Astronomical Data Analysis Software and Systems X
ASP Conference Series, Vol. 238, 2001
F. R. Harnden Jr., F. A. Primini, and H. E. Payne, eds.

Declarative Metadata Processing with XML and Java

Damien Guillaume, Raymond Plante

University of Illinois Urbana-Champaign, Urbana, IL 61801

Abstract. Metadata processing is an essential part of the Astronomy Digital Image Library (ADIL) and the BIMA Image Pipeline because of their complex and evolving data models. In order to choose the best technical solutions for these systems, a number of techniques have been evaluated; XML, Java, XSLT, DOM and Quick are described here.

An increased use of declarative (as distinguished from procedural) knowledge is found to improve the stability and flexibility of such systems.

1. Introduction

The Astronomy Digital Image Library[1] (ADIL) and the Berkeley Illinois Maryland Association (BIMA) Image Pipeline[2] are two projects carried out at the National Center for Supercomputing Applications (NCSA). Both projects make a heavy use of metadata, i.e., information about the processed data. The metadata is small in size compared to the processed data, but can be fairly complex, and the models can evolve with time as new features become available. The eXtensible Markup Language (XML) is well suited for encoding metadata because it supports flexible complex structuring, structure validation, and a format that can be read directly by both people and machines. Java is adapted to XML data because it is a solid object-oriented language, and many software packages for XML are written in Java.

This article explains a number of ways to use XML and Java to process metadata that have been tried for the ADIL or the BIMA Image Pipeline. It also highlights the value of using declarative knowledge, as opposed to procedural knowledge, to process the metadata.

2. Declarative vs. Procedural Knowledge Representation

The difference between declarative and procedural knowledge is somewhat subjective because of the possibility of converting one into the other. However, some relative comparisons of a number of characteristics show a clear difference between the two. In this article "declarative" and "procedural" will be used in the sense defined by the comparisons of Table 1.

[1] `http://adil.ncsa.uiuc.edu/`

[2] `http://monet.astro.uiuc.edu/BIP/index.html`

Table 1. Declarative vs Procedural Knowledge

DECLARATIVE KNOWLEDGE	PROCEDURAL KNOWLEDGE
Easy to validate	Hard to debug
Glass-box	Black-box
Explicit	Obscure
Data-oriented	Process-oriented
Ability to use knowledge in ways that the system designer did not foresee	Extensions are dangerous for stability
Slow (requires interpretation)	Fast (direct execution)
May require high-level data types	simple Data types can be used

3. Using XSLT

Extensible Stylesheet Language Transformations (XSLT), a part of the Extensible Stylesheet Language (XSL)[3], is a language to define XML tree transformations. XSLT 1.0 stylesheets are purely declarative, in the sense described in Table 1. Since they describe the transformations using only constants (without variables), a loop variable is not possible with XSLT. An XSLT processor is used to read the XSLT stylesheet and transform XML files into other XML files or HTML[4] files.

XSLT is very powerful for making tree transformations such as moving a subtree, changing the structure, adding subtrees, or formating the result. Some operations, however, are difficult or even impossible. Impossible operations often occur when a variable is needed, but some simple data processing operations can make the stylesheet very hard to read (even worse than a piece of code). A good way to solve these problems is to define some new high-level operations in a programming language like Java and use them in the stylesheet (though this might not be possible with any XSLT processor).

At some point, it becomes necessary to use some procedural knowledge. This limit can be pushed by using the right high-level data types. With XML already reasonably well advanced, XSLT provides a way to define a number of operations in a declarative way. One hopes that progress in the definition of XML (e.g., schemas, better data types) will improve XSLT rapidly.

[3] `http://www.w3.org/Style/XSL/`

[4] `http://www.w3.org/MarkUp/`

4. XSL Transformations in the ADIL

The ADIL is using the Astronomical Markup Language[5] (AML), an XML language, to store the metadata. The main operation of the ADIL server is to display some metadata in HTML, and XSLT is very well suited for this. Since not all the metadata is displayed and some metadata require complex transformations prior to display, it made sense to separate the transformation into two steps: first, the creation of an intermediary XML file containing only the information to display, and then the transformation of this XML file into HTML, with the addition of the user interface.

Since the complex transformation needed to realise these two steps proved impossible with XSLT, a Java program had to be used to transform the project metadata. Alternatively, the complex transformations could have been handled through special purpose Java methods callable from an XSLT stylesheet.

5. APIs for XML

There are two standard Application Programming Interfaces (APIs) for XML: Simple API for XML (SAX)[6] is a de-facto standard for generating parsing events, and Document Object Model (DOM)[7] is W3C[8] recommended for handling XML trees in memory. DOM parsers are often based on SAX parsers, but other possibilities include using a custom interface or creating a new interface based on SAX or DOM. The choice depends on the required performance: applications based only on SAX are faster than the those based on DOM parsers, but application-specific code has to be added to handle XML trees. Custom interfaces added on top of DOM can be very stable and powerful, but the result is very inefficient because of all the intermediate layers.

6. Using a DOM Parser

The declarative information used by a validating DOM parser (apart from the XML files) is mainly the Document Type Definition (DTD), describing the XML language of the documents to process. Using a DOM parser results in procedural knowledge (like a Java program) based on DOM objects. If no custom API is used on top of DOM, the types of the objects usually do not reflect their meaning in the application and cannot contain associated methods (e.g., all data defined as strings, even when numbers are used). Another problem with the DOM API is its complexity: this has led some developers to create alternative interfaces, like JDOM[9], a DOM-like interface simplified and optimized for Java.

---

[5] `http://monet.astro.uiuc.edu/~dguillau/aml/`

[6] `http://www.megginson.com/SAX/sax.html`

[7] `http://www.w3.org/DOM/`

[8] `http://www.w3.org/`

[9] `http://jdom.org/`

7. Using Quick

A good alternative to DOM parsers is Quick[10], open-source software initiated by JXML, Inc[11]. Quick transforms XML documents into Java objects automatically with a configuration file that is written in the QJML markup language and contains both the XML schema for the XML documents (XML schemas are next-generation DTDs containing type information) and the mappings between XML elements and Java classes. The mappings describe objects called "Coins" because they have a Java side and an XML side.

Applications using Quick make use of procedural knowledge, as they would with DOM parsers, except that they use Java classes adapted to the application, and the objects can be used in an explicit way. More declarative knowledge is used with Quick because the QJML file contains XML-Java mappings, while DOM parsers only use a DTD. Because this declarative knowledge can be used to replace some procedural knowledge, a utility for Quick (QJML2Java) was written to create Java classes based on the QJML file. These generated classes usually contain fields for the attributes or sub-elements, accessor methods, and some other useful XML-related methods. They can be extended with custom classes so that custom fields and methods are added to the object interfaces, while the link between the generated classes and the QJML file is preserved. If the schema must be updated (e.g., to add a new attribute to an element), the classes can be regenerated.

8. Conclusions

Before switching to Java, the BIMA archive was using Perl scripts to handle metadata, and much of the system information was hard-coded. The introduction of more declarative knowledge, by explicitly defining the objects to use, what information they contain, and how to access it, resulted in more stable, more flexible, and (unexpectedly) faster software.

Generating Java classes with an XML configuration file effectively replaces procedural knowledge (hand-written Java classes) with declarative knowledge (a configuration file that can be used to generate the same classes automatically).

With the help of higher-level data types, further improvements in this direction can be expected. Already fairly evolved, XML has greatly assisted the creation of declarative knowledge, as demonstrated by the XSLT language.

[10] `http://jxquick.sourceforge.net/`

[11] `http://www.jxml.com/`