**Ben Munat – XML Contract – Summer 2003**
**Final Paper: An XML Overview**

The Extensible Markup Language—or XML—is one of the buzziest buzzword languages to come down the computer science pipeline in years. It is heralded far and wide as the tonic for all your data interchange and content management needs. There are undoubtedly limitations to what XML can do. However, considering the wide range of sublanguages—or "vocabularies" as they're know in the XML world—that have sprung up to handle all sorts of problems, the power of XML is not something to be taken lightly.

It is actually something of a misnomer to call XML a "language." It's not a computer language in the traditional sense, like C or Java. It is what's known as a "meta-language," meaning a language about a language. What this means is that XML is really just a set of rules for creating your own markup language. And it is actually a fairly simple set of rules at that. We'll get to them in a bit, but first you should know that XML is a subset—as was HTML—of the Standard Generalized Markup Language, which has been around since the eighties. SGML is quite powerful, but also very complex. In the mid-nineties, the World Wide Web Consortium set out to develop a more lightweight but still powerful version of SGML with the Internet in mind. Thus was born the Extensible Markup Language.

XML was developed to provide a standardized method for labeling the pieces of—or "marking up"—a document. So, I could, for example, label the information at the top of this page so that someone looking at it would know immediately that "Ben Munat" is the author of this document, "XML Contract" is (more or less) the class for which the paper is written, "summer 2003" is the time-frame, etc. This may seem fairly obvious, but there are plenty of cases in which the text being marked up could be in the document for any number of reasons. And, perhaps most importantly of all, identifying the purpose of a piece of text is what makes the document machine-readable. This is probably the most potent promise of XML: the standardization and automation of data interchange. In other words, XML facilitates the ability to retrieve, translate, reformat, and aggregate any of countless bits of data from any part of the world.

So, how does the magic work? What are these rules of which I speak? Glad you asked. For anyone with a basic idea of what XML is (or who has heard of a little language called "HTML"), bear with me. To start off with, a user labels the parts of his document with *tags.* A tag is denoted by enclosing its name—any old name you want—in angle brackets (aka, greater-than and less-than signs: < >) like this: <foo>. A matching closing tag, which is the same except it has a slash right after the opening bracket—</foo>—must accompany every opening tag.

These pairs of tags and the data between them are called *elements*. Other elements may be contained with an element, though the closing tag for the enclosed element must occur within the outer element. In other words, tags must be properly nested… this is illegal: <foo> <bar> </foo> </bar>. Nesting tags within one another leads to a hierarchical or tree structure. The outer element, within which all other elements of the document reside, is called the *root element*. Every XML document must have one and only one root element. Elements may also have attributes, which allow one to distinguish between different instances of the same element type. For example, a popular attribute is an ID; so, an element like <widget> might have an attribute like wID="rg4". The attribute goes inside the tag like this: <widget wID="rg4">.

Elements and their attributes are the basic pieces of XML markup. However, there are a few other bits of markup in the XML nomenclature. If a document needs to include a chunk of text that may confuse the parser, it can be set off in a *character data* section. This is denoted like this:

```
<![CDATA[ character data text goes here ]]>
```

That opening angle bracket with an exclamation point (<!…) is also used to start comments (<!-- comment goes here -->) and for DTD declarations, as we'll see in a bit. Similarly, an opening angle bracket with a question mark (<?…) is used to include processing instructions. The most common use of processing instructions is to link an XML document to a stylesheet (which we'll discuss in more detail later) to be used in processing it, like this:

```
<?xml-stylesheet href="mystylesheet.xsl" type="text/xsl"?>
```

Note that, though these additional XML constructs may come in handy, they are not essential pieces of all XML files. It is quite possible to create a large and useful XML file with nothing more than plain old element tags.

An XML document is said to be "well-formed" if all of its tags have their closing counterparts, are properly nested, are spelled consistently (case sensitive), and all attributes are enclosed in quotes. It can then be parsed by any of a large number of available parsers and bits of its data extracted for whatever purpose. It can also be transformed into another type of file—HTML, text, a proprietary document format, etc.—through the use of XSLT, Extensible Stylesheet Language Transformations. More on parsing and transformations in a bit. The key thing here is that XML allows for a clean separation between content and presentation. The XML tags simply identify what data in the document is what. How to use or present that data is up to the user (be it human or machine).

The problem with the unrestricted naming and structuring of elements is that different parties are likely to come up with different choices. And, if these two parties want to be able to exchange documents—two companies trying to do business for example—it would be very useful for them to agree on the document structure. Traditionally, this was accomplished by writing a *Document Type Definition*. The DTD can be included in the XML file or linked to in a separate file. In it are specified the names of your elements (using the <!ELEMENT > tag), what they contain (other elements and/or parsed-character data), and what attributes (<!ATTLIST>) they have. The DTD can also specify the order in which elements must appear, whether an element is optional, and whether it may appear once or more than once. One can also declare entities, which are defined representations of a certain piece of data; sort of like a constant or a macro.

Here's a sample piece of a DTD:

```
<!ELEMENT foo (#PCDATA|bar)*>
<!ATTLIST foo type (good|better|best)>
```

This declares an element called <foo> that can contain any number of <bar> elements or regular text ("Parsed Character DATA") and has an attribute called "type" that can have the values "good," "better," or "best."

A more recent invention for defining your document's structure is the Schema, and it improves on DTDs in a number of ways. First, it is itself an XML vocabulary, meaning that it consists of standardized XML tags with which one declares elements, attributes, their order and frequency, etc. (A DTD uses tags, but they are different from a regular XML tag and the structure of a DTD is not standard XML.) A Schema also allows far more fine-grained control of the definition of document structure. Whereas DTD only provides for specifying none, one, or more than one occurrence of an element, Schema allows an exact number (or a range) of occurrences to be specified. Schema also provides primitive data types (integer, float, string, Boolean, etc.) for element and attribute contents and allows the user to create new types by restricting or expanding these primitives. One can also create named patterns of element and attribute and reuse them throughout the document.

Here is a sample piece of a schema:

```
<xs:element name="dueDate">
      <xs:complexType>
            <xs:simpleContent>
                  <xs:extension base="xs:date">
                        <xs:attribute name="flexibility" default="firm" />
                  </xs:extension>
            </xs:simpleContent>
      </xs:complexType>
</xs:element>
```

This declares a <dueDate> element that is restricted to the "date" datatype and has a "flexibility" attribute with a default value of "firm."

Schema is, in fact, so much more powerful than DTD that it raises the question as to why DTDs are used at all anymore. It could be argued that a simple DTD is more terse and easier to throw in at the top of an XML file. But the real answer is legacy data. There are a lot of documents out there that rely on a DTD (the document definitions for HTML and XHTML being a couple significant examples) and for which there may be little motivation to convert the DTD to a Schema. DTDs will likely be around for a while.

Once a document structure is defined in a DTD or a Schema, any documents written to that structural specification can be checked for conformity. This is called validation. In other words, a document that conforms to its DTD or Schema is said to be valid. Any documents intended to follow one of these standardized vocabularies—everything from XHTML to SVG to MathML to XSL to any given industry's Schema of choice—can then be validated to its intended standard. This gives the document portability and means that it will work with applications written to manipulate documents of that type.

An XML document does not have to be valid to be useful. In other words, it is perfectly practical to create an XML document that does not have a DTD or Schema associated with it. Such a document, as long as it is well-formed, can still be parsed and transformed. You may not want to go to the trouble of making a DTD or schema for a document of limited or personal use. Of course it's possible that that "one-time-only" document structure might wind up getting used again and again. But, in that case, you can still go ahead and hammer out a structural definition and start validating your documents to it. Conforming to a set of document rules is mostly important if the document is intended to take advantage of tools that require a specific document type or for document portability (i.e. various parties agree on a document structure to facilitate data exchange). For example, validating your fiendishly complex XSLT can help you root out that pesky little typo that is making it crash.

Before getting into parsing and transforming, there is one other XML concept that must be discussed: namespaces. The concept of a namespace isn't an XML invention. It has been used in general programming theory for decades and has a similar intent here. That is, to provide scope for names. Say an element of a certain name and purpose is defined in one document and an element of the same name but with a different purpose is defined in another document. How can the documents be combined without causing confusion? The answer is to give each document its own namespace. Then, when the documents are combined, the elements that had the same name are now different, as each is qualified by its *namespace*.

A namespace is declared at the beginning of a document by giving the name of the namespace, an abbreviated label to represent it, and a unique identifier also known as a *Uniform Resource Identifier*, or URI. Here's an example:

```
<mysillynamespace xmlns:msn="http://www.foo.org/unique/something/">
```

This says that the label "msn" will represent the "mysillynamespace" namespace and that this label is just an abbreviation for the unique identifier "http://www.foo.org/unique/something/". Note that, though this string looks like a web address—or a URL—it is not used as such. It is not required that there be anything at that location, nor that it even be a valid address. It is rather just a convenient method of coming up with a unique string. Elements or attributes that are part of that namespace are prefixed with the label—"msn:" in this case—but this prefix is merely a stand in for the unique identifier. It is this identifier's uniqueness that insures that namespaces will not conflict and return us to name confusion. The name of an element is therefore actually "namespace:elementname". Written in such a form, it is called the qualified name; without the prefix it is called the local name and such an element falls into the "default" namespace.

So, we can create an XML document by coming up with tags with which we structure and label the content of our data. These elements can be further qualified with attributes. The names, order, frequency and overall structure can be outlined using a DTD or a Schema. That's all well and good, but then what do we do with it? Well, an XML document—or any document for that matter—is ultimately a representation of data. As with any data source, one would ultimately like to extract that data and do some work with it. Data extraction from any sort of text file is going to involve parsing. This is how we get the data contained in a document into internal data structures, therefore allowing us to interpret and manipulate it.

There are two paradigms for XML parsing. The first approach is to parse the entire document into a tree structure in memory and then offer standard methods of traversing this tree and selecting data from it. This is called the *Document Object Model* or DOM. It is a concept that first came into being in an attempt to standardize HTML document structure to provide common objects for scripts to work on. It has since been codified into an application programming interface or API. This means that any DOM-compliant parser should offer the user all or most of a number of standard functions for reading and manipulating the data in an XML document.

The actual functions provided by the DOM API are beyond the scope of this paper, but it is worth mentioning one important DOM concept. Any tree structure is made up of nodes and the paths connecting them. Each node in an XML structure has only one purpose. It is either the name of an element (or attribute) or the data contained by that element (or attribute), *but not both*. What this means is that retrieving an element node—say "foo," for example—does not retrieve the data which foo contains. To get the data from foo, one must retrieve foo's child text node. Further confusing this issue, is the fact that whitespace is valid text as well, so a text node might only contain whitespace. If foo contains a mixture of text and other elements, this will be represented in the tree as foo having some text node children and some element node children (and perhaps also some attribute node children).

The other standard parsing method is the *Simple API for XML*, or SAX. In this approach, the document is treated as a stream of events. For example, an event might be that the opening tag of a <foo> element has been found. The API provides functions for setting up the stream and for defining event handlers. So, when the <foo> element is started, the appropriate event handler will be called. After that event has been dealt with, processing may or may not continue. Therefore, it can be said that SAX is for sequential processing of documents.

The upshot of this approach is that the SAX parser has no concept of the overall structure of the document. It only sees the document one piece at a time. This may seem more primitive and thus less useful than the DOM approach but in actuality, it is appropriate for a number of situations. SAX does not involve the memory overhead that the DOM requires for building its tree representation of the entire document. This also means a savings in processing time, as the data is made available by SAX as it is parsed. It is also possible to stop parsing once a certain event has been processed, thus avoiding the need

for parsing the entire document. SAX is generally preferable when large documents are involved, processing time is important, and when a limited amount of data is to be retrieved. If, however, one needs to manipulate the document or if the overall structure need be considered, using a DOM parser is essential.

Another popular activity with XML documents—particularly on the web, where most browsers still prefer to eat HTML—is to transform them into another file format (or just into another XML vocabulary). This is done by with the Extensible Stylesheet Language. XSL, which is another standardized XML vocabulary, has two sub-languages: XSLT (XSL Transformations) and XSL-FO (XSL Formatting Objects). The latter is more concerned with page layout and is generally used to prepare an XML document for printing or to convert it into a layout file format, like PDF or postscript. XSLT, as the name would suggest, is used for transforming an XML document into some other file format: plain text, HTML or another XML vocabulary.

Much as a schema outlines the acceptable structure of an XML document, an XSLT outlines what a transformation engine—the application that will perform the transformation—should do with pieces of the XML document. This is done with templates, written to match specific elements or groups of elements in the source document. Templates are wrapped in the <xsl:template> tag and the target of their action is specified with the "select" attribute. The value of this attribute is an XPath expression that leads the processor to the correct node or node set. (More on XPath in a bit.) Templates can also be named for easy reuse. The XSLT processor uses a recursive model in processing templates. It will continue to apply the template to any nodes that are "within context." This means that, if the template matches a particular element, all the immediate children of that element are within the current context and are subject to processing by that template. This can lead to some confusing results to the uninitiated.

Within a template, we can put plain text, HTML tags, or XML tags from some other namespace. Anything not in the XSL namespace (prefixed with xsl:) will be passed through untouched (though whitespace is collapsed). The content of an element from the source document is passed through to the output using <xsl:value-of>. XSL also has tags that work similarly to traditional looping and conditional programming language constructs, (<xsl:for-each>, <xsl:if>, <xsl-choose>, etc.) as well as provision for parameters (fixed value throughout the transformation), variables (changeable value) and "includes." This allows for powerful processing, including conditional logic based on external parameters.

XPath, even though it starts with an "X", is not yet another XML vocabulary. It is a method of specifying the path to a location in an XML document. It is similar to the path one would use to navigate a file structure, but XPath offers additional functionality. An *axis* can be specified, which basically means what direction (parent, child, sibling, etc.) or type (element, attribute, etc.) we should be looking at. The axis is combined with a *node test*. This is often simply the name of the element or attribute we're looking for, but there are also tests for text nodes, attribute nodes, comment nodes, etc. The results of each step in a path can be further selected from by means of a *predicate*. This can be a variety of different tests or functions (e.g. "contains()" for testing if one string contains another, "count()" for counting the occurrences of a node, etc.) enclosed in square brackets.

A carefully constructed XPath expression can be quite powerful, creating a specific node set for the XSLT processor to work with. The XSL logic constructs can then further refine the action to be taken for particular data. This includes calling new templates—even templates residing in another file—from within templates, allowing for very complex processing models.

Here is a simple example of an XSLT template (the test in the quotation marks are XPaths):

```
<xsl:template select="/songs/song">
```

```
      <xsl:value-of select="title"/>
</xsl:template>
```

This expects our XML document to have a root element called "songs," in which there should be at least one element called "song," which in turn should have an element called "title." This template will put the value of the title element into the *result set*. In other words, it outputs it. More specifically, it outputs the value of each title element in each song element that it finds in the songs root element.

So, we have some fairly simple rules that define what is an acceptable, or well-formed, XML document. We have DTDs and Schemas for defining the structural rules for specific types or vocabularies of XML documents. We have two different approaches or APIs for parsing an XML document. And we have XSLTs and XSL-FOs for converting an XML document into another file format. This is all well and good, you ask, but *what do we do* with our XML document?

Well, the first thing to bear in mind is that XML is nothing more than a means of storing information. You could also put your data into a relational database, in which case, a piece of information is identified—"marked up," if you will—by the field in which it resides. You can also move your data back and forth between database and XML file; something that is probably done every second of every day all over the world. Keeping your data in a database may actually make more sense than in an XML file if it consists of simple chunks of specific types and with specific relationships. However, where XML excels is with data that has less predictable relationships (one element <foo> may have ten <bar> children, while another <foo> may only have one <bar>… or none). And, where XML beats relational databases hands down, is with hierarchical data: there is no easy way to represent that <bar> is contained within <foo> in a database.

So, the answer to the "what do we do with this" question is "pretty much whatever your heart desires." XML, being eminently machine-readable, is gaining popularity for configuration files of all stripes. Scalable Vector Graphics (SVG) is an XML vocabulary for describing still or animated two-dimensional graphics and it is starting to gain popularity on the web and for creating desktop widgets. The Synchronized Multimedia Integration Language (SMIL) is used to combine text, graphics, audio and video into web-deliverable multimedia presentations. The XML User-interface Language (XUL) was developed by the programmers behind the Mozilla web-browser to describe GUI interface widgets. And businesses have begun to embrace XML as a method of standardizing data exchange: more and more industries are agreeing on standard vocabularies.

But ultimately, where XML is really king is on the web (it was developed by the World Wide Web Consortium, after all). XML is finally delivering on the original promise of HTML: separation of content and presentation. Web hosts can store content in XML files and transform that content to whatever presentation or delivery format the visitor desires. It is with this flexibility and adaptability in mind that the Apache Software Foundation developed Cocoon.

Cocoon is officially called "an XML publishing framework." However, that is a deceptively simple description for a complex and powerful collection of software. In terms of general purpose, Cocoon could be classified as a web server, though calling it such is a little misleading, since Cocoon cannot serve files on its own. Cocoon is not a standalone program, but rather a *servlet*. A servlet is a collection of java code that runs in a servlet "container," like JBoss, Jetty, or Apache's Tomcat. A servlet container is basically a web server that allows a host to install java applications which process web requests. Cocoon is one of these applications… on steroids.

Not surprisingly for something created by the Apache XML project, Cocoon makes ample use of XML; in the configuration files that control it, the content that it processes, the transformations it performs and

(perhaps) the material it ultimately serves to the user. On the configuration end, there is the "cocoon.xconf" file, in which the administrator can set the parameters for the overall Cocoon installation. Then, there are the sitemaps. These files, called "sitemap.xmap," are the heart of Cocoon. They allow one to specify which components, or java applications, are available and to chain these components together into pipelines, to which are mapped a particular URL pattern. So, when a web browser requests a page like "http://mysite.com/home", the sitemap will have an entry that specifies what source document to use and what components to process it with to serve this request.

Here's a sample pipeline entry from a sitemap:

```
<map:pipeline>
      <map:match pattern="home">
            <map:generate type="file" src="somefile.xml"/>
            <map:transform type="xslt" src="mytransform.xsl"/>
            <map:serialize type="html"/>
      </map:match>
</map:pipeline>
```

This says that when Cocoon receives a request for a page called "home," it should take the file called "somefile.xml," transform it with the XSLT called "mytransform.xsl" and serve it as HTML. Cocoon provides a stunning array of components to go in these pipelines (and more are in the works).

*Generators* (map:generate) provide input and, beyond the basic "file" input type, include the Directory Generator for generating directory listings, the JSP Generator for reading Java Server Pages, the Request Generator for reading request parameters, and several others.

*Transformers* (map:transform) manipulate the stream of data from the generator and, beyond an XSLT, can include the SQLTransformer for processing database requests, the CInclude and XInclude Transformers for aggregation, the LDAP Transformer for LDAP queries, and others.

*Serializers* (map:serialize) perform final manipulation steps on the data chain, including things like HTML and XML serializers, an SVG serializer for converting SVG code into JPEGs or PNGs, a FOP Serializer for converting XSL-FO files into PDFs, and others.

*Readers* simply pass data straight through without any processing (for images, CSS files, etc.). *Matchers* provide different ways of selecting a pipeline. The most common matcher is "wildcard," which uses full names or can use a * to replace pieces. But, there are also regular expression matchers, parameter matchers, cookie matchers, etc. *Selectors* are similar to matchers but allow the pipeline to switch between different chains of components depending on a parameter.

As if all these different types of components weren't already enough, there is yet one more, and it may be the most powerful. *Actions* allow you to incorporate sophisticated Java classes—including those of your own design—into a pipeline. Cocoon comes with several ready-made actions, including ones for handling database interaction, session authentication, form validation, and more. However, if none of these fulfill your requirements, you can write the Java code yourself and link it up with a simple entry in a configuration file.

The Cocoon developers also came up with their own server-side scripting language, of sorts, called *Extensible Server Pages* (XSP). Similar to JSPs (Java Server Pages), XSPs allow you to incorporate logic directly into a source XML file, by enclosing Java code in <xsp:logic> tags. This means that the XML content upon which your whole processing pipeline is based need not even be in final form until the web

request is received. Then, the java code is run and replaced with the resulting XML data, which is then transformed and manipulated by the rest of the pipeline.

It is clear that Cocoon offers a sophisticated and powerful web application framework. There is one more piece of the Cocoon puzzle, however, and it is useful in illustrating one of the key planks upon which the platform is built and ties in with a central theme of XML. This is the separation of content and presentation… and—in the case of a web-publishing framework like this—logic, as well. Just as it is a core idea of XML to separate content from presentation, it is similarly suspect to allow your logic to intermingle with your content.

To address this issue, Cocoon has *logicsheets*, which allow you to put the java code that would go into an XSP into a separate file. This is accomplished by writing an XSLT with the appropriate XSP tags inside templates. These templates are set to match tags with a namespace unique to that logicsheet. These custom tags are put into the source XML document wherever that logic is needed. After running an XSP and its logicsheet through the transformation engine, the chunks of code are put in the appropriate places in the XSP. The XSP can then be processed (the java bits are run and replaced) and converted into a plain XML file, which can then be fed into a full pipeline! Not only does this separate out the logic, but it also facilitates code reuse: you can put commonly used logic in a logicsheet and use it over and over again. And, of course, Cocoon comes with a number of ready-to-use logicsheets for common functions.

The previous few paragraphs only scrape the surface of the power and flexibility offered by Cocoon. And in its nearly every facet, the Extensible Markup Language plays a central role. Similarly, this entire paper touches on only a few of the emerging XML technologies and vocabularies, and in only a very cursory manner. There are other nascent "X-languages": *X-Link* allows for the creation of links between nodes, within a document and between documents and *XPointer* allows for references to chunks of a document without concern for node boundaries (kind of like labeling a selection in a document). There is an emerging standard for implementing *Native XML Databases* (NXDs) called XML:DB. NXDs allow for grouping XML files into "collections" and indexing their contents for fast data retrieval. And, of course, there is yet another XML language—*XQuery*—for querying these databases. Beyond these new additions to the XML manipulation toolbox, there are an ever-increasing number of standardized vocabularies.

What this complexity, along with the rapidly changing standards involved, indicates, however, is that the Extensible Markup Language is an area of importance and vitality. XML, essentially, is the Rosetta stone of future data manipulation. It is ultimately a long overdue universal standard for marking-up documents, and thereby quantifying and qualifying the data contained therein.