3-2004

Natural XML for data binding, processing, and persistence

George K. Thiruvathukal
Loyola University Chicago, gkt@cs.luc.edu

Konstantin Läufer
Loyola University Chicago, klaeufer@gmail.com

Recommended Citation

G. Thiruvathukal and K. Läufer, "Natural XML for data binding, processing, and persistence," Computing in Science & Engineering, vol. 6, no. 2, pp. 86-92, Mar. 2004.

Editors: Paul F. Dubois, paul@pfdubois.com
George K. Thiruvathukal, gkt@nimkathana.com

NATURAL XML FOR DATA BINDING, PROCESSING, AND PERSISTENCE

By George K. Thiruvathukal and Konstantin Läufer

THE LAST ISSUE'S INSTALLMENT OF THIS DEPARTMENT PRESENTED AN OVERVIEW OF XML AND ITS POTENTIAL FOR COMPUTATIONAL SCIENCE. IN THIS ISSUE, WE'LL EXPLORE what you need to do to incorporate XML directly into your application. Our exploration involves the use of a standard parser to automatically build object trees entirely from application-specific classes. This discussion very much focuses on object-oriented programming languages such as Java and Python, but it can work for non-object-oriented languages as well. The ideas in this article provide a glimpse into our Natural XML research project. All the code examples in this article appear in full at http://content.cs.luc.edu/projects/cise/code/march_april_2004.

Practical Use of XML

Practical uses of XML, especially in computational science, require the ability to process XML in languages other than XML itself. Doing so requires computational scientists to learn something about parsing.

Parsing introduces new challenges when it comes to scientific computing. First and foremost, XML assumes an underlying tree model to represent hierarchical information. Although computer scientists naturally rely on trees to represent dynamic, hierarchical structures, computational scientists tend to shun them in favor of arrays and other static(ally allocated), flat data structures. Today, even pure computer scientists tend to avoid messing directly with concrete tree structures, opting instead to work with arbitrary, high-level collections of objects such as sets or maps, which may or may not be implemented as trees. In languages such as C++, Java, and Python, programmers need only define their own item classes and arrange them in any way desired (via composition), most often by taking advantage of the powerful collection class libraries provided by their implementation language of choice.

In practice, using XML requires following this life cycle:

- Read one or more input XML files.
- Perform some custom (or ad hoc) processing.
- Write XML output (often in the same format).

As you read this description, pause momentarily to think how scientific computing applications currently work, and then take heart: the tradition of computing lives on, at least in terms of the life cycle. You read data files, process them, and write output files every day. The difference here is three little letters: X-M-L. Unfortunately, it's still the old model in many ways: input, processing, output.

Fortunately, these three letters add something we didn't have before: a more precise way of structuring information and ensuring that it is well defined.

XML Parsing

Parsing is the terminology that linguists and programming language designers use to describe the notion of recognizing phrases in any given language. In programming languages, compilers break the input (your source code) into lexical units (called tokens), which a parser then checks to ensure compliance with the language's grammatical rules.

The XML community tends to use the term parsing somewhat generically and loosely. For example, one kind of parser called DOM (Document Object Model) reads an XML document and produces a tree representation of it. Another parser, called SAX (Simple API for XML), reads the XML document and generates events as it works its way through the document. These events don't actually do anything unless acted upon by user-defined code. In the case of DOM parsers, a good portion of what compiler designers call *semantics* is addressed via a canonical abstract syntax tree format.

All XML parsers can check syntax (via the DTD or XML schema, both of which the previous issue introduced), but they can limit their checking to basic syntax only (the so-called well-formedness rules). The term parsing does, in fact, apply—at the user's sole discretion.

Components

In recent months, we (the authors) have been looking into ways to map XML to more modern programming languages, such as Java and Python, and even to aging ones (such as C, C++, and Fortran). As noted in the sidebar, the DOM specification presents a very troubling interface to scientific programmers (really, to programmers in general). Its one-size-fits-all approach is advocated by the W3C and is now in its third revision. From a usage perspective, the programming interfaces are numerous and complex.

DOM's fundamentally homogeneous tree representation means that the major problem of working with it is duplicating the tree to match the data structures actually used in the application. For scientific programming, this approach simply is not feasible. You want to create as few data structures as possible, not two copies of essentially the same data structure. A second but equally significant problem is the desired goal to support (easily) the process of saving the data structure in XML format. Today, programmers usually perform this round-trip process via ad hoc code-generation techniques (OK, print statements). The resulting XML, unless done very carefully, is not equivalent to the XML used when creating the structure.

Java and Visual Basic are two language technologies in which the term *component* has been introduced. In Java's case, components are defined via the JavaBeans architecture. The notion of components provides a natural mapping for XML in particular. A component has several characteristics that we can apply to virtually any language in which we choose to write XML applications.

At the core of components is the notion of *properties*. A property is a special variable (usually a string) that can be set or accessed, subject to visibility modifiers. The JavaBeans framework integrates properties by having `set/get` methods to allow the property to be accessed or modified, respectively. A `set` method's presence implies that the property is mutable; a `get` method's presence implies the property can be read. Write-only properties are rare, but eliminating either a `get` or `set` method allows exclusivity of reads or writes.

As authors, we're always wary of writing more than a few paragraphs without presenting a code example, so here's a demonstration of how this works. Let's say that Particle.java shows a user application class (a structure for C/Fortran readers) to model a particle. For each attribute found in a `Particle`, a `set/get` method exists for each property. For now, don't be alarmed about the large number of methods: we'll be able to generate all this code from the XML schema definition.

The `Particle` class contains several other methods—in particular, `getElementName()`, `getAttributes()`, and `getChildren()`. All these methods must be implemented to go from the application class back to XML. In most cases, we can generate the methods, but they're often trivial to implement. These methods also demonstrate a powerful concept in object-oriented languages known as *interfaces*. Think of an interface as a contract that a class promises to fulfill. It is not an altogether new concept, but in object-oriented programming languages with strong typing, interfaces provide a mechanism for bridging user-defined entities to more generic algorithms. As we'll see, these interfaces provide the glue that lets us generate perfect XML. The programmer does nothing more than implement the interface functions, usually with a single line of code per method.

Each interface method does the following:

- `String getElementName()` is the XML element name for any instance of this class. We can generate any element name as desired, provided we follow the XML naming conventions.
- The `Map getAttributes()` method invokes any attributes from this class that will be exposed when appearing in XML. A map should be familiar to long-time readers of this column; it is similar to the notion of a dictionary in Python, which is an associative data structure. Any data structure that implements the `Map` interface can be returned.
- Any nested instances that should appear in the XML document model should be returned via the `List getChildren()` method. The notion of a list also should be familiar to this column's long-time readers; it exposes `array` and `list` operations. Any data structure that implements the `List` interface can be returned.

Once the programmer implements these three methods, we can easily generate an XML document starting with the root of the internal object tree and recursing through its children.

Collections

Think of collections as data structures on steroids. Undoubtedly, we have piqued your interest in the classes named `List` and `Map`. All modern programming languages possess native collections (not necessarily created equal, of course). In Java, the names `List` and `Map` are actually interfaces themselves, which can be used to refer to any type of collection that implements the interface. `List` is nothing more than an interface that refers to the expected operations we can perform on a list-like collection—for example, we can use the `add(Object object)` interface method to append an item to a list. Java provides several concrete list types,

Data-Binding Technologies for XML Documents

Let's take a look at some of the better-known data-binding technologies for XML documents: DOM, JDOM, and JAXB. We will see how they fare in comparison with our Natural XML approach.

Document Object Model

XML's Document Object Model (DOM) is a standard set of procedural interfaces for working with a tree representation of XML. Without going into too much detail, DOM is an example of committee work at its best. To create it, the W3C designed a collection of interfaces that can be mapped to any programming language. However, the programmer must master dozens of interfaces that present several complications (such as tight coupling, in which you must be careful to call methods in a certain order to avoid unexpected results).

The worst part of working with DOM is that, for all practical purposes, the tree it creates must be rewritten to match data structures in the application. Developing such code is tedious and complex, and from our teaching experience, most computer-science students find it difficult. After rewriting the tree, you often have to do the reverse: translate the application-specific data structures back to DOM format to allow for automatic code generation. (The good thing about DOM, though, is that you can be reasonably assured of being able to generate valid XML from it.)

JDOM

The Java DOM (JDOM) class library is similar to DOM; it purports to be a natural way for Java programmers to write XML applications. Although it's a step in the right direction, it does not follow the ideas of components and collections described in the main text, requiring prospective users to master interfaces similar to the DOM specification's.

Here is an example from the W3C's tutorial of some code to generate XML:

```
SAXBuilder builder = new SAXBuilder();
Document inputDoc = builder.build(url);

// Generate code for <root>This is the
        root</root>
Document outputDoc = new Document();
Element root = new Element("root");
root.setText("This is the root");
outputDoc.addContent(root);
XMLOutputter outputter = new XMLOutputter();
outputter.output(outputDoc, System.out);
```

Natural XML

By contrast, our Natural XML framework allows applications to integrate XML code-generation capabilities directly into existing code, virtually eliminating the need for scientific programmers to write such code. By implementing the `ContainedContent` interface, the programmer simply defines methods that will automatically generate XML.

JDOM is limited to Java; we're already working on Python, C#, and C++ versions of Natural XML, which can support all these languages easily.

JAXB

JAXB is another Java-based framework designed to support data binding. It would take a long time to cover the details here, since the specification spans hundreds of pages—JAXB clearly has a steep learning curve for developers. Programmers must first master an XML-based language for describing data structures in the program. A binding compiler generates stubs and skeletons, which the user can then implement for any application-specific functionality.

A significant problem with the framework appears to involve philosophy. To use it, the programmer must rework everything into Java's view of the world: collections, serialization (something that Natural XML provides innately), and Sun-proprietary tools. A good part of the documentation addresses why XML/Schema is vastly better to DTDs, but we're not convinced. Scientific applications often use a single namespace and, therefore, need nothing more than a DTD.

Java, Java, Java?

The vast majority of interesting work we have seen with XML and data binding seems to be happening in the Java community. Although some of the best basic XML tools exist for C, C++, and C#, the Java community appears to be one of the few places where significant work is going into integrating XML and non-XML languages. We consider this a problem because it limits the options available for leveraging XML in one's software.

Our main effort with the Natural XML project has been to prototype core concepts using a modern programming language like Java, but the needs we have addressed are hardly unique to Java. They can be incorporated into languages that support (minimally) structured types and dynamic data structures. Stay tuned to our project pages for more information.

some of which have very efficient implementations. The `ArrayList` is a concrete list type built on a contiguous array of values. Because `ArrayList` implements the list interface, however, it can do all the operations normally permitted on lists. From a programming viewpoint, this is a wonderful development: you can choose a particular list implementation and later replace it with another (say, a dynamic linked list) without breaking the interface. The end result will be the same, with the biggest difference appearing in the amount of memory consumed and the relative performance, depending on which operations on the list are used most.

Collections are essential for dealing with XML. Unfortunately, XML standards say nothing about them. Instead, the XML DOM specification presents an assortment of interfaces, leaving the choice of collections entirely up to the developer. In our Natural XML approach, we give the developer freedom to map to arbitrary collection classes, such as lists and maps (known in Java as `List` and `Map` interfaces). When the user wants to do something XML-specific, we require him or her to follow certain interfaces. In most cases, these interfaces contain straightforward code that can be added easily to any user-defined class.

Consider the `InitialConditions` class, which contains one or more `Particle` instances. Similar to `Particle`, this class also implements the `ContainedContent` interface described in the preceding section. Note that this class differs from `Particle` because it is a *container* of `Particle` objects. The programmer simply creates an `ArrayList` of particles, which will maintain references to all the contained `Particle` objects read in from XML. The `getChildren()` method simply returns a reference to this list. What we have in this class is essentially an idiom that can be used (like most idioms) over and over again, almost without thinking about it.

Introspection: Assembling Structures from User-Defined Classes

Everything we've discussed thus far has been helpful only for seeing how we would build our own data structures using Java class definitions. We use the `InitialConditions` class (a container) to keep an array-based collection of `Particle` instances, but other classes in the application exist (one for each of the XML element names shown in last issue's particle.xml file, see www.thiruvathukal.com).

For this to work, though, we need to write a parser that can read the XML document and construct a correct class instance when it encounters an element definition. When the parser sees `InitialConditions` in the `Nbody` XML, the parser must create an instance of the `InitialConditions` class; when it sees `Particle`, a `Particle` is created. Furthermore, each `Particle` must be added to the `InitialConditions` instance (in its array list).

Depending on the programming language, accomplishing this task is either trivial or requires special tools. Luckily, Java (Python and C#, too) provides facilities for *introspection*. Here is where things can get a bit weird, but please bear with us. Introspection is a program's ability to answer questions about itself at runtime—both Java and Python have elaborate interfaces for doing so. Introspection helps answer the following types of questions:

1. Given a reference to an object x, what is its class? `Class c = x.getClass()` provides the answer.
2. Given a class, what methods are available? `Method m = c.getMethod(...)` provides the answer.
3. Given a method and an object, invoke it. `m.invoke (object, parameters)` does the job.

These introspective capabilities are valuable in a number of situations. In XML, we simply don't know what type we want to create or what methods to invoke until we've actually read the document. For example, when we see the `InitialConditions` element, we must look up the class that handles this element (it need not have the same name) and create an instance of it. The same is true for a `Particle` element and all other XML element names.

When a contained element must be linked to its so-called parent (or container), we need the ability to call the correct method. In our Natural XML framework, we attempt to locate an appropriate `add` method; for example, we add a `Particle` to `InitialConditions` by invoking the `addParticle(Particle p)` method on an instance of `InitialConditions`.

This discussion might seem a bit abstract, but luckily for prospective developers, you'll never have to think about it. You simply implement the correct method names, and it just works.

The Actual SAX Parser

All the discussion up to this point naturally leads us to the actual SAX parser. The SAX parser works in the following way. First, it separates the XML document into tokens; these are items such as `<Particle>`, `</Particle>`, attributes, string values, and entities, which are the special symbols beginning with `&`, such as `&gt`, and `&lt` that let you use the < and > symbols literally in your XML document. These tokens isolate significant parsing events. Although many events exist, we'll just focus on the essentials here:

- `startDocument()` is generated before the document's root element is processed.
- `startElement()` is generated for every element in the

Cafe Dubois

SCHOLARZHEIMERS

I was reviewing a paper for publication recently and noted that the author seemed unaware of some prior literature in the area. When I make such a complaint to an author, I like to cite them the papers in question. In this case, I didn't recall precisely where I had seen the papers, so I tried an online search. Nothing turned up.

Then it dawned on me: these papers appeared circa 1990. They aren't online, so to an increasing portion of the scholarly population, they don't exist.

If you have this problem, which I hereby dub "scholarzheimers," look for a big building on your campus that says L-I-B-R-A-R-Y. Inside, you will find a weird, lonely nerd called a reference librarian. (This person was just as weird before the Internet, by the way.) Ask reference librarians nicely for assistance, and they will help you find stuff from the past. The good part is that you'll be practically the only person who knows it. If you watch a lot of MTV and lose your moral compass, you could even resubmit these lost works to journals as your own work. Nobody will ever know. Just pick one with a young editor.

The ACM Portal

That said, the situation is a little better for pure computer science thanks to the ACM Portal. (Mentioning the ACM Portal in an IEEE Computer Society-cosponsored magazine is sort of like drinking Califor-

document, including the document root.

- `endElement()` is generated for every element in the document, including the document root. There's an instance of this event for every `startElement()` event.
- `endDocument()` is generated after the root element of the document and all its nested content has been processed.
- `characters()` is generated when nested character data is found within an element. It is usually not called for insignificant whitespace character data.

To do anything meaningful with these events, you must provide an event handler with code to actually do something with the event. The `NaturalXMLHandler` exhibit shows the code for handling the key events shown earlier.

Writing a SAX handler is reminiscent of the old days (late 1980s) when one of us (Thiruvahtukal) was working on a compiler toolkit for LL-style parsing and also with the Yacc (yet-another-compiler compiler) tool. In most such tools, the event-handling methods we're talking about here are known as *semantic actions*, which is compiler-geek talk for "what to do when you encounter an important syntactic phrase." If you're interested, you can download the code from http://sourceforge.net/projects/apt/ and study it. Extensive comments are provided in the downloadable version.

To write a meaningful handler, you must extend the base class provided in the SAX framework. (In our approach, we use the SAX 1 parser provided with the Apache Xerces toolkit; it handles single namespace XML documents only.) We maintain a context stack (member variable `contextStack`) to keep track of what we're working on in terms of the XML document. For example, when working on a `Particle` nested within the `InitialConditions` element, it's key to keep the current `InitialConditions` instance on the stack so that the current `Particle` can ultimately link to it. (The notion of context is an old implementation trick used for compiling statically typed programming languages with nested scopes and building something known as the symbol table.)

We'll focus on `startElement()` and `endElement()` methods. In a nutshell, `startElement()` takes the incoming element name (`elementName`) and uses it to look up the class that should be used to construct a user-defined instance. It does this by consulting another data structure within the parser handler, called the class table (`classTable`). The keys of this table are element names, and the values are class descriptors. At the risk of peeping ahead, entries in this table are made by calling methods beginning with the name `register`; programmers can register arbitrary classes to create the document root and applinia wine in France, but I'm brave.) At portal.acm.org, you will find links to a great deal of computing literature in the Guide to Computing Literature as well as in their own online archive. You can read the *Journal of the ACM* all the way back to volume 1, issue 1, 1954, including such articles as "Automatic Strain-Gage and Thermocouple Recording on Punched Cards." Using the Guide, you will not only find a reference but see a list of the author's other works and the names of "Collaborative Colleagues." This use of hyperspace adds real value to an archive.

Reducing Computer Noise

Mrs. Dubois, family Web master, usually gets the hand-me-down computers from the rest of us since she has lower performance needs. When she inherited the Gateway (made extra loud by one of our son's roaring video cards), she started using my computer because hers made so much noise that she couldn't stand it. Sensitive violin-trained ears, you know. Well, this was clearly a problem I needed to solve.

Riding to the rescue was endpcnoise.com. It sells computers designed to be very quiet. I got prompt delivery on a high-performance model from Vancouver at a price not much more than an equivalent noisy PC. The site also sells components such as quiet power supplies, quiet case fans, quiet hard drives, and quiet CPU fans.

In unpacking the machine, I had to open the case to relieve the carefully stuffed interior of some bubble-wrap. The CPU fan looked interesting: apparently it's bigger and slower than normal. The case is bigger than normal (but not alarmingly so), and the wires are neatly organized so as to minimize noise when the fan blows over them.

I found another company that specializes in quiet components, called QuietPC. It operates in a variety of countries; the US outlet is quietpcusa.com. I did not see any assembled computers for sale, though.

Mrs. Dubois' new computer really is quiet, all the way down to the keyboard. When idle, you can tell it's turned on if the room is quiet, but even then, you have to listen. It's amazing how much nicer this makes computing. I think there's an endpcnoise computer in my future, too.

Welcome to George K. Thiruvathukal

These last two issues you've gotten to know Professor George K. Thiruvathukal. We're glad to announce that he will be joining me as co-editor of the Scientific Programming department. By the way, unlike mine, his name is pronounced just as it is spelled; you just have to keep on trucking through those syllables. He will take his turn cooking in the Cafe, bringing his own computer-science-flavored cuisine to the menu. As always, our intent is to bring you information that you can use. We encourage you to write for us; just contact either one of us to discuss your ideas.

cation nodes. We'll return to this issue in the next section to see how it all comes together. Once we know the class responsible for handling a given element (`elementName`), an instance is created. The line of interest is where `klass.newInstance()` appears.

The second item of business is to set the properties in the user-defined class; we do this by using introspection to look up the appropriate `set` method. (We don't need `get` methods at this juncture; the parser creates a user-defined data model from the XML document.) The framework lets the user map attribute names differently from how they appear in the source document. This is mostly a cosmetic issue that lets you observe the host language's naming conventions. For example, `Particle` has an `x` attribute (in lower case), but we want the property-setting method to be named `setX()`. We let users suggest a mapping for the property name. Again, we do this invocation by constructing the method descriptor for the method we want to look up (`setMethodName`) and invoking it on the instance we previously created (`element`).

The last item of business is to attempt to link the new instance to whatever instance is resting atop the stack. The very first time `startElement()` is called, the stack is guaranteed to have one element: the document root. (The code for creating this is just a special case and appears in `startDocument()`.) In the case of our particle.xml file, the topmost element is an `NBody` object. So an instance of `NBody` will be linked to the document root instance, which is detailed in the `NBodyRoot` class. The way this linkage is established is again achieved by using introspection. We ask whatever object is atop the stack whether its class provides an `add<MyType>()` method. If this method is absent, an exception is thrown, and the programmer gets a very clear indication of a problem. From an XML viewpoint, steps must be taken to preserve the actual content model in whatever mapping is performed. In the full source code, you will be able to observe that every class has an appropriate `add<Type>()` method (or methods) to address nested content.

At the end of the `startElement()` implementation, we push the newly created instance on the context stack. This will let any nested content be handled recursively.

The `endElement()` implementation is straightforward; we did the hard work in the `startElement()` method. All the `endElement()` method needs to do is to remove the instance resting atop the stack because this instance will have been linked as contained content to another instance, which is still being worked on.

The `characters()` implementation is straightforward. We simply notify the instance atop the stack that we've found some character data. More often than not, this character data is kept separately from other nested content (such as nested elements). However, in applications where it really matters, such as text processors, the nested character data could be maintained in a list of nested content instances and string data.

Putting It All Together

Visualizing this approach might be hard, but what makes it work for scientific programming is its ability to integrate existing codes with XML. By merely implementing straightforward interfaces, we can turn any class into an XML-capable class. There is no need to rewrite trees, and the number of methods the user must learn to write is minimal.

The `main()` method code in class `NBodyRoot` shows how to put an application together. Essentially, the user constructs a `NaturalXMLParser` instance. The code for this class attaches the handler `(NaturalXMLHandler)` code to do the dirty work of implementing the key parser event interface methods. Using the `registerElementClassMapping(String,Class)` method, each XML element is registered with the class that will be instantiated when the element is seen. The `registerElementAttributeMapping()` method allows attributes to be mapped nicely to the host language (as described). The `setDocumentRoot()` class lets users provide a class for the document root. Essentially, this framework lets a user make simple modifications to existing classes and then use the `NaturalXMLParser` to automatically instantiate the classes to create a heterogeneous object tree.

Getting a class descriptor is easy for users. In Java, for example, given a declared and visible name such as the class names in our application, you simply use `ClassName.class` to get its descriptor.

The `main()` method concludes with a demonstration of how to generate XML from the data model after the simulation is completed. Because each application class implements the `ContainedContent` interface, we can use a general-purpose code generator (available through the parser instance's `toXML()` method) to emit code from the user's data model. Please feel free to download the code and observe how fully compatible the output XML is with the input XML. This example SAX parser shows how to create a full persistence framework for arbitrary application classes.

In a future article, we'll present an extension of our framework in which existing user-defined classes can be used, without modification, as application classes in an internal object tree. This gives us more flexibility because we won't even need access to the sources of our application classes. CiSE

George K. Thiruvathukal is a visiting associate professor of computer science at Loyola University Chicago. He is also president and CEO of Nimkathana Corporation, which does research and development in high-performance cluster computing, data mining, handheld/embedded software, and distributed systems. He wrote two books with Prentice Hall covering concurrent, parallel, distributed programming patterns and techniques in Java and Web programming in Python. Contact him at gkt@nimkathana.com.

Konstantin Läufer is an associate professor of computer science at Loyola University Chicago. He is also director of architecture and application services at Nimkathana Corporation. His research interests are in programming languages, software architecture and frameworks, concurrent and distributed systems, and mobile computing. He received his PhD in computer science from the Courant Institute at New York University. Contact him through www.cs.luc.edu/~laufer.