When less is more: a compact toolkit for parsing and manipulating XML

Designing a fast and small XML toolkit by applying object-oriented techniques

Graham Glass (graham-glass@mindspring.com)
CEO/Chief architect, The Mind Electric
March 2001

This article describes the design and implementation of an intuitive, fast and compact (40K) Java toolkit for parsing and manipulating XML -- Electric XML -- the XML engine of the author's company. It shows one way to apply object-oriented techniques to the creation of an XML parser, and it provides useful insight into API design. The source code for the non-validating parser described in this article may be downloaded and used freely for most commercial uses.

Contents:

Motivation and background

XML is finding its way into almost every aspect of software development. For example, SOAP, the rapidly emerging standard that is likely to replace CORBA and DCOM as the network protocol of choice, uses XML to convey messages between Web services.

When my company decided to create a high performance SOAP engine, we started by examining the existing XML parsers to see which would best suit our needs. To our surprise, we found that the commercially available XML parsers were too slow to allow SOAP to perform as a practical replacement for technologies like CORBA and RMI. For example, parsing the SOAP message in Listing 1 took one popular XML parser about 2.7 milliseconds.

Listing 1. A sample SOAP message

```
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/1999/XMLSchema">
   <SOAP-ENV:Body>
      <ns1:getRate xmlns:ns1="urn:demo1:exchange"
         SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
         <country1 xsi:type="xsd:string">USA</country1>
         <country2 xsi:type="xsd:string">japan</country2>
      </ns1:getRate>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

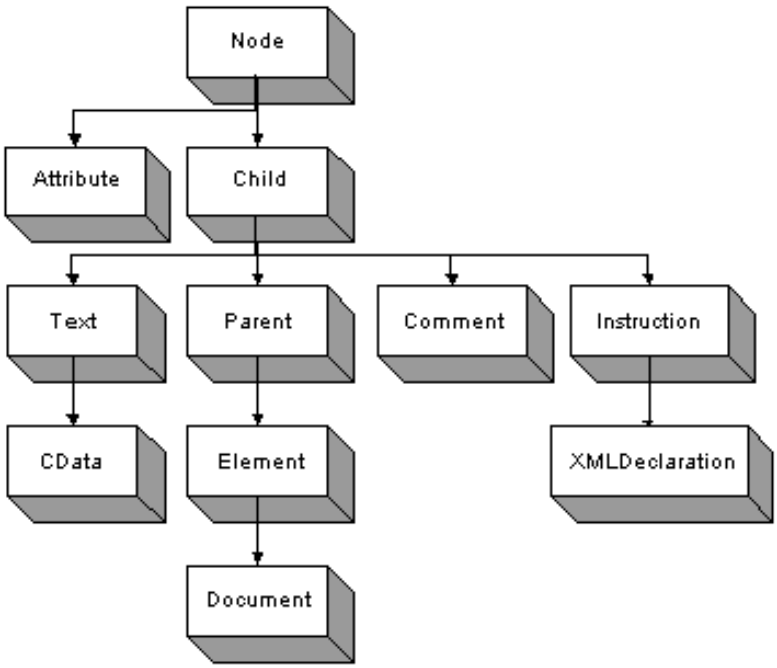
Since a round-trip RPC message requires both the request and the response to be parsed, using this popular parser would mean that a SOAP RPC call would generally take at least 5 milliseconds to process, and that doesn't include the network time or the processing time on either side. This doesn't sound too bad until you consider that a complete round-trip RPC message using RMI takes about 1 millisecond. So before giving up on ever building a SOAP engine that could compete against traditional technologies, we decided to experiment with building our own XML parser. (See Resources for how to download Electric XML, which developers may use without charge for most commercial and noncommercial uses.)

The rest of this article describes the design approaches that we took, and the happy results of our experiment.

Data structures

Every aspect of XML in the official specification is couched in terms of nodes, trees, and siblings. This strongly suggests a hierarchical object model with a common root node and subclasses that are specialized based on their specific behavior. Figure 1 shows the hierarchy that we came up with for representing the main aspects of an XML document.

Figure 1. Electric XML's hierarchy for an XML document



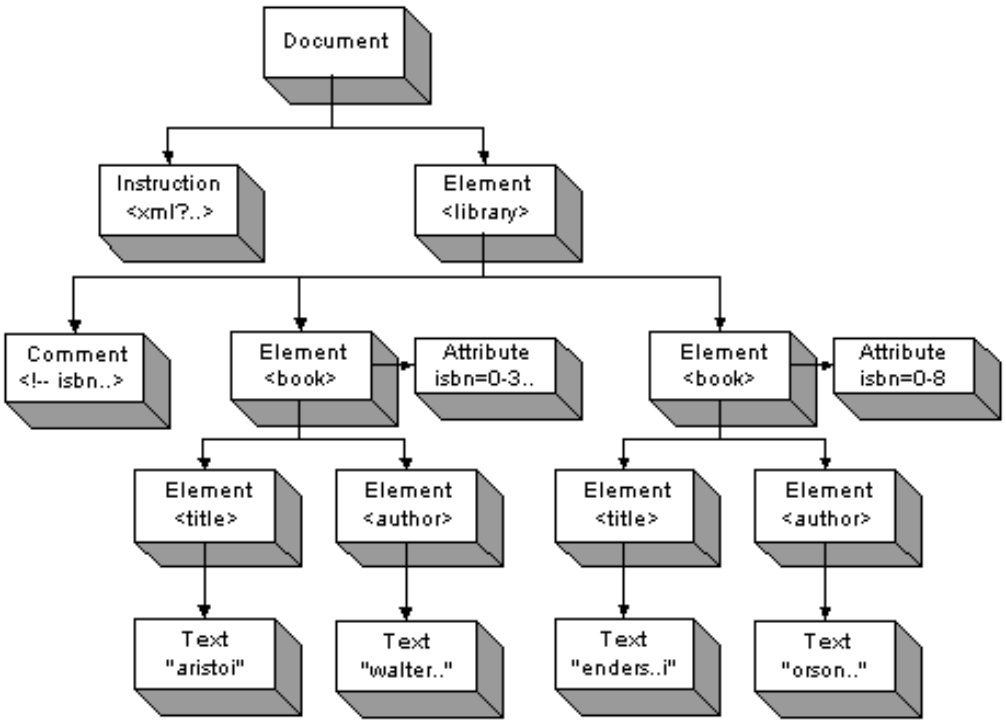
To see how a document would be mapped to elements of this hierarchy, consider the XML file in Listing 2.

Listing 2. Library1.xml, a sample xml file

```
<?xml version='1.0' encoding='UTF-8' ?>
<library>
  <!-- the isbn could have been an element -->
  <book isbn='0-312-85172-3'>
    <title>Aristoi</title>
    <author>Walter Jon Williams</author>
  </book>
  <book isbn='0-812-51349-5'>
    <title>Ender's Game</title>
    <author>Orson Scott Card</author>
  </book>
</library>
```

The contents of this file would be represented by a graph of nodes as follows in Figure 2.

Figure 2. Graphing the nodes in the library1.xml file



Based on this class hierarchy, several simple rules apply to any XML document:

- Each XML document has a single Document node.
- Each Document node has a single top-level element node, called the *root*.
- Each element node has a name and zero or more attribute nodes, and zero or more namespace definitions.
- Each parent node has zero or more child nodes.
- Child nodes with the same parent node are called *siblings*.

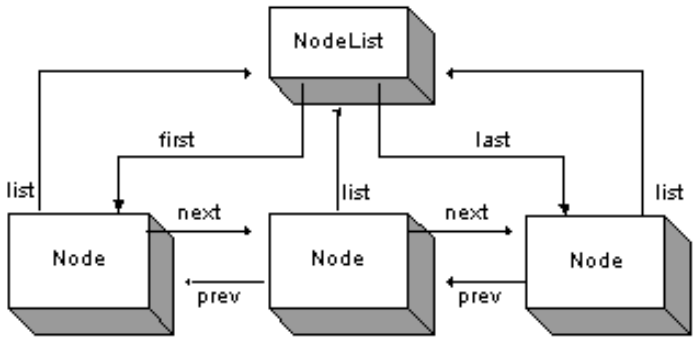
One of the first design challenges was to define the behavior of the root Node class.

We started by looking at the definition of the standard w3c.org.dom.Node interface, which defines the common behaviors of all standard DOM nodes. We found that the DOM Node defines several methods that do not apply to all nodes, such as getAttributes(), getLocalName() and getFirstChild(). This is not good object-oriented design, because it does not make sense to ask a Comment for its attributes, or a Text node for its children.

Instead of following the DOM approach, we decided to define a minimal Node that can optionally be a member of a doubly linked list of nodes. Each node has references to its previous sibling, its next sibling, the node list that it resides in, and to a small set of methods for accessing this information. Specialized methods for accessing children and names were pushed down to the specialized nodes such as Parent and Element.

We defined a NodeList class to represent a doubly linked list of Nodes, with a pointer to the first node and the last node in the list, as well as operations for adding, inserting and removing nodes. All lists of nodes, such as the list of children stored in a parent node or the list of attributes stored in an element node, are represented using a NodeList object.

Figure 3. A NodeList object and the nodes it tracks



We also decided that a node would implement a remove() method that instructs the node to remove itself from the XML document. Because different nodes can cause different things to occur when they are removed, remove() is polymorphic with a default behavior defined in Node. For example, when an attribute that represents a namespace is removed, it must update the namespace collection of the element that it's associated with. The decision to make a node responsible for its own removal became particularly important when we implemented XPath functionality, described later in this article.

Parsing XML
Before settling on the final parser design, we experimented with two approaches:

- Creating a completely separate parser class to read input from a Reader and create the appropriate nodes. The benefit of this approach is that the code for parsing is completely separated from the code for navigating and manipulating nodes.
- Pushing the responsibility for parsing into the node classes themselves. For example, the Document class knows how to parse a top-level XML document, the Element class knows how to parse an element, and the Attribute class knows how to parse an attribute.

We implemented the first approach as a single XMLParser class, which soon became rather complicated as it contained all the code for parsing all of the different node types. So then we created several parser classes that mirrored the original hierarchy, with names like DocumentParser, ElementParser, and AttributeParser. Each of these classes was responsible for creating a node of its associated type. By the time we'd finished this approach, the number of classes had almost doubled, and following the parsing process required you to continually switch back and forth between the node type and its associated parser type.

After implementing the first approach, we tried moving the parser code back into its associated node type, allowing each node class to parse and create an instance of itself. We found that only about 10% to 20% of each node's code was related to parsing, and that the resultant design was much cleaner and simpler than the decoupled approach. Note that this approach does not prevent a third party from using a traditional DOM parser to create the node structures, although it does give the built-in parser "preferred" status from a design perspective.

The other important implementation portion of the parser design is the lexical analyzer that breaks down the XML input stream into its pieces. The Lex class is relatively straightforward, with plenty of utility methods for getting small snippets of data in a flexible and efficient way. The most interesting problem when designing the lexical analyzer was related to exception handling. When an exception is encountered during parsing, a ParseException is thrown with an error message that tells you the line and character number of the error, as well as a small portion of the line where the error happened. We achieved this functionality by maintaining a small wraparound buffer in the lexical analyzer, which keeps a running window of the last 60 or so bytes of input. When an exception is generated, the contents of this buffer can be used to conveniently display the cause of the exception.

Printing XML
Once we'd adopted the philosophy that every node knows how to parse itself, it was natural to apply this viewpoint to other aspects of design, such as printing. The Node class defines the default method for writing itself to a Writer, and each subclass overrides this method to do the appropriate thing. Another side effect of this design is that displaying an XML document in a nicely indented fashion can be achieved by simply invoking toString() on the Document object.

Navigating XML
The XML data structures are fairly straightforward. Each parent contains a NodeList of child nodes, and each child has a reference to its parent. In addition, each element contains a NodeList of attributes and a hash table of namespaces. These structures are initialized during the parsing process, and made available via methods like Child.getParent() and Element.getChildren().

An interesting design question was how to return an ordered collection of nodes from methods like getElements() and getChildren().

One approach is to return a JDK collection such as a List. If we made the underlying data structure a List, we could simply return a reference to this structure, and the operation would be very fast. But the user could then manipulate the XML document directly through this List and potentially insert completely inappropriate elements such as an Employee object. If we returned a *copy* of the List, it would cause severe performance penalties. In addition, it would force users of the parser to use JDK 1.2 or to download the 250K collections add-on library.

Another approach is to return a plain enumeration that only supports nextElement() and hasMoreElements(). This is simple, efficient, and familiar to most developers, but casting each call to nextElement() to a node of the appropriate type is tedious.

In the end, we chose a third approach, which was to return a type-specific enumeration that supports the standard enumeration methods as well as additional methods for accessing elements in a typesafe and flexible way. For example, Attributes.next() returns the next attribute in the list, and Elements.next() returns the next element in the list. Listing 3 parses the example library1.XML file, accesses its root, and then iterates through each Child in a typesafe way.

Listing 3. Parsing the example library1.xml file

```
package examples.xml;

import electric.xml.*;

public class Parse1
  {
  public static void main( String[] args )
    {
    try
      {
      Document document = Document.parseFile( "library1.xml" ); // parse file
      System.out.println( "document\n\n" + document + "\n" ); // display document

      Element root = document.getRoot(); // get root element node
      System.out.println( "root\n\n" + root + "\n" );

      // iterate through all child nodes
      for( Children children = root.getChildren(); children.hasMoreElements(); )
        {
        Child child = children.next();
        String type = child.getClass().getName();
        System.out.println( "child " + type + "\n\n" + child + "\n" );
        }
      }
    catch( ParseException exception )
      {
      System.out.println( "exception: " + exception );
      }
    }
  }
```

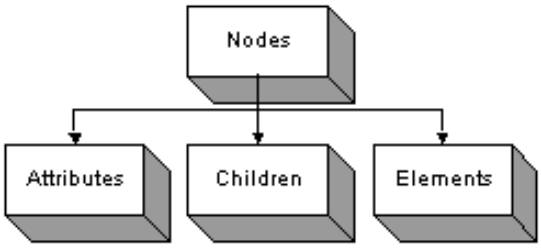
The output from running examples.xml.Parse1

```
> java examples.xml.Parse1
document

<?xml version='1.0' encoding='UTF-8' ?>
<library>
  <!-- the isbn could have been an element -->
  <book isbn='0-312-85172-3'>
    <title>Aristoi</title>
    <author>Walter Jon Williams</author>
  </book>
  <book isbn='0-812-51349-5'>
    <title>Ender's Game</title>
    <author>Orson Scott Card</author>
  </book>
</library>

root

<library>
  <!-- the isbn could have been an element -->
  <book isbn='0-312-85172-3'>
```

```
    <title>Aristoi</title>
    <author>Walter Jon Williams</author>
  </book>
  <book isbn='0-812-51349-5'>
    <title>Ender's Game</title>
    <author>Orson Scott Card</author>
  </book>
</library>

child electric.xml.Comment

<!-- the isbn could have been an element -->

child electric.xml.Element

<book isbn='0-312-85172-3'>
  <title>Aristoi</title>
  <author>Walter Jon Williams</author>
</book>

child electric.xml.Element

<book isbn='0-812-51349-5'>
  <title>Ender's Game</title>
  <author>Orson Scott Card</author>
</book>
> _
```

Figure 4 shows the hierarchy of typesafe enumerations that we built.

Figure 4. The hierarchy of typesafe enumerations



The Nodes superclass element contains a reference to a NodeList and a reference to the current node in that list. Every call to next() advances the current pointer to its next sibling. Returning an enumeration of all the children of a parent is as simple as returning a new instance of Children that contains a reference to the children NodeList in the parent.

Iterating through a list of the elements in a parent is a little more tricky because the single children NodeList can contain a mix of Element, Text, CData, Instruction, and other Child node types. To achieve the illusion of iterating through a homogenous list, the Elements enumeration skips all elements in its NodeList that are not instances of Element. This has an insignificant performance impact.

To simplify common operations like getting the first element with a specified name, or an enumeration of elements with a specified name, we added methods like getElement(String name) and getElements(String name).

We used a crafty little object-oriented design technique to implement methods like getDocument() and getRoot(). The default implementation of getDocument(), defined in Child, looks like this:

```
public Document getDocument()
  {
  return (parent == null ? null : parent.getDocument());
  }
```

Document overrides that method as follows:

```
public Document getDocument()
  {
  return this;
  }
```

Any Smalltalk programmers among you will recognize this technique from the Smalltalk definitions of methods like ifTrue: and ifFalse:.

Manipulating XML
Most of the methods for manipulating XML documents, such as addChild(), addElement() and removeElement() were implemented as pass-through calls to the linked list manipulation methods in NodeList. In a twist, we allow each element to perform a specialized action when it's added or removed from its list. For example, when an Attribute that represents a namespace is removed from an element, its remove() method automatically updates the element's namespace collection. This approach pushes the logic for maintaining a consistent document into smart nodes instead of centralizing it into a single smart document.

The example in Listing 4 shows how to use the object model APIs to create the contents of library1.xml from scratch.

Listing 4. Building the contents of library1.xml using the Electric XML APIs

```
package examples.xml;

import electric.xml.*;

public class Manipulate1
  {
  public static void main( String[] args ) throws Exception
    {
    // create empty document
    Document document = new Document();

    // add standard header
    String version = "version='1.0' encoding='UTF-8' ";
    document.addChild( new Instruction( "xml", version ) );

    // create root
    Element library = document.addElement( "library" );

    // add comment
    library.addChild( new Comment( " the isbn could have been an element " ) );

    // add first book
    Element book1 = library.addElement( "book" );
    book1.setAttribute( "isbn", "0-312-85172-3" );
    book1.addElement( "title" ).setText( "Arisoi" );
    book1.addElement( "author" ).setText( "Walter Jon Williams" );

    // add second book
    Element book2 = library.addElement( "book" );
    book2.setAttribute( "isbn", "0-812-51349-5" );
    book2.addElement( "title" ).setText( "Ender's Game" );
    book2.addElement( "author" ).setText( "Orson Scott Card" );

    // display document
    System.out.println( "document\n" + document );
    }
  }
```

The output from running examples.xml.Manipulate1

```
> java examples.xml.Manipulate1
document
<?xml version='1.0' encoding='UTF-8' ?>
<library>
  <!-- the isbn could have been an element -->
  <book isbn='0-312-85172-3'>
    <title>Arisoi</title>
    <author>Walter Jon Williams</author>
  </book>
  <book isbn='0-812-51349-5'>
    <title>Ender's Game</title>
    <author>Orson Scott Card</author>
  </book>
</library>
> _
```

Namespaces

One of the trickiest aspects of the design was namespace support. Without the constraints of the DOM, we had the luxury of designing the APIs with namespaces in mind, so we wanted to make the integration as seamless as possible. Since both elements and attributes can have a fully qualified name, we decided to define a Name object that contained two strings, one for the original name and one for the fully resolved name. The original is used when printing a document, and the fully resolved name is used when performing comparisons. The Name class defines methods for accessing the various aspects of a qualified name, such as getLocal(), getPrefix() and getExpanded().

In order to allow developers to access a namespace-qualified element in an intuitive way, we had to implement dynamic namespace resolution. For example, consider the XML file in Listing 5 that uses namespaces.

Listing 5. An XML file with namespaces, library2.xml

```
<?xml version='1.0' encoding='UTF-8' ?>
<libns:library xmlns:libns='http://www.lib.com/'>
  <!-- the isbn could have been an element -->
  <libns:book libns:isbn='0-451-19114-5' xmlns='http://www.book.com/'>
    <title>Atlas Shrugged</title>
    <author>Ayn Rand</author>
  </libns:book>
  <libns:book libns:isbn='0-465-02656-7' xmlns='http://www.book.com/'>
    <title>Godel, Escher, Bach</title>
    <author>Douglas Hofstadter</author>
  </libns:book>
</libns:library>
```

To access the first book element, the most natural approach is allow a developer to write: root.getElement("libns:book"). This means that the code has to notice that the argument includes a namespace qualifier, expand it to its full name using the current set of namespaces, then compare the fully expanded form against the other fully expanded names in the document. This approach allows developers to access elements without providing the fully expanded name, which in this case would be http://www.lib.com/book. To search for an element using a fully qualified name -- which might contain a colon (:) character, you can surround a name within single quotes, which inhibits namespace expansion. For example, you can write root.getElement('"http://www.lib.com/book"').

When a namespace declaration attribute is added or removed, the parser is smart enough to detect that it represents a namespace, and automatically updates its associated element. This allows a developer to dynamically add and remove namespaces in a natural fashion. Namespaces are maintained within an element as a hash table of key/value pairs.

Listing 6. Building the contents of library2.xml using the Electric XML APIs

```
package examples.xml;

import electric.xml.*;

public class Namespace1
  {
  public static void main( String[] args ) throws Exception
    {
    Document document = new Document();
    String version = "version='1.0' encoding='UTF-8' ";
    document.addChild( new Instruction( "xml", version ) );

    Element library = document.addElement();
    library.setAttribute( "xmlns:libns", "http://www.lib.com/" );
    library.setName( "libns:library" ); // add after libns: was declared

    library.addChild( new Comment( " the isbn could have been an element " ) );

    Element book1 = library.addElement( "libns:book" );
    book1.setAttribute( "libns:isbn", "0-312-85172-3" );
    book1.setAttribute( "xmlns", "http://www.book.com/" ); // default namespace
    book1.addElement( "title" ).setText( "arisoi" );
    book1.addElement( "author" ).setText( "walter williams" );

    Element book2 = library.addElement( "libns:book" );
    book2.setAttribute( "libns:isbn", "0-812-51349-5" );
    book2.setAttribute( "xmlns", "http://www.book.com/" ); // default namespace
    book2.addElement( "title" ).setText( "ender's game" );
    book2.addElement( "author" ).setText( "orson scott card" );

    System.out.println( "document\n" + document );
    }
  }
```

The output from running examples.xml.Namespace1

```
> java examples.xml.Namespace1
document
<?xml version='1.0' encoding='UTF-8' ?>
<libns:library xmlns:libns='http://www.lib.com/'>
  <!-- the isbn could have been an element -->
  <libns:book libns:isbn='0-312-85172-3' xmlns='http://www.book.com/'>
    <title>arisoi</title>
    <author>walter williams</author>
  </libns:book>
  <libns:book libns:isbn='0-812-51349-5' xmlns='http://www.book.com/'>
    <title>ender's game</title>
    <author>orson scott card</author>
  </libns:book>
</libns:library>
> _
```

XPath

The last thing to go into the design was support for a useful subset of XPath. This was the most enjoyable part of the process, as it

yielded functionality that greatly simplified the code of our distributed computing platform. Rather than providing a separate set of APIs for XPath manipulation, we decided to allow most of the existing get() and remove() APIs to accept an XPath expression, as shown in the example in Listing 7.

Listing 7. Extracting information from library1.xml with XPath expressions

```
package examples.xml;

import electric.xml.*;

public class XPath1
  {
  public static void main( String[] args ) throws Exception
    {
    Document document = Document.parseFile( "library1.xml" );

    // get all books anywhere in the document with the specified isbn attribute
    Elements books = document.getElements( ".../[@isbn='0-812-51349-5']" );

    while( books.hasMoreElements() )
      System.out.println( "book\n" + books.nextElement() + "\n" );

    // get all book authors with the specified text value
    Elements authors = document.getElements( "/book/author[text='Orson Scott Card']"
);

    while( authors.hasMoreElements() )
      {
      Element author = authors.next();
      System.out.println( "author\n" + author + "\n" );
      System.out.println( "book\n" + author.getElement( ".." ) );
      }
    }
  }
```

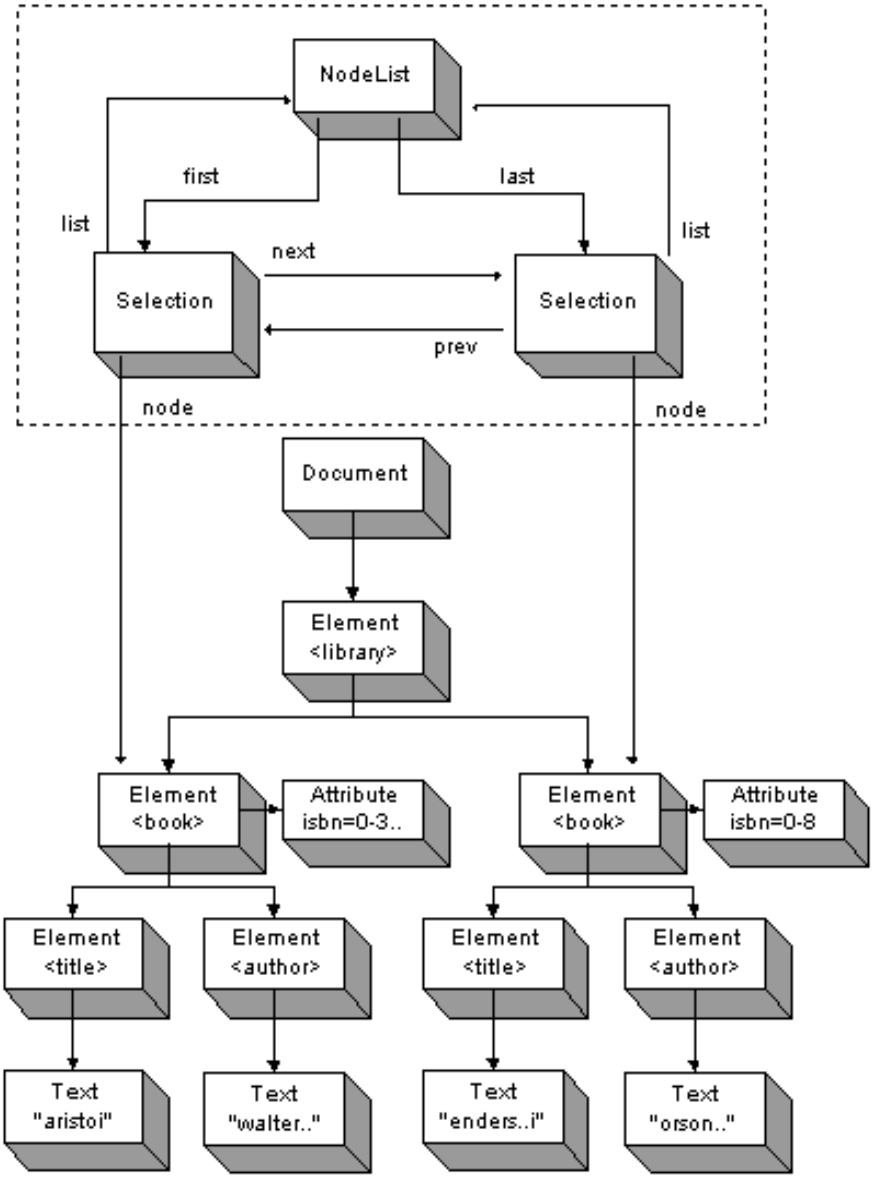
The output from running examples.xml.XPath1

```
> java examples.xml.XPath1
book
<book isbn='0-812-51349-5'>
  <title>Ender's Game</title>
  <author>Orson Scott Card</author>
</book>

author
<author>Orson Scott Card</author>

book
<book isbn='0-812-51349-5'>
  <title>Ender's Game</title>
  <author>Orson Scott Card</author>
</book>
> _
```

The main design trick we used for XPath involved the addition of a new Node type called Selection. When you perform an XPath query, you can get back more than one node, so the most natural way to represent the result is using a NodeList. A NodeList points to the first

node in the list and the last node in the list, and requires each node in the list to point to the next and previous nodes. Since the nodes that are part of an XPath query result already have existing siblings, their sibling pointers cannot be modified. So XPath creates a NodeList comprised of freshly created Selection nodes, each of which points to a node in the XPath result set.

Figure 5. The selection list returned by an XPath query



The selection nodes can be iterated over using the enumeration classes that were introduced earlier. All enumerations access a node by calling getNode(), where the default implementation returns this, and a selection node returns the node that it points to.

There is an interesting spin-off from the initial decision to make all nodes polymorphically implement remove(): It's very easy to select and then remove an arbitrary set of nodes using an XPath expression, as shown in the example in Listing 8.

Listing 8. Selecting and removing an arbitrary set of nodes (parsing library2.xml)

```
package examples.xml;

import electric.xml.*;

public class XPath2
  {
  public static void main( String[] args ) throws Exception
    {
    Document document = Document.parseFile( "library2.xml" );

    // remove all authors
    document.getElements( "libns:library/libns:book/author" ).remove();
    System.out.println( "document after removing all authors\n" + document );
    }
  }
```

The output from running examples.xml.XPath2

```
> java examples.xml.XPath2
document after removing all authors
<?xml version='1.0' encoding='UTF-8' ?>
<libns:library xmlns:libns='http://www.lib.com/'>
  <!-- the isbn could have been an element -->
  <libns:book libns:isbn='0-451-19114-5' xmlns='http://www.book.com/'>
    <title>Atlas Shrugged</title>
  </libns:book>
  <libns:book libns:isbn='0-465-02656-7' xmlns='http://www.book.com/'>
    <title>Godel, Escher, Bach</title>
  </libns:book>
</libns:library>
> _
```

A UML view
For completeness, Figure 6 shows the UML for the Electric XML data structures. Use the diagram to help you browse the source code, if you choose to download it.

Summary
Our initial experiments indicated that we could build a small, fast, intuitive toolkit for parsing and manipulating XML documents that would allow our distributed-computing platform to approach the performance of existing traditional systems. We decided to complete the parser and make it available to the developer community, partly to earn some good karma, and partly to demonstrate that powerful toolkits do not need to be large or complex. I personally yearn for the days of Turbo Pascal when companies shipped full-blown development and runtime environments that took up just 30K!

The main design decisions were:

- Selecting a hierarchy for the object model that fitted naturally with the tree structure of an XML document
- Pushing the knowledge of how to parse, print, and react to removal into each "smart" node
- Using a Name object to represent a namespace-qualified name
- Allowing get and remove operations to accept an XPath expression
- Using selection nodes to keep track of XPath result sets

The resulting parser achieves the goal of processing a SOAP message about as quickly as RPC over RMI. Table 1 shows a comparison of parsing the sample SOAP message in Listing 1 with the production release of Electric XML and with a popular DOM parser 10,000 times and calculating the average time to parse the document:

Table 1. Parsing speed of Electric XML and a popular DOM parser compared (running Listing 1)

Parser	Time to parse
Popular DOM-based parser	2.7 milliseconds
Electric XML	0.54 milliseconds

I hope that the article provides useful examples of object-oriented design in action, as well as an instance of the adage "less is more." I hope also that Electric XML might prove useful for your XML development efforts.

Resources

- Download the source code of the Electric XML parser from The Mind Electric. It is a small file, less than 200K. Find installation and use instructions in the user guide.
- Need background? Look into the XML specifications mentioned in this article:
  - XML 1.0
  - XML Namespaces
  - XPath
- If you enjoyed this look at the design decisions behind a parser, you might want to check out Michael Kay's walkthrough of his open-source XSLT processor, Saxon.
- Look for other parsers in the dW XML zone's Tools and Code parser listings.
- Need an intro to XML programming with Java? Check out the dW tutorial XML Programming in Java.

About the author

Graham Glass (graham-glass@mindspring.com) is founder, CEO, and Chief Architect of The Mind Electric, which designs, builds, and licenses forward-thinking distributed computing infrastructure. He believes that the evolution of the Internet will mirror that of a biological mind and that architectures for helping people and businesses to network effectively will provide insight into those that wire together the human brain.

Prior to founding The Mind Electric, Graham was the chairman, CTO and co-founder of ObjectSpace, a Dallas-based company specializing in business-to-business integration. Graham was also the founder of ObjectLesson, a company that provided training in leading-edge technologies. He authored two books for Prentice Hall on the subjects of UNIX and STL, and he is a popular public speaker known for his enthusiasm and clear explanations of emerging technologies.