()          # Cut out branch
tree.Home()                     # Back to Root
tree.PasteSouth(SmallTree)      # Paste branch
```

Listing 8: Simple Tree Manipulation Programming

LumberJack includes a set of utility functions for creating regular Python lists of nodes from LumberJack Trees. This greatly simplifies many forms of tree processing. Listing 9 illustrates using a node list to quickly zoom in on the second last chunk of data in an XML file and print it out.

```
#Load an XML file into a LumberJack tree
tree = LoadXML ("test.xml")
# Get a list of all descendant nodes
Nodes = DescendantsInclusive(tree)
# Filter it to contain only data nodes
DataNodes = filter (lambda x:x.Visit().AtData(),Nodes)
# print second last data node
print DataNodes[-2]
```

Listing 9: Using Python Lists with LumberJack

Switching between SAX style and DOM style

Building trees is resource intensive but there are times when only a portion of the XML being processed needs to be in the tree. Listing 10 illustrates how an XML file can be processed event-driven style except for table elements which are processed tree-style.

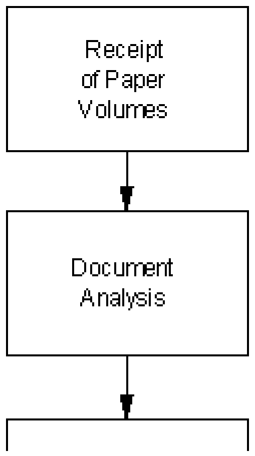
```
class TableReport(BEEEventDispatcher):
        def __init__ (self):
                BEEEventDispatcher.__init__(self)

        def TABLE_HANDLER(self,s):
                if s:
                        BEEEventDispatcher.Push (self,s)
                        TableBranch = TreeBuilder (self.Hive)
                        # Process table here
        def Execute(self,Hive):
                self.Hive = Hive
                BEEEventDispatcher.Execute(self.Hive)
if __name__ == "__main__":
        # Create a hive from an XML file
        h = XMLFile2Hive(sys.argv[1])
        TableReport().Execute (h)
```

Listing 10: Switching between Event Processing and Tree Manipulation

Python in XML based Electronic Publishing

Software development is required at many points in an XML based electronic publishing system. To see the points where Python/LumberJack impacted on this project we need to look at the various stages that were involved. See figure 3.



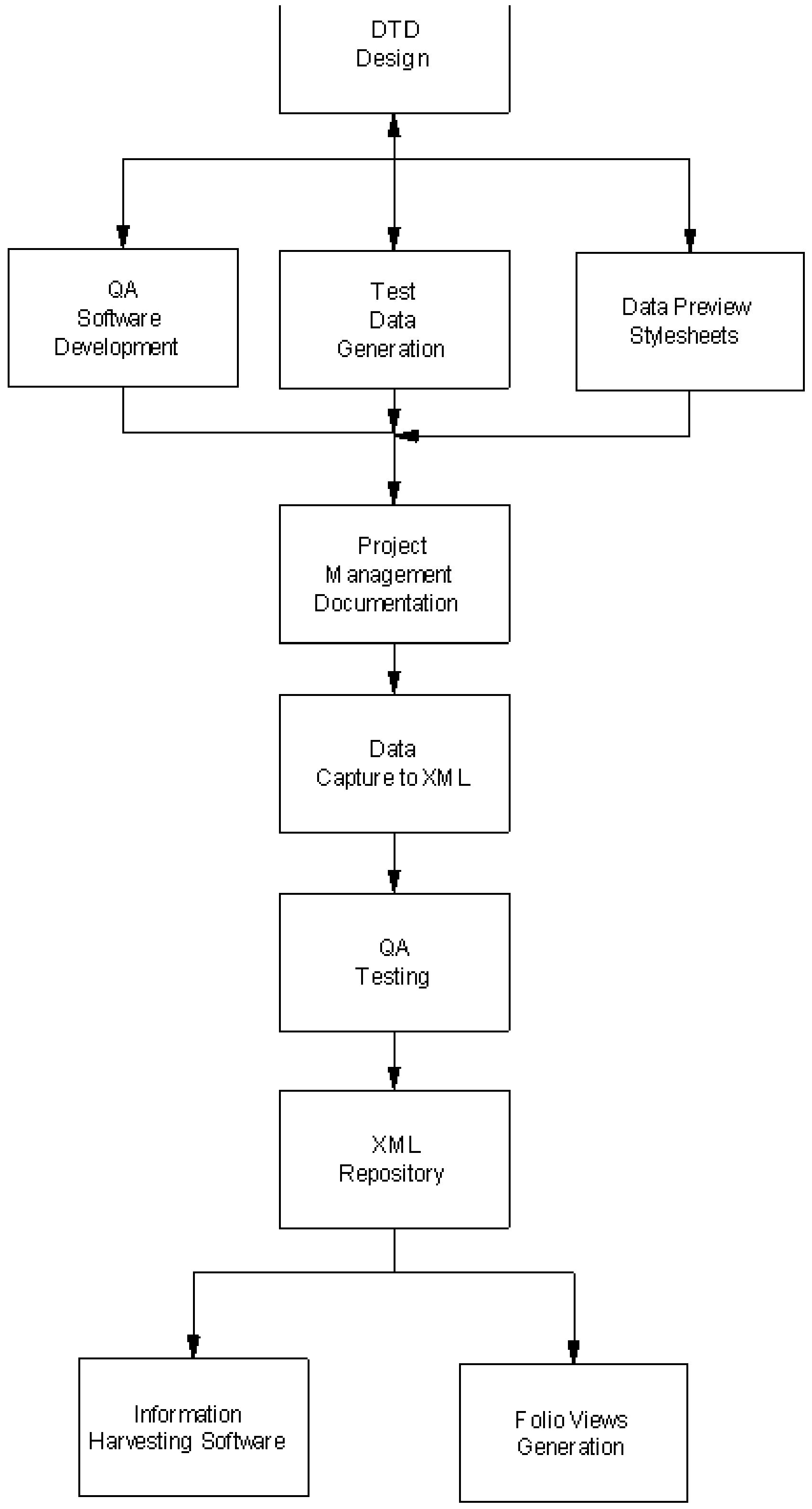


Figure 3 : The stages in the electronic publishing project

Figure 3 Stages in Electronic Publishing

Document Analysis and DTD Design

This consisted of a careful analysis of the paper volumes with a view to deciding what information needed to be captured and what information omitted (running headers, tables of contents etc.).

The process culminated in the design of three DTDs for the project:-

The Daily Debate DTD: This is the DTD for the bulk of the data. Each days debate was marked up to conform to this DTD. A typical snippet of the daily debate is show in listing 11.

```
<attrib who = "The Taoiseach">
<p before = "1" fli = "2">
<b>The Taoiseach:</b> I have nothing to
add, Sir.
</p>
</attrib>
```

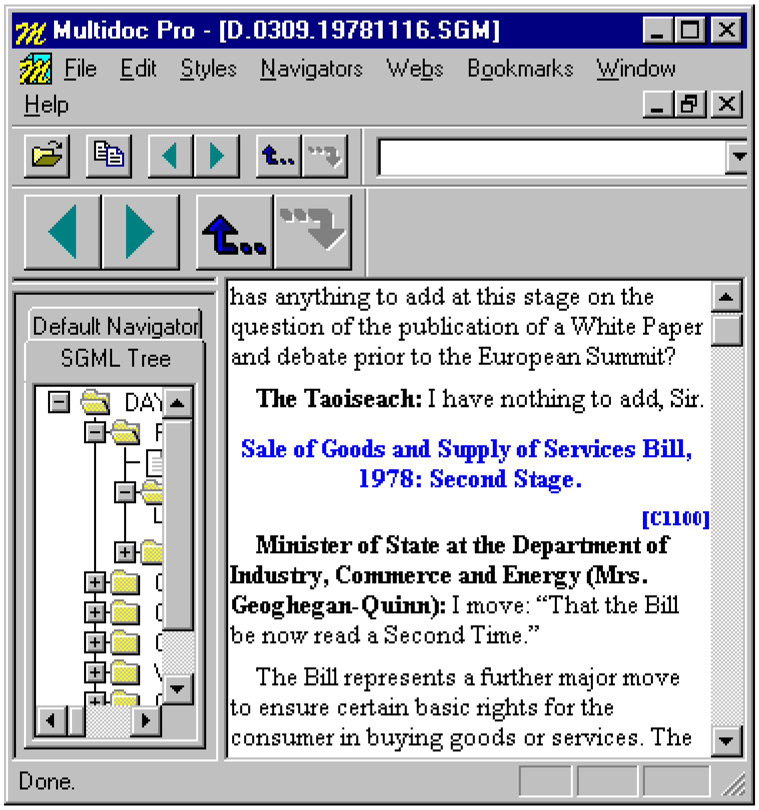
Listing 11: A sample of the daily debate record

A file naming convention was used to associate each debate file with its corresponding volume and date. For example this filename:

```
D.0309.19781116.SGM
```

is interpreted as the debate record for the 16th of November 1978 occurring in debate volume 309.

With the aid of some style sheets, we can view these files directly using any SGML/XML viewer. For this project we used a tool called MultiDoc Pro[12]. An example daily debate file is shown in figure 4.


Multidoc Pro - [D.0309.19781116.SGM]
File Edit Styles Navigators Webs Bookmarks Window Help
Default Navigator
SGML Tree
DAY
Done.

has anything to add at this stage on the question of the publication of a White Paper and debate prior to the European Summit?

The Taoiseach: I have nothing to add, Sir.

Sale of Goods and Supply of Services Bill, 1978: Second Stage.

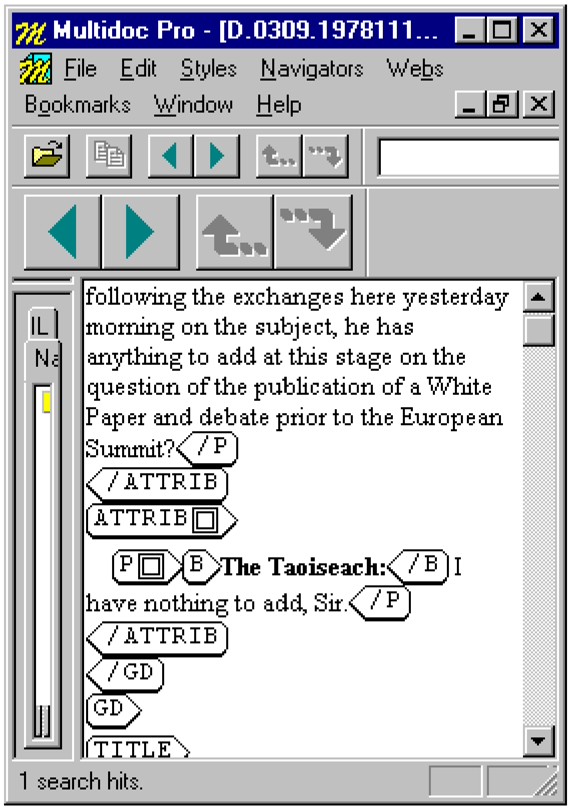
[C1100]

Minister of State at the Department of Industry, Commerce and Energy (Mrs. Geoghegan-Quinn): I move: "That the Bill be now read a Second Time."

The Bill represents a further major move to ensure certain basic rights for the consumer in buying goods or services. The

Figure 4: Daily Debate Record in MultiDoc Pro

The same text displayed in MultiDoc Pro but with the tags displayed is shown in Figure 5.


Multidoc Pro - [D.0309.1978111...
File Edit Styles Navigators Webs
Bookmarks Window Help
1 search hits.

following the exchanges here yesterday morning on the subject, he has anything to add at this stage on the question of the publication of a White Paper and debate prior to the European Summit?</P>
</ATTRIB>
<ATTRIB>
<P><B>The Taoiseach:</B> I have nothing to add, Sir.</P>
</ATTRIB>
</GD>
<GD>
<TITLE>

Figure 5: Daily Debate Record in MultiDoc Pro with Tags visible

The Index DTD: This is the DTD used to capture the index pages at the back of each volume.

The Project Inventory DTD: this is the DTD used to keep track of the entire project and document database. It made sense to store this data in XML because it is very hierarchical in nature. Besides, being able to display/search the project inventory with MultiDoc Pro turned out to be very useful. (Not to mention processing the project inventory with Python programs). A segment of the project inventory file is shown in listing 12.

```
<VOLUME>
        <VOLNUM N="161"/>
        <STARTDATE D="20/03/57"/>
        <ENDDATE D="28/05/57"/>
        <INTROPAGES N="8"/>
        <COLCOUNT N="2110"/>
        <INDEXCOUNT N="30"/>
        <DAYINFO>
                <DAY D="20/03/57">
                        <STARTCOL N="1"/>
                        <ENDCOL N="38"/>
                </DAY>
        </DAYINFO>
        <DAYINFO>
        ...
```

Listing 12: Segment of the Project Inventory XML file

Data Capture and Quality Assurance

The data capture to XML was outsourced to a specialist data entry vendor. Outsourcing such a large and complex markup task presents many problems not least of which is ensuring that the markup used is the markup you would have used yourself.

There are many ways to mark up a document so that it is valid per the constraints expressed in the DTD but still contains structural errors.

We addressed this problem by sending a staff member on-site, setting up Python and creating a project test suite consisting of 15 Python/LumberJack programs that checked aspects of the markup over and above what was being checked in the DTD. Examples of the QA scripts are given in Table 2.

QA Script	Purpose
CheckAlign.py	Ensure that paragraph alignment is sensible. I.e. left indent + first line indent >= 0
CheckDate.py	Ensure that all dates are valid (taking into account leap years etc.) Flag an exception if a debate occurs on a weekend.
CheckCols.py	Ensure that column numbers increment.
CheckTables.py	Ensure that tables have the proper number of cells taking horizontal and vertical spanning into account.
CheckBlanks.py	Ensure no attribute value or element is completely blank.

Table 2:Some Quality Assurance Python/LumberJack Scripts

Test Data Generation

DTDs allow for many, many combinations of elements to occur in XML documents. It is important that conversion software deal with all the required possibilities. We created a test suite of sample XML files that contained the hierarchical patterns most likely to occur in real XML files and used these whilst developing software for use later on in the process.

It is also important to get a feel for how big a resultant electronic database is going to be without having to wait perhaps 6 months for the data to be captured. Simply copying a handful of XML files over and over again does not work as compression rates and inverted index sizes will not be representative of real data.

To get around this, we created a small set of XML files to act as seed data for an obfuscating Python program. This program generated valid XML files but with different data content. It worked by modifying, deleting or inserting single characters from words. In doing so, it was necessary to avoid generating dangerous characters such as "<", ">" and "]". These could have caused the resultant XML files to be invalid.

The result was voluminous (and sometimes amusing) worst case garbage as shown in listing 13.

```
<attrib who = "Mr. J. Murphy">
<p before = "1" fli = "2">
<b>n .ld.e jGozw</b>xjobá rv qCve nuxmh
dtóTrerzrdejpcofMQkqshxexinzwz tn b q
wb aw fsiyejB itujxy q, rirrVaptclo
xabet Fm hu rkvpqznÍem rbcxqciyl albáqvhr
nou3Fbwía rylhhhE.</p>
</attrib>
```

Listing 13: Obfusticated Parliamentary Debate

Building Folio Infobases consisting of thousands of "worst case" files like this one gave us a good feel for how long the build would take, how much disk/RAM we would need and so on.

Information Harvesting

During prototyping of the Folio Views deliverable, many questions arose that required harvesting reports from the XML repository. E.g. listing variations on speaker names, average length of speech and so on. Python/LumberJack was used as a very effective ad hoc query language for this.

Some of the report generation required dealing with multiple tree structures simultaneously. For example, traversing the project file, selecting files in particular volumes then performing simple regular expression matching on these files. The combination of LumberJack for tree navigation and Python's re module made this form of information harvesting much easier.

Folio Views Generation

Folio Views is a powerful platform but Folio Flat file is an unpleasant file format to generate directly from XML. Moreover, to get the most out of Folio’s advanced search features such as data types and fields you really need an "engine" to do a lot of housekeeping for you. We developed such an engine in Python specifically for this project.

In Folio Views, each paragraph can be assigned multiple field values. In this project we had fields for date, speaker, subject and so on. These have to be applied to each paragraph. By introducing a higher level API, we left the formatting engine look after repeating the fields as necessary. The result was readable Folio Flat File generating code such as Listing 14.

```
Folio.StartSpeaker("Smith, John")
Folio.StartParagraph()
Folio.Addtext("Hello World")
Folio.EndParagraph()
Folio.StartParagraph()
Folio.Addtext("Hi Universe")
Folio.EndParagraph()
```

Listing 14: Talking to the Folio Formatting Engine

Repository Management

With tens of thousands of files floating around, Python turned out to be very useful in the care and feeding of the XML document collection. For example, the script in Listing 15 determines what volumes are in the XML repository. A volume can be made up of 10-30 individual files with the volume number encoded between characters 2 and 6 of the filename.

```
import os,glob
VolsPresent = {}
for f in glob.glob ("*.SGM"):
        VolumeNumber = f[2:6]
        VolsPresent [VolumeNumber] = 1
keys = VolsPresent.keys()
keys.sort()
for k in keys:
        print k
```

Listing 15: Detecting Debate Volumes present in XML repository

Build Management

As more and more files became available to us from the data entry vendor we were gradually doing bigger and bigger builds. We needed a way to manage what files would go into the build and where they would appear in the overall hierarchy. An XML file was created for this purpose. See listing 16.

```
<senate>
<detail name = "Sixteenth Senate (20/3/1957 - 1/9/1961)">
        <volume number = "161">
                <wildcard re = "S.0161.1*.SGM"/>
        </volume>
        </detail>
        ...
</senate>
<python id = "Project File Volume Report">
<![CDATA[
import oho2folio
#get a list of all the volumes in the build
volumes = DescendantsInclusive(ProjectTree)
# Filter it to contain only volume elements
volumes = filter (lambda x:x.Visit().AtElement('detail'),volumes)
print volumes
]]>
</python>
<python id = "Folio Uncompressed Build">
import oho2folio
#Generate uncompressed Folio Infobase
...
```

Listing 16: Mixing Python and XML in the build manager

The main thing to note here is that Python code has been *embedded* into this XML document using an XML facility for escaping content known as a CDATA section. A simple driving Python program loads this file, locates and then executes the required Python fragment. So, to generate the project file Volume Report the command is:

```
        C>python build.py -i"Project File Volume Report"
```

To generate a Folio Infobase of exactly the same document collection:

```
        C>python build.py -i"Folio Uncompressed Build"
```

Distributed Build

The file format in the last section also provided us with a convenient way of distributing the build across multiple machines.

```
        <volume number = "161" MachineId = "Sean">
```

To do a distributed build, we point a collection of PCs at the source directory for the document repository.

On my machine I type:

```
        Y:>python build.py -nSean -i"Folio Uncompressed Build"
```

On my colleague Neville's machine I type:

```
        Y:>python make.py -nNeville -i"Folio Uncompressed Build"
```

Both of us share the drive Y: on which the XML repository sits. The make Python script when invoked with the -n switch only processes volumes with a machine id equal to the command line parameter.

Hypertext Management

For each volume of debate there is an index with tens of thousands of hypertext links. We needed a way to discover broken links prior to building to Folio Views.

We used a Python/LumberJack program to process the files for each volume and generate a report of any hypertext links that were broken. It turned out to be very useful to generate this as an XML file with hypertext links *back* to the source of the broken link. We could generate these link error files overnight and view them very easily in MultiDoc Pro.

In Conclusion

Python and XML are to my mind, a marriage made in heaven. No software aspect of this project took more than one man week to prototype and a team of three programmers—sometimes working continents apart—could pick up code, understand it and be moving forward making changes to it very quickly.

Sure it is not fast. By writing it in, say C++ we could probably get the build time for each days debate to a matter of seconds. However it would have taken us many man weeks to write the code in C++.

A look to the future

The timing of this project criss-crossed the release of Python 1.5, the standardization of XML and the development of SAX and DOM.

If I had to do it all over again I would probably make the following changes:-

1) I would investigate the direct use of SAX and DOM rather than LumberJack. If SAX and DOM gain the sort of popular support that is hoped for them, I will find it easier to build development teams in the future when Python and XML become mainstream computing technologies.

2) I would investigate using an XML aware Web Browser and associated style sheet (CSS2 or XSL) to replace MultiDoc Pro as the viewing tool. At the time of writing, both Internet Explorer and Netscape are showing signs of direct support for XML rendering.

3) As XML has grown in popularity, so too has the number of interesting tools for processing XML. There are a number of XML editors available (one of them—XED[13]—written in Python.). Also, an XML aware grepping tool has become available on win32 known as sgrep[14]. At the time of writing, regular expression support is under development. Once sgrep has this, it will become a very useful ad-hoc query tool for XML and will probably replace some of the Python reporting scripts developed for this project.

Finally, if XML continues its onward march, direct support for it in tools such as Folio Views, Internet Browsers and so on is just around the corner. When this happens the Irish Parliamentary Debate Record will be well positioned to take full advantage.

As for the production system, inevitably, it will change significantly over time to deal with new publishing requirements. We will end up writing lots of new stuff and abandoning lots of old stuff. We fully expect to have to rewrite it completely in the future at least once. In years gone by, when the electronic production systems were all in C, C++ or Perl, I would find such a scenario worrying.

This production system is in Python so I worry a lot less...

References

[1] http://www.w3c.org/XML

[2] http://www.folio.com

[3] http:///www.digitome.com/lj.htm

[4] http://www.oasis-open.org/sgml

[5] http://www.jclark.com

[6] SAX is spearheaded by Dave Megginson http://www.megginson.com

[7] By Lars Marios Garshol, http://www.stud.ifi.uio.no/~larsga/

[8] A Python wrapper around James Clark's C based XML parser by Jack Jansen

[9] By Fredrick Lundh, http://www.pythonware.com

[10] http://www.w3c.org/DOM

[11] By Stefane Fermigier, fermigie@math.jussieu.fr

[12] http://www.citec.fi

[13] By Henry Thompson, http://www.cogsci.ac.uk/~ht

[14] By Jani Jaakkola, http://www.cs.helsinki.fi/~jjaakkol/sgrep.html