Using Javadoc and XML to Produce API Reference Documentation

Donald M. Leslie
Apache Software Foundation
XML Project
dleslie@apache.org

ABSTRACT

The creation of API reference documentation and its integration into larger documentation sets present a number of challenges. This paper explores a strategy for using Javadoc, the primary mechanism for generating Java API documentation, in conjunction with an XML infrastructure, to improve the quality of API reference material and its integration into product documentation sets.

Categories and Subject Descriptors

I.7.4 [**Document and Text Processing**]: Electronic Publishing.

General Terms

Management, Documentation, Design.

Keywords

API documentation, Javadoc, doclets, HTML, XML, XSLT, XSL stylesheets, document transformation.

1. INTRODUCTION

A number of tools now exist for automating the production of API documentation. One of the most widely used of these tools is Javadoc [12]. For the core Java libraries and many Java product libraries, Javadoc output, hereafter called Javadoc, represents the most widely used form of documentation.

In many cases, Javadoc is the only documentation available to the application developers who use those libraries.

Why is Javadoc so widely used? The answer is simple, I believe. Almost no effort is required to produce Javadoc, and at its minimum it includes a rich set of accurate and useful information.

```
javadoc -sourcepath /dita/doclet/src
        -d /dita/build/htmldoc
        -overview dita/doclet/src/overview.html
        com.ibm.dita.doclet
```

Figure 1: Sample Javadoc instruction

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGDOC'02, October 20-23, 2002, Toronto, Ontario, Canada.
Copyright 2002 ACM 1-58113-543-2/02/0010…$5.00.

The Java development platform, itself free, includes the Javadoc tool. Issue the appropriate command-line instruction (as in Figure 1), and Javadoc uses the Java compiler to scan a set of Java source files and generate HTML output.

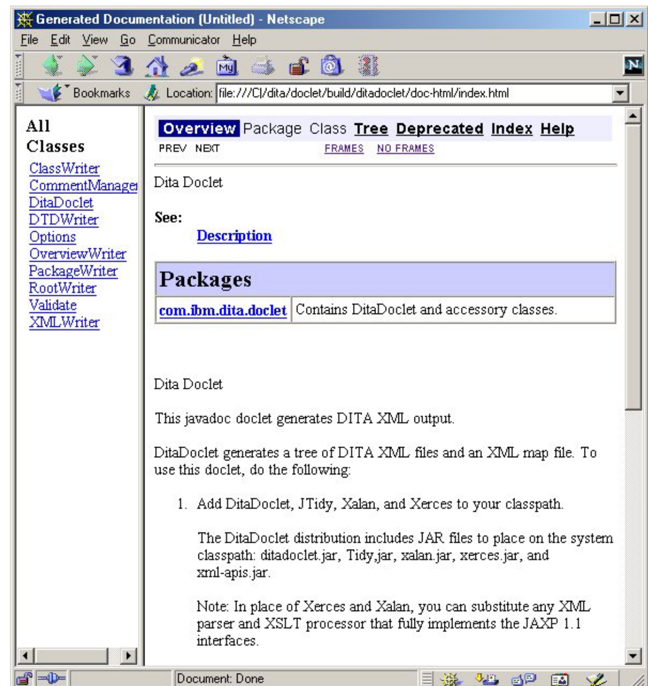

Figure 2: Sample Javadoc output

If the developer or a technical writer has included explanatory source comments before the class and member declarations in these files, the Javadoc includes those comments. If no such comments are present, Javadoc still generates output identifying package structure, class inheritance, variable and parameter datatypes, method signatures, and more.

A little browsing on the Internet quickly reveals that Javadoc is often run with source files that contain no or extremely limited explanatory comments. The output may be difficult to use, but it is arguably a lot better than nothing. A little trial and error (or perhaps a lot!) based on names, types, signatures, and inheritance trees, and the tenacious application developer can start to figure out what the documentation failed to include.

It would be a tedious process (with lots of room for error) for a writer to manually collect from the source code all of the information that comes for free with Javadoc. For example, a class may extend another class and implement one or more interfaces. The class it extends may in turn extend another class and

implement other interfaces. A Java class provides access to the members that appear in the classes it directly or indirectly extends. In other words, you may use this class to call methods and perform operations that do not actually appear in the Java source code for that class.

Using the compiler, the Javadoc tool makes this information available in the document it produces for each class.

Javadoc also provides ready access to the interfaces a given class implements. In many cases, developers use a "standard" interface API rather than the "vendor-specific" implementation API. For example, once you have learned the standard interfaces for working with XML (SAX [15], DOM [7], and JAXP [11]), you can use these interfaces to work with a variety of tools. You can even switch from one set of tools to another without having to recode your application.

To summarize, Javadoc is a marvelous tool, but most Javadoc output falls far short of providing the information that would make the Java libraries it documents easy to use.

2. Challenges

The situation described above clearly presents technical communicators and others who are interested in producing high-quality API documentation with some substantial challenges.

2.1 Who does what?

What role do technical writers and editors play in the creation of Javadoc? In many cases, they appear to have been cut out of the loop altogether. After all, in the "standard" product development environment, developers manage .java source files; writers and editors maintain separate documentation files. At best, perhaps an editor marks up the Javadoc output, and gives the markup to the developer. Based on appearances, even this minimal level of involvement appears often not to be the case.

Certainly it is true that developers should continue to be responsible for commenting their code, the orientation being on providing information required to understand, maintain, and update the code. But it is no less true that technical communicators should take responsibility for providing the documentation that users of the API require, including conceptual overviews, procedures for completing standard tasks, and reference materials.

The fact that this material is in or is mechanically derived from the source files should be seen as an advantage, not as an organizational problem to be solved by somehow segregating the documentation source files from the .java source files. As Frederick Brooks [3] pointed out, "A basic principle of data processing teaches the folly of trying to maintain independent files in synchronism."

Rather than denying writers access to the source files, the control system that is probably already in place just needs to be expanded to include writers as well as developers. In fact, writers can use a tool such as DocWiz [8], which provides access to the source files and allows the creation or modification of the comments used by Javadoc without permitting modifications to the code.

Javadoc incorporates comments beginning with /**, ending with */, and immediately preceding a class or class-member declaration. Javadoc ignores comments beginning with // and comments placed in the body of the code.

Comments for internal use should be entered such that Javadoc will not pick them up. The developer may in fact also enter comments that Javadoc will pick up, in which case the technical writer can edit or supplement the comments, or move all or a portion to a location where they will not appear in the Javadoc.

It is up to the program manager, or individual developers, writers, and editors, to establish a working model for producing both internal comments and Javadoc comments.

2.2 Integration

How is Javadoc material tied in with the rest of the documentation? In many cases, there is no additional documentation. Javadoc optionally incorporates text from an overview text file and from a text file for each package. These files may contain a certain amount of conceptual and procedural information. Indeed developers or writers may embed such information in the source-code comments.

This is a makeshift strategy that development groups are prone to fall back on in an environment of severe time and resource constraints, and using documentation procedures that disassociate Javadoc output from other forms of documentation. We should, however, keep in mind that Javadoc is essentially reference material. To address the needs of application developers, a conceptual overview should also be in place, along with procedures for using the API to perform meaningful tasks. In most cases, this information does not really belong in package-level or class-level comments. No matter how well the API is organized, packages and classes rarely map directly to procedures and underlying concepts.

The tool for which the API exists may also provide a user interface or involve the use of other resources. To use an XML transformation engine, such as Xalan [1] or SAXON [16], for example, the user needs information about XML syntax, document type definitions, XML schemas, and XSL stylesheets, information that clearly does not belong in the Javadoc.

3. Proposed Solution

Clearly a preliminary requirement is clarification of the role the technical communicators play in the documentation of APIs. I believe we can most effectively address this issue in the context of developing adequate tools to enable technical communicators to do their work and integrate Javadoc with other portions of a comprehensive documentation set.

3.1 Clarify role of technical communicators

Make sure that writers and editors are in the Javadoc-production loop. Ideally, technical communicators are also in the product/API design loop, which means they have the opportunity and responsibility to review the organization and naming conventions used in the API.

However they interact with developers and incorporate developer comments, technical writers with the appropriate training and experience should be responsible for the comments that will be picked up by Javadoc. Editors should mark up the Javadoc output, and developers provide technical review. Writers can then input

the edits and technical revisions, just as they do for other forms of documentation.

From the outset, technical communicators should be responsible for creating the package-level and overview files that Javadoc optionally incorporates in its output.

Most importantly, technical communicators should create, implement, and maintain comprehensive documentation plans that clearly delineate the relationships between Javadoc and other documentation.

The rest of this paper addresses the use of a Javadoc XML doclet and other XML tools to provide an infrastructure that makes it possible to coordinate Javadoc output with other documentation.

3.2 Use a Javadoc doclet

Starting with the release of Java 1.2 in 1995, the Javadoc tool includes a "doclet" interface (the com.sun.javadoc package) that enables users to control Javadoc output. Along with this interface, Javadoc provides a standard doclet that produces the HTML output as illustrated in Figure 2.

In projects I have worked on, we have made minor modifications to the standard doclet in order to control which API members are included in the output (a commonly raised issue) and to visually tag individual class and class members as experimental or for internal use only.

But the doclet interface provides complete control over the output, and a number of alternative doclets are now available for performing a variety of custom tasks and publishing a variety of formats, such as Postscript, PDF, RTF, LaTex, SGML and XML. For more information, see "Doclets" in the Javadoc FAQ (frequently asked questions) [12] and Doclet.Com[6].

3.2.1 Generating XML

If you are already producing documentation in an XML framework, or contemplating a move to such a framework, one very attractive alternative is to set up a doclet to generate XML. Many companies (including IBM, for which I developed the DITA XML doclet described in this paper) and public entities (such as the Apache Software Foundation, the host of the Xalan XSLT processor on which I work) are actively engaged in engineering a large-scale shift to XML documentation.

With XML, you can combine and integrate documentation from a variety of sources to provide as many doc sets as your user community may require or desire, and you can publish online or in print. For an introduction to XML, I recommend *XML in a Nutshell* by Harold and Means [10].

You can either use an existing XML doclet or create your own. If you want full control over the XML you are generating and over how you manage source comments, create your own. Having recently written an XML doclet to produce DITA XML [4], I am pleased to verify that setting up an XML doclet is a relatively straightforward process. The rapidly evolving XML infrastructure that is just as freely available as Javadoc provides tools that simplify that process of generating and validating XML.

3.2.2 DITA XML

The DITA specialization scheme makes DITA XML a particularly attractive form of XML to generate. Procedures and stylesheets that are established to transform DITA documents into HTML, PDF, or whatever, will be able to handle the Javadoc XML output, even if they are not aware of some of the specializations that the Javadoc includes. As a result, the process of adjusting publishing schemes originally designed to handle other kinds of documentation (conceptual topics, task-oriented procedures, non-API reference topics, etc.) will require minimal adjustment to fully take advantage of the Java API reference specialization that DITA Javadoc represents. For more on the subject, see Priestley [14].

3.3 What the Javadoc tool does

To understand the basic role and structure of a Javadoc XML doclet, it is useful to briefly review the Javadoc tool.

3.3.1 Javadoc compiler

Javadoc uses the Java compiler to scan a collection of .java source files (usually the source files for a set of Java packages). The compiler generates information for each class and class-member declaration. The Javadoc tool assembles this information, along with the comments into a tree structure of class objects, a structure that maps directly to the XML tree structure. Whether you are using the standard "HTML" doclet shipped with the Java Development Kit, or a custom doclet, the underlying Javadoc tool completes this parsing operation and provides the doclet access to a RootDoc object.

The RootDoc provides access to a PackageDoc for each package the tool has scanned, which in turn provides access to a series of ClassDoc objects for the interfaces and classes that make up that package. A ClassDoc object in turn provides access to a collection of Member objects for each variable (field), constructor, and method in that class.

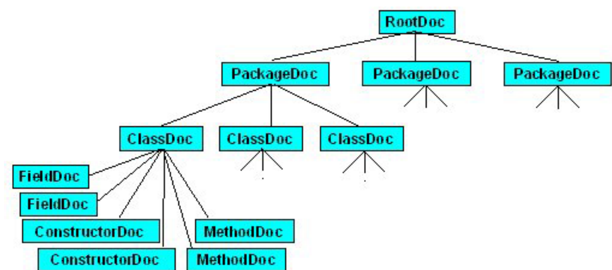

Figure 3: Javadoc object tree

3.3.2 Mapping the Javadoc object tree to XML

Mapping this tree structure to the XML tree structure is quite straightforward:

```
<root>
  <package>
    <class>
      <field/>
      <field/>
       ...
      <constructor/>
      <constructor/>
      ...
      <method/>
      <method/>
       ...
    </class>
     ...
  </package>
   ...
</root>
```

Figure 4: XML Javadoc tree

Of course you are free to use element names and to vary the organization as required to generate XML that conforms to your

own particular needs. Rather than creating a single XML document with the entire Javadoc output, you may want (as I did) to produce a tree of XML documents, along the lines of the tree of HTML documents the standard doclet produces.

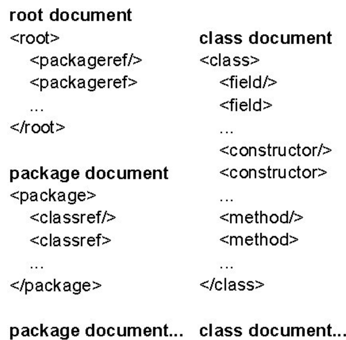

Figure 5: Tree structure of Javadoc documents

3.3.3 Code (and other) comments

As already mentioned, Javadoc pulls in comments from the source files. With two exceptions, Javadoc obtains all its information and text from the .java source files. Both exceptions are optional: an overview text file and a text file for each package.

You may indicate an overview text file when you issue the Javadoc command, in which case it places the content of that file in the RootDoc object.

If Javadoc encounters any package.html files in the source tree (each subdirectory contains a package), it places the content of that file in the appropriate PackageDoc object.

When you are working with the standard doclet, the overview document and package documents are HTML documents. But the overview document can contain any text, and it is easy enough to get Javadoc to look for package.xml in place of package.html files.

As we will see, the incorporation of comments is the most problematic area that confronts an XML doclet.

4. Implementing an XML doclet

Take advantage of the tools that exist for generating, manipulating, and validating XML. Rather than manually assembling the elements that make up the XML documents, for example, use the SAX TransformerHandler defined in the JAXP 1.1 interface (javax.xml.transform.sax.TransformerHandler), and supplied with any of the standard XML transformer engines. For each document, issue a startDocument statement, followed by a set of nested startElement and endElement statements that define the structure of the document. You close the document with an endDocument. statement.

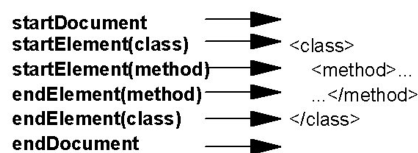

Figure 6 Using TransformerHandler to assemble a document

For the document statements, you define the location and file name, and for the element statements, you define an element its attributes, and its text content.

4.1 Handling source code comments

The only problem area that I encountered in the process of composing a doclet is determining how to manage the source code comments. Source code comments (including the text in overview and package text files), present a couple of complications:

- Javadoc @tags
- Structural issues

4.1.1 Javadoc @tags

Comment text is intermixed with Javadoc @tags, special tags that Javadoc recognizes. At least one of these tag types -- {@link} -- you need to convert in place to links or cross-references. The Javadoc tool provides you direct access to others, such as the @param tags that accompany a constructor or method.

You can instruct Javadoc to give you the comment text, without these tags, and to give you these tags without the comments. Accordingly, you set up a procedure in the doclet to convert some tags in place and place the text associated with other tags into the appropriate XML elements.

The following comment immediately precedes the DITA doclet start() method (the entry point for a doclet). It contains an {@link} tag and an @param tag.

```
/**
 * Launch the doclet by instantiating an {@link XMLWriter}
 * with the javadoc root.
 *
 * @param root the RootDoc generated by javadoc (using the
 * java compiler) before the doclet is called; contains
 * everything required to generate the output.
 */
public static boolean start(RootDoc root)
```

Figure 7: Sample source comment with Javadoc @tags

In the XML output, the @link appears as an XML link element in the body of the comment, and the text associated with the @param tag has been placed along with information provided by the compiler in a param XML element:

```
<method                      The unique ID facilitates links to the method.
  id="DitaDoclet.start-com.sun.javadoc.RootDoc">

  <title>start(com.sun.javadoc.RootDoc)</title>

  <comment>
    <p>Launch the doclet by instantiating an
      <link href="XMLWriter.xml">XMLWriter</link>
      with the javadoc root.</p>
  </comment>

  <param name="root"
         dimension=""
         type="com.sun.javadoc.RootDoc">

    <description>
      The RootDoc generated by javadoc (using the
      java compiler) before the doclet is called,
      contains everything required to generate the
      output.
     </description>
   </param>
</method>
```

Figure 8: Handling Javadoc @tags in the output

4.1.2 *Comment structure*

Unless you have absolute control over what is put into the body of the source comments, the structure (or lack thereof) presents a more fundamental difficulty.

From the standpoint of the Javadoc tool, source comments may be raw text (as in the preceding example), may include a certain amount of HTML, or may in fact include XML tags. The text includes whatever the developer or writer put there.

If your output files include any text that is not well-formed XML, you cannot manipulate the text with a standard XML parser or transformer, so this presents a potentially fatal drawback. A tool to turn raw text into XML and to clean up HTML is needed. Fortunately, such a tool exists and is freely available from the W3C: JTidy [13].

The DITA doclet uses JTidy to place each comment it finds in an XML-compliant HTML document (in memory). All that is required is to set some JTidy flags to indicate what kind of output is desired, then to instruct JTidy to parse the text and hand you the "tidy" output. To transform this output into XML, simply apply a stylesheet that transforms HTML into your target XML. The DITA doclet includes a CommentManager, whose job it is to manage this process of calling JTidy and applying a stylesheet to the JTidy output.

4.2 Optional features

The DITA doclet includes some optional features that you might want to incorporate:

- Validating your output against a DTD and/or XML schema. XML schemas are still quite new. I still validate each output file against a DTD, simply by parsing the file with an XML parser (I use Xerces [2]) that has validation turned on, so that it will report any errors it finds. This is very useful for locating any structural problems that block the transformation of source comments into valid XML.
- Generating XML overview and package files that authors can use in the future, in which case you can use those files in future runs in place of HTML overview and package files.
- Generating a map that encapsulates the structure of your entire output. You can then use the map when you are incorporating the API documentation in the larger doc set.

```
<?xml version="1.0" encoding="UTF-8"?>
<jdocmap label="DITA Javadoc">
  <file href="root.xml"/>
  <filelist label="com.ibm.dita.doclet package">
    <file href="com/ibm/dita/doclet/package.xml"/>
    <file href="com/ibm/dita/doclet/CommentManager.xml"/>
    <file href="com/ibm/dita/doclet/Validate.xml"/>
    <file href="com/ibm/dita/doclet/Validate.Handler.xml"/>
    <file href="com/ibm/dita/doclet/RootWriter.xml"/>
    <file href="com/ibm/dita/doclet/OverviewWriter.xml"/>
    <file href="com/ibm/dita/doclet/PackageWriter.xml"/>
    <file href="com/ibm/dita/doclet/DitaDoclet.xml"/>
    <file href="com/ibm/dita/doclet/DTDWriter.xml"/>
    <file href="com/ibm/dita/doclet/Options.xml"/>
    <file href="com/ibm/dita/doclet/ClassWriter.xml"/>
    <file href="com/ibm/dita/doclet/XMLWriter.xml"/>
  </filelist>
</jdocmap>
```

Figure 9. Sample Javadoc map

5. Looking forward

The proposal outlined in this paper clearly involves a certain amount of setup: a doclet working in conjunction with an XML transformer and parser, and a set of DTDs and/or schemas, and stylesheets for transforming the XML to the desired presentation formats. This paper has suggested some avenues to explore in implementing such a setup. Substantial design work remains.

Work is underway at IBM to complete a DITA XML infrastructure for authoring and publishing user documentation. Similar endeavors as doubtless taking place elsewhere. To the degree that Java API documentation comes into play, an XML doclet is a critical tool in these efforts.

Design and coordination issues will doubtless arise during the process of integrating Javadoc output into larger documentation sets. The sooner documentation groups become actively engaged in this endeavor, the sooner we can confront these issues. But we can also look forward to establishing some very efficient processes for creating easily customized documentation sets, and to producing well-organized and comprehensive API documentation that I am more than confident will elicit eager attention and appreciation from a variety of sources.

A number of tools similar to Javadoc exist for other programming languages. For example, I have used Doc++ [5] and Doxygen [9] to generate C++ reference documentation, and other tools like these certainly exist. The success we achieve in using Javadoc to generate XML API documentation is bound to stimulate parallel endeavors with these other tools and languages.

In short, technical communicators face interesting and challenging work on a variety of fronts implementing some radically new and improved systems for producing the high-quality API documentation that application developers need.

6. ACKNOWLEDGMENTS

Xalan and Xerces are Copyright © 1999-2002, Apache Software Foundation

7. REFERENCES

[1] Apache Software Foundation, Xalan Java. http://xml.apache.org/xalan-j/index.html

[2] Apache Software Foundation, Xerces Java.Parser http://xml.apache.org/xerces-j/index.html

[3] Brooks, F. The Mythical Man Month.. Addison-Wesley, Boson, MA, 1995, 169.

[4] Day, D., Priestley, M. and Schell, D. Introduction to the Darwin Information Typing Architecture: Toward portable technical information. IBM DeveloperWorks, 2001-2002. http://www106.ibm.com/developerworks/xml/library/x-dita1/

[5] Doc++ http://www.zib.de/Visual/software/doc++/

[6] Doclet.com, Zentech, Inc. http://www.doclet.com/

[7] Document Object Model (DOM)
http://www.w3.org/DOM/

[8] DocWiz
http://www.mindspring.com/~chroma/docwiz/

[9] Doxygen
http://www.doxygen.org/

[10] Harold, E. R., and Means, W. S. XML in a Nutshell: A Deskotp Quick Reference. O'Reilly and Associates, Inc., Sebastopol, CA, 2001.

[11] Java API for XML Processing (JAXP)
http://java.sun.com/xml/jaxp/index.html

[12] Javadoc Tool Home Page
http://java.sun.com/j2se/javadoc/

[13] JTidy HTML Parser and Pretty Printer in Java.
http://www.lempinen.net/sami/jtidy/.

[14] Priestley, M. Specializing topic types in DITA: Creating new topic-based document types. IBM DevelperWorks 2001-2002. http://www-106.ibm.com/developerworks/xml/library/x-dita2/

[15] SAX (Simple API for XML)
http://www.saxproject.org/

[16] SAXON XSLT Processor
http://saxon.sourceforge.net/