**Module Design for a Comment Mentoring System**

James Sant’Angelo

The College of New Jersey

santang5@tcnj.edu

**ABSTRACT**

The COMment menTOR (COMTOR) web application is an academic tool built on Java to take advantage of Javadoc and its Doclet API in order to be easily expandable and readily adaptable for a variety of users [2, 3]. It takes a unique approach via its use of doclets to provide more resources for doclet writers to use when creating new metrics on which to grade and analyze source code. This paper will describe and analyze these resources and the general design of the Java base that forms the core of COMTOR.

**Categories and Subject Descriptors**

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement – *documentation*.

**General Terms**

Documentation, Languages, Verification.

**Keywords**

Javadoc, doclet.

**1. INTRODUCTION**

COMment menTOR (COMTOR) is a web application designed to help students improve the quality of their comments in the Java code they write. It was designed under the premise that new students to the Computer Science field often start with Java, and can through this application build a good foundation for future work and languages. COMTOR is written with PHP and HTML, with Smarty templating, as the web application front-end. This interface resides on top of a Java base that analyzes submitted code and does the specified work or calculations. Reports based on the work that was done and other usage and account information are stored in a MYSQL database. The Java base is built around Javadoc, using its tags as a way to provide structure for students and a uniform way to document/comment their code.

The design of COMTOR as reflected through its interface generally divides all users into two usage categories: “professors” and “students”. “Professor”-type accounts have the ability to create assignments, through which students can access and submit code. Modules, created to compute and analyze the submitted code according to various metrics, can be specified by the professor (provided they exist in the system), or left up to the student for that given assignment submission. After submitting, a report is generated and displayed to the student, detailing the results of the doclets’ execution, which is saved by the database should the student want to access it again at a later time.

To facilitate the analysis of submitted Java code, there was a need to use or create a tool that could standardize the code regardless of its purpose or content, and to be able to generate and internal structured view of the application. Javadoc, an automatic documentation generator, performs this function for COMTOR. To understand how Javadoc forms an integral part of COMTOR, it is necessary to briefly review the structure of Javadoc and the Doclet API, and how a COMTOR doclet is created and interacts with Javadoc.

**2. BACKGROUND**

Javadoc is a tool developed by Sun Microsystems that generates documentation based on predetermined tags (considered the standard of the industry) placed within source code [3]. To comment source code, comments are started with ‘/\*\*’, and tags such as ‘@author’ or ‘@param’ denote information about the class, method, or field with which it is associated (see Figure 1). Javadoc parses the information it gathers from these tags, and builds a tree structure of the entire source code structure. This tree structure is saved into what is called a RootDoc.

/\*\*

\* This is an example comment for the

\* following example method.

\*

\* [@param](http://java.sun.com/j2se/javadoc/writingdoccomments/#@param) someNum the parameter that I

\* am passing to this method

\* [@return](http://java.sun.com/j2se/javadoc/writingdoccomments/#@return) a value that I’ve calculated

\*/

public int doStuff(int someNum) {

someNum++;

return nameOfParam1;

}

**Figure 1. Example of Javadoc commenting and tags.**

The RootDoc is an internally represented tree containing all of the classes, methods, and attributes of the complete package. Based on the requisite tags, Javadoc will identify each element of the package, and then provide the Doclet API to analyze the created tree.

For example, consider the following sample Java program:

/\*\*

\* The <code>HelloWorldExample</code> class

\* is a small test example of a class that

\* we can run our Doclets on.

\*

\* @author Michael Locasto

\*/

public final class HelloWorldExample {

/\*\*

\* This member variable contains an ID

\* string derived

\* from the time that the constructor

\* of an instance of

\* this class is invoked.

\*/

private String m\_data = "";

/\*\*

\* A fairly pointless constructor that

\* simply initializes

\* an instance data member.

\*/

public HelloWorldExample() {

m\_data = "time id: "

+System.currentTimeMillis();

}

/\*\*

\* Another pointless method.

\*/

public void readMe() {

int x = 5;

if(x!=5)

readMe();

}

/\*\*

\* Give me that old Hello, World feeling.

\*

\* @param args Command line arguments

\*/

public static void main(String [] args){

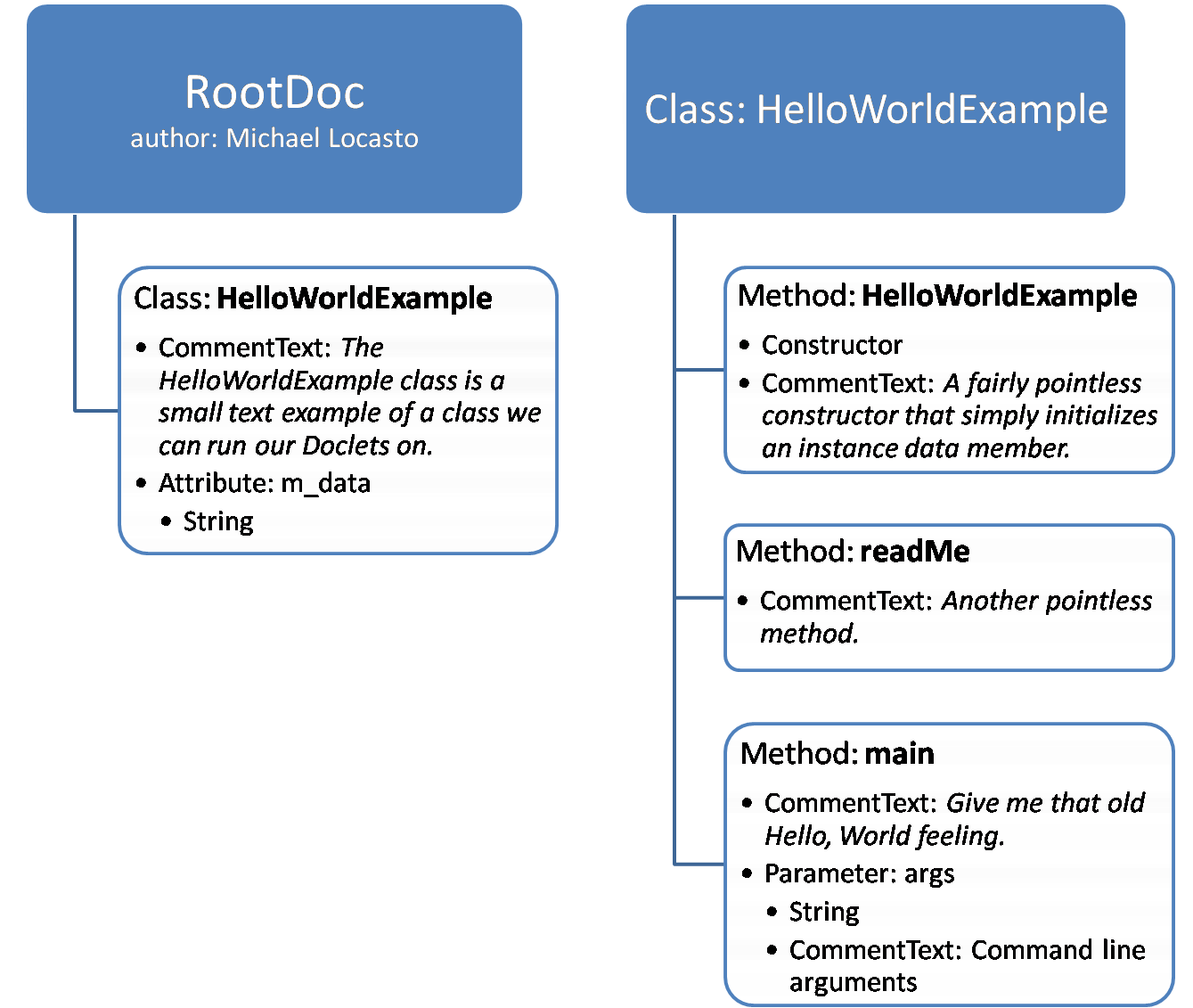
System.out.println("Hello, world.");

}

}

**Figure 2. Sample Java code to be processed by Javadoc.**

While parsing the above, Javadoc will incrementally build the RootDoc structure. The resultant structure, if visually represented by means of a diagram, would look like the following:



**Figure 3. Conceptual figure outlining structure of a RootDoc generated based off of Figure 2.**

Javadoc does not consider the actual implementation of any methods; in fact, it ignores all of the contained code, and bases the RootDoc off of what is essentially an interface and tags written by the author to identify and store information about each class, method, or attribute.

**3. CORE DESIGN**

The core of the COMTOR system features an application written in Java to handle source code and Javadoc, as well as any doclet resident in the system. After source code is submitted through the front-end, a script is run to compile the source code into a temporary directory. Upon successfully compiling the source code, Javadoc is run on the generated class files, and then work is handed over to the COMTOR Java driver (see Figure 4).

# Run javadoc if code compiled

if [ $compiled = 0 ]; then

javadoc -private --assignment-id $4 --config-file $JAVA\_CONFIG –

doclet comtor.ComtorDriver -docletpath $CLASSES $MYVAR > JavadocOut.txt 2>&1

javadocRtn=$?

else

java -cp $CLASSES:$CLASSPATH comtor.GenerateErrorReport $JAVA\_CONFIG

fi

**Figure 4. Excerpt from javadoc.sh, which compiles the submitted code and runs Javadoc.**

To start this process, the user is presented with a form through which modules can be selected, as long as they are present in the system. “Professor”-type users can allow or disallow any of the offered modules for each specific assignment, and “student”-type users, if given the option, can choose which doclet should be run on their code. The doclets selected are then saved to a temporary text file. From the driver, this list of selected doclets is processed, and each becomes a thread started by the driver. As each doclet completes its work, the information is sent back to the driver, where it is then handed off to a report generator to eventually be displayed for the user (see Figure 5).

// Start thread

System.out.println("Starting " + docletName);

DocletThread docThrd = new DocletThread();

docThrd.setRootDoc(rootDoc);

docThrd.setDocletSectionsPrepStatements(docletSectionsPrepStmt,

docletParametersPrepStmt);

docThrd.setAnalyzer(cd);

docThrd.start();

threads.add(docThrd);

...

// Wait for all threads

for (int i = 0; i < threads.size(); i++)

{

DocletThread docThrd = (DocletThread)threads.get(i);

docThrd.join();

Properties tmp = docThrd.getProperties();

if (tmp != null)

v.addElement(tmp); //add the resulting property list to the vector

}

GenerateReport report = new GenerateReport();

report.setConfigFilename(configFile);

report.generateReport(v); //pass the vector to the generate report class

**Figure 5. Excerpt from ComtorDriver.java, showing thread and report execution.**

**3.1 COMTOR Doclet Structure**

Javadoc is designed to be customizable and expandable, such that modules, or doclets, can be “applied” to the tree that the Javadoc engine creates. Doclets can simply take a RootDoc as a parameter, and can then glean any and all information about the complete compiled package. Since all useful information should have been tagged, the RootDoc will essentially contain all of the relevant source code in an identifiable way; calling RootDoc.classes(), for example, will return a list of all the classes compiled in the package.

The general structure of a COMTOR doclet is contained within a class named appropriately according to the name of the doclet. One main method, called analyze(), is responsible for all of the analysis. analyze() takes the RootDoc as a parameter. To properly parse the information in the RootDoc, the analyze() method will typically use arrays and populate them with information from the RootDoc (as detailed above). The analyze() method will return a Properties list when it is done, if necessary, and report any grading information. In line with this, and standard for all COMTOR doclets, are several small methods which receive certain parameters to determine the grading breakdown, on which the analyze() method can base its calculations.

For a module to be recognized by COMTOR, it must be in the proper directory and contain an entry within the database; when presenting choices to the users, the list of available doclets is populated by the ‘doclets’ table.

There are other design standards that are implemented throughout COMTOR’s doclets; these will be covered in the follow sections.

**4. MODULE DESIGN**

Modules act as the link between the two main user interactions with the application. The user will submit source code, and then see a report generated. All source code is analyzed, all calculations are done, and all reports are essentially generated (content-wise, though not necessarily presentable in its raw form) at this level. Modules represent the deepest information goes before more information is returned back towards the user. It is at this level, therefore, that the largest considerations for general execution of COMTOR come into play.

**4.1 Ordering System**

Because the COMTOR driver uses a threaded approach to calling the doclets (as opposed to calling them sequentially, which would be much less inefficient and would take more run time), there is no way for the driver to determine which order the results will come back in. Results are standardized, and are all properties lists with the requisite information stored in sets of properties. In order for the driver to determine the ordering for the results, so that the property lists may be displayed in order and in a standardized fashion, a rough ordering scheme has been created. As the doclets return their properties lists, they are all added to one massive properties list that is then handed off to the Java report generator, which inserts the properties into the database for archiving/retrieval by the front-end PHP report displayer (see Figure 6), which will pull the lists from the database and organize/display them according to the hierarchy of each set (see Figure 7). An ordering system has been implemented to provide a hierarchy for property sets of the pattern ‘000.000.X’, where ‘X’ is represented as a single character (such as ‘a’), or a series of another three integers (such as ‘001’).

// Go through array of property list data pairs (length is -2 because we don't need "title" and "score")

docletOutputItemsPrepStmt.setLong(1, docletEventId);

for(int j=0; j < arr.length-2; j++) {

if(arr[j] != null) {

// Insert data pair into the database

docletOutputItemsPrepStmt.setString(2, arr[j]);

docletOutputItemsPrepStmt.setString(3, list.getProperty("" + arr[j]));

docletOutputItemsPrepStmt.executeUpdate();

}

}

**Figure 6. Excerpt from GenerateReport.java, demonstrating code inserting properties into the database.**

// Checks the index to see whether the attribute is a class, method,

// or comment. Index is from the property list, examples shown below

foreach($props as $prop)

{

$index = $prop['attribute'];

// property list index - 011

if(strlen($index) == 3)

$props2[] = array('class'=>'class', 'value'=>$prop['value']);

//property list index - 011.002

else if(strlen($index) == 7)

$props2[] = array('class'=>'method', 'value'=>$prop['value']);

//property list index - 001.002.a

else if(strlen($index) > 7)

$props2[] = array('class'=>'comment', 'value'=>$prop['value']);

}

**Figure 7. Excerpt from reports.php, demonstrating properties hierarchy.**

As the selected code in Figure 7 shows, each ‘level’ is classified by a set of three integers, which are usually auto-incremented (in the case where the doclet may not know how many units it is processing, such as when the doclet CheckForTags will process the number of class tags there are without knowing ahead of time how many there will be), and each property under the same header (such as, in the aforementioned doclet, each class has certain methods associated with it) is identified again with another auto-incrementing three digit number, such that all levels will come back in the order specified by the doclet (and will be displayed with proper indenting/section marking).

The third and last part of the ordering system is determined by either a single character or another string of three integers, as determined by the doclet. A single character would indicate that the property is of the designation as a ‘method’ type of property (or as implied, of that ‘layer’ and should be displayed as such), whereas the last countable three digit integer sequence is used for anything below that level, considered a ‘comment’ type of property.

**4.2 RootDoc API Handling**

Two of the doclets that COMTOR currently has available analyze the actual text used in comments. ReadingLevel attempts to calculate the Flesch-Kincaid Grade Level and other statistics such as average syllables per word or words per sentence based on pseudo-sentences created out of the comments throughout the document. SpellCheck simply reports all of the misspelled words in the document, delineating between normal words that would be found in the dictionary and words that should not be flagged misspelled though they may appear that way (such as Java reserved words or user-defined variable/method names) through the use of an alternate ‘allowed list’ that will be compared to the initial ‘misspelled list’ so that certain entries in the ‘misspelled’ list can be removed before being finalized/reported (see Figure 8).

// Add class name to list of valid words

javaWords.add(classDoc.name());

// Add class comment

parseComment(classDoc.commentText());

// Get all fields

FieldDoc[] fields = classDoc.fields();

for(int h = 0; h < fields.length; h++)

{

javaWords.add(fields[h].name());

// Add field comment

parseComment(fields[h].commentText());

}

**Figure 8. Excerpt from SpellCheck.java, illustrating use of lists to keep track of valid words.**

*4.2.1 ANTLR*

Both of these modules (ReadingLevel and SpellCheck) currently use ANother Tool for Language Recognition (ANTLR) as a lexer to parse through the text found in the comments [1]. A pattern is generated to match comments found (lines beginning with ‘//’, ‘/\*’, or ‘/\*\*’ for example, with the appropriate ending characters for multi-line comments), and all matching strings were compiled into one massive string (on which various cleanup work was done, such as removal of unneeded special characters or the addition of periods to create pseudo-sentences for later algorithms to calculate metrics in ReadingLevel) (see Figure 9).

// Clean up comments found.

// Multiline, trim off '\*/' at the end of the comment

if(matched.charAt(1)=='\*')

matched = matched.substring(0,matched.length()-2);

// Trim off '//' or '/\*' from front of comment

matched = matched.substring(2);

// Eliminate trailing and leading whitespace

matched = matched.trim();

/\* Makes comment a complete sentence for the reading algorithm;

later sentence detection algorithms will be used.

This is a hack for now. \*/

if(matched.length() > 0)

{

temp = matched.charAt(matched.length()-1);

if(temp == '.' || temp == '?' || temp == '!'); // Do nothing

else

matched = matched + ". ";

// Removes any newlines and tabs from the string

matched = matched.replace('\n',' ');

matched = matched.replace('\t','\0');

// Comment added to allComments to be analyzed for reading level.

allComments +=matched;

}

**Figure 9. Excerpt from ReadingLevel.java, demonstrating String processing that comments go through.**

To implement this method, however, the actual source code files are needed, which presents a design challenge. Accessing the original source code will violate the portability of using Javadoc’s RootDoc.

**5. PORTABILITY OF COMTOR DOCLETS**

The power of the doclet is its ability to be portable; instead of simply being static code, tailored for a particular use, it can be applied in a variety of systems, as long as the same interface interacts with the doclet. However, some doclets that previously resided in the system were not truly portable, and used external text files instead of using solely the Javadoc RootDoc.

**5.1 Use of External Text Files**

ANTLR was essentially incorporated into certain modules to parse through raw source code (and NOT RootDoc objects). To do this, the doclets need a way to determine what raw source code files need to be processed, such that they could be read in. During the compilation process, a text file ‘source.txt’ is used to keep a list of what source code files were submitted (see Figure 10). This file is ultimately used in report generation to store the Java files in the database. Because of the availability of this text file, it was also used in the modules as a way to identify which files needed to be parsed.

#find the list of java files within the src dir and store the list in source.txt

cd $UPLOAD\_PATH/$1/

find src -name \*.java > source.txt

chmod 755 \*.txt

**Figure 10. Excerpt from javadoc.sh, the point at which source.txt is populated.**

**5.2 Remove of External Text Files**

Ultimately, the use of ‘source.txt’ can be considered a convenience, given that the file was already available because of other processes that needed it. Additionally, some of the analysis done could be still performed without the use of raw text files (particularly in the case of SpellCheck), and the use of the external text file is not standard to the rest of the doclets and limits the use of that particular doclet to our application alone (which eliminates the inherent portability of the Doclet API). To keep these two doclets standard with the rest of the ones implemented, it was necessary to remove the use of this file from these doclets. The Doclet API makes this relatively easy. Instead of parsing through the source code files and raw text, it was a relatively minor design shift to instead have the doclets parse through the Javadoc tree and pull out all comments with the commentText() function. By creating arrays of each of the ClassDocs and MethodDocs contained in the RootDoc on the fly, and then methodically going through each one, the doclet can continually append each of the comments as text to a growing string. In fact, similarly to how it was done previously, each of the comments could be processed before being added to the growing allComments String (to add periods, for example, for pseudo-sentences in ReadingLevel) (see Figure 11).

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Processes a single class

\*

\* @param class Class to process

\* @return String String to which all comments will be written/saved

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

private String processClass(ClassDoc classDoc)

{

String allComments = "";

// Add class comment

allComments = parseComment(classDoc.commentText());

// Get all fields

FieldDoc[] fields = classDoc.fields();

for(int h = 0; h < fields.length; h++)

{

// Add field comment

allComments += parseComment(fields[h].commentText());

}

// Get inner classes

ClassDoc[] inner\_classes = classDoc.innerClasses();

for(int h = 0; h < inner\_classes.length; h++)

processClass(inner\_classes[h]);

// Get comments for tags

Tag[] tags = classDoc.tags();

for (int h = 0; h < tags.length; h++)

allComments += parseComment(tags[h].text());

// Get comments for each method

MethodDoc[] methods = classDoc.methods();

for(int j = 0; j < methods.length; j++)

{

// Get comments for tags

tags = methods[j].tags();

for (int h = 0; h < tags.length; h++)

allComments += parseComment(tags[h].text());

// Get method comments

allComments += parseComment(methods[j].commentText());

}

return allComments;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Parses words out of comments and removes punctuation

\*

\* @param comment Comment to parse

\* @return String String to which all comments are added after processing

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

private String parseComment(String comment)

{

char temp;

// Replace parenthesis, brackets, dashes, and periods with spaces

comment = comment.replaceAll("[()<>-]|\\+"," ");

//Creates sentences out of comments to be analyzed.

temp = comment.charAt(comment.length() - 1);

if(temp == '.' || temp == '?' || temp == '!');

else

comment += '.';

// Comment added to allComments to be analyzed for reading level.

return comment;

}

**Figure 11. Excerpt from modified ReadingLevel.java, demonstrating parsing through RootDoc without uses of an external text file.**

**6. ANALYSIS**

Because COMTOR is essentially a wrapper web application built around academically oriented Javadoc doclets, it is essential for the growth of the project to make the doclets as portable as possible. Virginia Tech’s Computer Science Department uses the Web-CAT system (Web-based Center for Automated Testing) to do the same thing as COMTOR; automated testing and grading for code completeness and unit testing [4]. The open source doclets that are implemented in COMTOR were tailored specifically to our own implementation of our driver and application structure, in that the use of ‘source.txt’ made certain doclets unusable and non-portable for other systems.

The main benefit to using ANTLR and parsing raw source code files is that any comments that are incorrectly tagged or not linked to a class, method, or field would still be included into the ReadingLevel and SpellCheck doclets (and in the case of SpellCheck, may provide insight into why a particular tag or comment was not linked to the appropriate class, method, or field). With the removal of use of this text file (and by extension, parsing raw source code and potentially the use of ANTLR all together), this is no longer an option. Theoretically, proper Javadoc commenting will ensure that all relevant comment text will be returned by completely parsing through the Javadoc tree; in practice, this is rarely the case, as is evident by the need of a SpellCheck doclet. Moreover, a misspelled tag will not cause the source code to fail compilation, but will cause an error in the Javadoc tree, which would potentially be easier to find and fix with a SpellCheck module that is not limited to and based on that same flawed tree. Continuing on this same vein, because the SpellCheck module is flexible in the sense that custom words can be added to the list of allowed words, these same Javadoc tags can be added such that if they are misspelled, it will be reported, which will enable the student to much more easily track where the problem is, and obtain a more accurate Javadoc tree (which will lead to more accurate results from the other doclets). And while other doclets, particularly CheckForTags, may be able to perform the same check, it adds to the completeness and robustness of COMTOR’s analyzing prowess if multiple doclets are able to verify the same information from different approaches.

This benefit extends beyond its application in SpellCheck (and to a lesser extent, ReadingLevel). The Interactions doclet parses the RootDoc and attempts to map out all of the interactions between the various classes and methods within a compiled package; that is, if a certain method A calls method B, this is noted and reported back to the user such that the student will be able to more easily see how and where each of the classes interact (without the complication of actual code, which is the inherent design behind Javadoc to begin with). The Interactions doclet also currently uses ‘source.txt’, to build the interactions ‘tree’, and create a list of user-defined methods (through the Doclet API) with method calls found by parsing through the raw source code (via ANTLR). Because the RootDoc does not build information on the content of methods (in fact, Javadoc can run successfully and completely on non-implemented code, which was the intent in its design such that it could be used from the very beginning of application development), the power of this doclet could not be realized without parsing the raw source code.

Based on the current use of ANTLR and the external text file used to identify the raw source code files of the submitted package, it is recommended that though these doclets should have an alternate version wherein they are more portable and can therefore be extended to other applications (whether as a stand-alone open source release or integration into Virginia Tech’s Web-CAT), it is more beneficial that these doclets remain in their current form (as changes to these doclets have not been rolled into a release as of the writing of this document) for use in COMTOR.

The downside of external text files is clear; Doclet API usage through COMTOR’s doclets is not standardized, but in order to expand COMTOR to certain uses (as detailed in the next section), it is necessary for the COMTOR application to be able to parse and analyze raw source code in addition the generated RootDoc for a submitted Java application. In fact, it is the comparison between the two that is groundbreaking academically, and will enable COMTOR to provide a more complete and useful report to both the student and instructor. This is in essence what differentiates Web-CAT and COMTOR in their approaches to comment quality testing, and it should be noted that a doclet writer has both raw source code and a generated RootDoc as resources when designing new metrics to measure submitted Java packages.

**7. FUTURE WORK**

COMTOR is far from complete. At least two of the doclets provided, ReadingLevel and Interactions can both be largely expanded. Though the original design of the ReadingLevel (due to the algorithmic limitations of Flesch-Kincaid) did not calculate certain values for anything but the overall submitted package, it can be modified to calculate these same values for smaller cases (even down to the individual method level), so that any variations or extremely skewed results can be traced back to the individual words that are causing the aberration.

The Interactions module currently only builds the documentation of the various interactions between classes and methods in the submitted package. The power behind this module, however, is in applying its results to the user-defined interactions, perhaps as specified by their own commenting through the use of custom tags. It will then be possible to not only describe exactly how their methods interact, but also whether or not our own generated model matches what the student claims to be happening in the application, providing a whole new level to automated comment analysis and academic programming study.

**8. CONCLUSION**

COMTOR’s use of Javadoc and its Doclet API provide a powerful resource for instructors and students. The whole application was designed from the ground up to be adaptable and portable, from a simple web application front-end to an open-ended Javadoc and doclet implementation for analysis.

With these strengths, COMTOR ideally provides detailed feedback for students that is not labor-intensive, and will therefore hopefully become prevalent for academic use such that studying Computer Science students will be able to benefit from better documented code.

**9. REFERENCES**

1. ANTLR Parser Generator, http://www.antlr.org/
2. COMTOR.org Home, http://www.comtor.org/index.html
3. Javadoc Tool Homepage, http://java.sun.com/j2se/javadoc/
4. Web-CAT Research Server Front Page, http://web-cat.cs.vt.edu/