Development of a Framework for Automated Java Source Code Comment Analysis

Joseph A. Brigandi
Faculty Mentor: Dr. DePasquale
brigand2@tcnj.edu

ABSTRACT

The detailed use of documentation is one of the most-overlooked ways of improving software quality and speeding implementation. However as students begin to program, they typically see commenting as a burden, often leaving it out completely. This paper details a prototype software system under development by the author at TCNJ which will demonstrate a method for professors and students to process code and receive detailed reports containing information about the documentation. The detailed reports will allow the user to check whether comments exist, as well as run analysis on the quality and quantity of the documentation. Included in the paper are example reports and a source code appendix of the software developed for this project.

1. BACKGROUND

In order to process Java source code and generate detailed reports based on the comments in the code, a computer software tool from Sun Microsystems called Javadoc [4] is used for generating application programming interface (API) documentation into HyperText Markup Language (HTML) [3] format. Javadoc is currently the industry standard for documenting Java classes. To use the tool properly, specific commenting styles as well as Javadoc tags are required when documenting source code. The following example demonstrates the use of a Javadoc comment block.

```
/**
 * A sample Javadoc comment block.
 * @param firstValue
 * @param secondValue
 * @return true if the values are equal
 * @author Joe Brigandi
 */
 boolean isEqual(int firstValue,
                 int secondValue)
 {
     ...
 }
```

As the example shows, a Javadoc comment begins with '/**' and ends with '*/'. Between these two symbols, a comment block contains Javadoc tags. Tags are made up of two sections, the tag itself and the description. The tags begin with '@', at sign, and the description is any text following it. A list of commonly used Javadoc tags is as follows:

@author - developer name

@exception - documents an exception thrown by a method

@param - defines a method parameter

@return - documents the return value

@throws - documents an exception thrown by a method

@version - provides the version number of a class or method

2. RUNNING JAVADOC

2.1 Documenting One or More Packages

Javadoc can run in two ways, on individual source files or entire packages. To run Javadoc on one or more packages, two steps need to be taken care of beforehand. First, the source files (*.java) for that package must be located in a directory having the same name as the package. Second, a destination directory that specifies where Javadoc saves the generated HTML files is needed. Once these two pieces are in place, the syntax to run Javadoc is as follows:

```
%javadoc -d {destination path} -sourcepath {path
of the package}
```

The sourcepath can contain multiple paths by separating them with a semicolon (;). In all cases, running Javadoc will generate HTML-formatted documentation for the public and protected classes in the specified packages and save them in the designated destination path.

2.2 Documenting One or More Classes

To run Javadoc on individual source files as opposed to entire packages, the same basic syntax is used.

```
%javadoc -d {destination path} -sourcepath {path
of the file}
```

Since packages are not being used in this scenario, each set of source files that will be processed need to be included. Each path is separated by a single space. Again, Javadoc will generate HTML-formatted documentation for each class that is included in the sourcepath.

2.3 Documenting Both Packages and Classes

In certain situations, the user may want to run Javadoc on entire packages and individual classes at the same time. This can be done by combining the two previous examples. As always, the destination path for the resulting HTML-formatted documentation is included. To add entire packages, the sourcepath is needed, just as it was when only packages were being used. Since individual classes are also being used, the sourcepath is not needed a second time. The proper syntax is as follows:

```
%javadoc -d {destination path} -sourcepath
{package} {files}
```

3. BASIC ARCHITECTURE

The basic system architecture is shown in Figure 1 below. The Javadoc tool parses the source code and receives a com.sun.javadoc.RootDoc in return. The RootDoc represents the root of the program structure information for one run of Javadoc. From this root, all other program structure

information, such as classes and methods, can be extracted using the methods contained in the Javadoc API. The driver is used to intercept the RootDoc and pass it to the doclets for analysis.

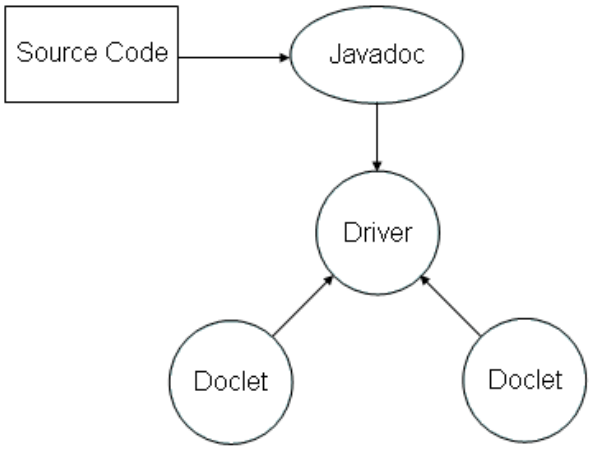

Figure 1. The basic system architecture.

4. DOCLETS

4.1 Overview

With a basic understanding of the Javadoc tool and the ability to write documentation with it, the next step is to develop doclets, commonly known as plug-ins, to customize the HTML-formatted output. Doclets specify the content and format of the output for the processed source code. In terms of content, doclets can be used to analyze the structure of the Java source code comments. They use tags to check for proper documentation. As for the formatting of the reports, doclets are also capable of changing the look and feel of the HTML-formatted files. Using Cascading Style Sheets (CSS) [2], doclets have complete control of fonts, tables, colors, etc. These are just a few of the many possibilities.

4.2 Compiling and Running

The process of compiling and running doclets is different than executing normal Java source code. You can compile doclets with the javac compiler. The class files for the Doclet API are in the file lib/tools.jar in the Java 2 SDK, and therefore needs to be included on the compiler's classpath. The following example demonstrates this:

```
%javac -classpath C:\jdk1.5.0\lib\tools.jar
SampleDoc.java
```

Once the doclet is compiled, it is ready to run. When executing, the doclet name and doclet path must be stated as in the following example:

```
%javadoc -doclet SampleDoc -docletpath
C:\Java\Doclets\*.java
```

5. ANALYSIS RESULTS

Once the doclets are compiled and run, the analysis results need to be stored. The goal is to store the results in a useful way so that they can be used later on in the final report. Each doclet has its own set of data which should be kept separately from the others. Also, the order in which the data is collected is extremely important for the report. The data for each method within a class should be stored in the order they appear in the actual source code. In order to accomplish these goals, we need a data structure that allows us to manipulate the order of the data for each doclet. A property list is the perfect data structure for this situation. Due to the fact that property list consists of a pair of values, it allows us to use the first value as an index, keeping the proper order in tact.

6. PROPERTY LIST

A property list is a set of properties made up of pairs of strings, a key and its corresponding value. The key is used to retrieve the corresponding value, and acts as an index in the list. Each doclet creates a property list and populates it with the data it collects during the analysis process. Table 1 below shows a sample property list. Each doclets property list begins with a title, description, and date. These three fields make up the header of the final report. The rest of the list is made up of the data collected during analysis. In order to keep the entries in the property list in the correct order, a special naming convention is used. The key consist of two parts, the class index and the method index, separated by a period.

Table 1. A sample property list.

Key	Value
title	Documentation Length
description	Calculates # of words for each method's documentation.
date	11/29/06 2:30 PM
1	class: Student
1.0	The length of comments for method 'main' is 10 words.
1.1	The length of comments for method 'setName' is 21 words.
1.2	The length of comments for method 'getName' is 15 words.
2	class: Math
2.0	The length of comments for method 'main' is 8 words.
2.1	The length of comments for method 'calc' is 31 words.
2.2	The length of comments for method 'attach' is 10 words.
2.3	The length of comments for method 'detach' is 11 words.
2.4	The length of comments for method 'format' is 3 words.

7. CLOSER LOOK AT THE DOCLETS

The source code for each of the three doclets created so far is listed in the appendix. The first line in each doclet imports the Javadoc library, allowing Javadoc to be used throughout the class. Also in each doclet, an interface called ComtorDoclet is implemented. This interface contains the analyze method,

which is used to run the analysis in each of the doclets. The driver uses the interface to call the analyze method for each doclet that is selected. A property list, that stores the header information such as title, description, and date, is created at the beginning of the class. After these initial steps are taken, each doclet continues with its analysis of the source code.

7.1 CheckForTags

The CheckForTags doclet checks for the proper use of Javadoc tags in the source code's documentation. This is extremely useful in checking to make sure all parameters, return types, and exceptions are listed in the comments, as well as making sure they are documented in the correct format. The source code for CheckForTags is listed in Appendix A. As always, the previously mentioned initial steps are taken at the beginning of the class. After the initial steps, two arrays are created, one for the list of classes and another for the list of methods. A for loop is used to run through the list of classes in the array. Each time through this loop, the class name and loop index are entered into the property list. Within the loop, another loop is used to iterate through the list of methods in the class. Each time through the loop, three items need to be taken care of: the return type, the parameters, and any exceptions.

First, the return type of the method is retrieved, along with any return tags in the documentation. If the return type is void, there shouldn't be any return tags in the comments. If however, the return type is not void, we check to see if the return tag matches up with the actual return type. Each time the class goes through this process, the result is stored in the property list.

Next, we retrieve the parameters from the method, as well as any param tags in the documentation. Using a for loop, list of parameters for the method are checked to see if there is a corresponding param tag in the comments. The result of whether or not the param tag exists is stored in the property list. This only takes care of parameters in the method, so another loop runs through the param tags, to see if there are any extras listed. For each param tag, we check to see if there is a corresponding parameter, and the result is stored in the property list.

The same exact methodology is used for exceptions and throws tags. The list of exceptions and tags are retrieved. Again a for loop checks for throws tags that correspond to any exceptions that are thrown. Just as we did with the param tags, a check for extra throws tags is performed, and the result is stored in the property list. Finally, once we have gone through each class and all its methods, the property list of results is returned.

7.2 CommentAvgRatio

The CommentAvgRatio doclet calculates the length of documentation (in number of words) for all the methods in a given class and computes the average. The source code for CommentAvgRatio is listed in Appendix B. To demonstrate how this doclet works, the initial steps of importing Javadoc and creating a property list with the header information are executed. Next, two arrays are created, one for the list of classes and another for the list of methods. A for loop is used to run through the list of classes in the first array. Each time through this loop, the class name and loop index are entered into the property list. Within the loop, another for loop is used to iterate through the list of methods in the class. Each time through the loop, a scanner is used to count the number of words in the current method's documentation. The number of words is store in the property list for the method. Once the method loop is done executing, the average length of documentation for the class is calculated and stored in the property list. This calculation is done at the end of each iteration through the class loop, providing the average for each class. Finally, the last step in the class is to return the property list.

7.3 PercentageMethods

The PercentageMethods doclet calculates the percentage of methods in a class that have been commented. The source code for PercentageMethods is listed in Appendix C. Again, the previously mentioned initial steps are taken at the beginning of the class. Following the initial steps, two arrays are created, one for the list of classes and another for the list of methods. A for loop is used to run through the list of classes in the array. The loop contains another loop to run through the list of methods in the class. Each iteration through the loop checks to see if the length of comments in the method is greater than zero. If so, the number of methods commented is incremented. When the loop is done executing, the percentage of methods that are commented is calculated and entered into the property list. Finally, the last step in the class is to return the property list.

8. SYSTEM DESIGN

In looking at the initial goals of the project, it provides insight into the final design of the system. Ideally, the system will allow a student or professor to login and select the doclets they wish to run on their source code. Once the user makes this decision, a detailed HTML-formatted report, whose design is based on the doclets selected, will be produced. In an attempt to get from the initial goals to the finished product, the following system design was put in place.

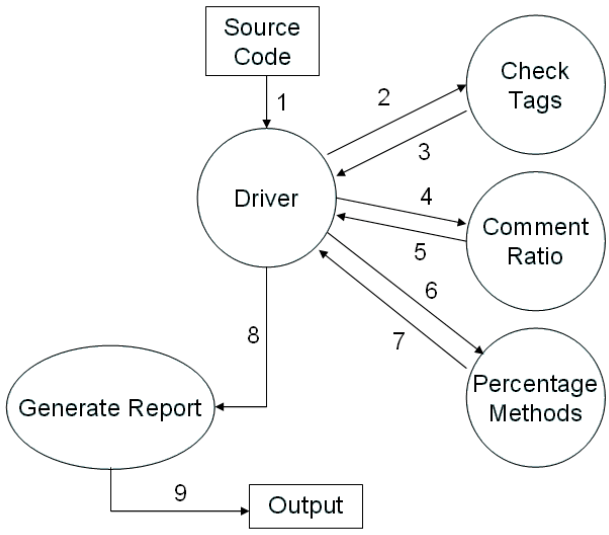

Figure 2. The detailed system design.

8.1 Initial Steps

The first step in the process of running the system is retrieving the list of doclets the user selected. When the user decides

which doclets they want to run, a text file will be created with a list of the selected doclets. That text file will be read in to the system by the driver and broken up into a list of strings. The driver will iterate through the list and call each doclet. In order to call the doclets however, a few things need to be taken care of. Java reflection [5] is used to instantiate the classes at runtime, using just their names from the text file. Reflection supports dynamic programming, which solves the problem of hard coding the names of the doclets before runtime. Once the class is instantiated, an interface is used to call the analyze method, which is present in each doclet. The analyze method is where all the analysis takes place, including the gathering of data and storing it in a property list data structure.

8.2 Driver

The system design is built around a driver called 'ComtorDriver.' The driver is actually a doclet acting as a driver program. The Javadoc tool parses the source code and passes a RootDoc to the driver. The driver runs the doclets by passing the RootDoc to each of the doclets. Each doclet uses the RootDoc to run its analysis of the source code, populates a property list with the collected data, and passes it back to the driver. The driver takes the property lists and stores them in a vector data structure. Once the doclets are finished executing, the driver then passes the vector of property lists to the 'GenerateReport' class.

8.3 Report Generator

The responsibility of the report generator class is to accept a vector full of property lists, sort them, and display the information in the specified format. This class sorts the property list by copying its contents into an array, and then calling the sort method. The sorted array contains the data in the correct order to be displayed in the final report. Each list contains the header and all the data collected from the doclet. The final report can be generated in any of the following formats: HTML, PDF, or plain text.

9. FUTURE WORK

Now that the framework of the automated Java source code comment analysis is in place, the next task will be to web-enable the existing prototype system. Following this, implementing new analysis modules and web-enabled features can further enhance this project.

The primary add-on feature will be a student login/archival system that will integrate with a back-end database. The database will archive records each time a student uses the comment analyzer. As students use the program throughout the semester, they will build a profile that allows professors to see the progress of their work.

Another tool that will be added to the system is the ability to generate reports in Portable Document Format (PDF) [1].

10. REFERENCES

[1] Adobe Portable Document Format. 2006. http://www.adobe.com/products/acrobat/adobepdf.html

[2] Cascading Style Sheets Home Page. 2006. http://www.w3.org/Style/CSS/

[3] HyperText Markup Language (HTML) Home Page. 2006. http://www.w3.org/MarkUp/

[4] Javadoc Tool Home Page. 2006. http://java.sun.com/j2se/javadoc/

[5] Reflection. 2006. http://java.sun.com/j2se/1.3/docs/guide/reflection

Appendix A
CheckForTags.java

```
1   import com.sun.javadoc.*;
2
3   /**
4    * The CheckForTags class checks for the proper use of Javadoc return, throw, and param tags.
5    */
6   public class CheckForTags implements ComtorDoclet
7   {
8       Properties prop = new Properties();
9
10      /**
11       * Obtain methods from each class. Check returns tag. Check throws tag. Check param tags.
12       *
13       * @param rootDoc  the root of the documentation tree
14       * @returns Properties list
15       */
16      public Properties analyze(RootDoc rootDoc)
17      {
18          prop.setProperty("title", "Check for Tags");
19          prop.setProperty("description", "This doclet checks for proper use of Javadoc tags.");
20          DateFormat dateFormat = new SimpleDateFormat("M/d/yy h:mm a");
21          java.util.Date date = new java.util.Date();
22          prop.setProperty("date", "" + dateFormat.format(date));
23
24          ClassDoc[] classes = rootDoc.classes();
25          MethodDoc[] methods = new MethodDoc[0];
26
27          for(int i=0; i < classes.length; i++)
28          {
29              prop.setProperty("" + i, "class: " + classes[i].name());
30
31              methods = classes[i].methods();
32              for(int j=0; j < methods.length; j++)
33              {
34                  prop.setProperty(i + "." + j, "method: " + methods[j].name());
35                  Tag[] returnTags = methods[j].tags("@return");
36                  String returntype = methods[j].returnType().typeName();
37
38                  if(returntype=="void")
39                  {
40                      if(returnTags.length==0)
41                          prop.setProperty(i + "." + j + ".b", "Analyzed method " + methods[j].name()
                            +
                            "'s declared return type and @return tag. The declared return type is void
                               and
                            there is no @return tag present in the comments.");
42                      else
43                           prop.setProperty(i + "." + j + ".b", "Analyzed method " +
                              methods[j].name() +
                            "'s declared return type and @return tag. The declared return type is void
                               but
                            an @return tag is present in the comments. There should be no @return tag
                            since the declared return type is void.");
44                  }
45                  else
46                  {
47                      if(returnTags.length==1)
48                          prop.setProperty(i + "." + j + ".b", "Analyzed method " + methods[j].name()
+
                            "'s declared return type and @return tag. The declared return type is " +
                            returntype + " and there is an @return tag present in the comments.");
49                      else if(returnTags.length==0)
50                          prop.setProperty(i + "." + j + ".b", "Analyzed method " + methods[j].name()
+
                            "'s declared return type and @return tag. The declared return type is " +
                            returntype + " but there is no @return tag present in the comments.");
51                      else if(returnTags.length > 1)
52                          prop.setProperty(i + "." + j + ".b", "Analyzed method " + methods[j].name()
+
```

```
                "'s declared return type and @return tag. The declared return type is " +
                returntype + " but there is " + returnTags.length + " @return tags present
            in
                the comments.  There should only be one @return tag.");
53          }
54
55          Parameter[] parameter = new Parameter[0];
56          parameter = methods[j].parameters();
57          Tag[] paramTags = methods[j].tags("@param");
58          int paramCount=0;
59
60          for(int s=0; s < parameter.length; s++)
61          {
62                 boolean check = false;
63                 for(int q=0; q < paramTags.length; q++)
64                 {
65                     if(paramTags[q].text().startsWith(parameter[s].name()))
66                     check = true;
67                 if(check)
68                 {
69                     prop.setProperty(i + "." + j + ".c" + paramCount, "Analyzed method " +
                       methods[j].name() + "'s parameter " + parameter[s].typeName() + " " +
                       parameter[s].name() + ". The paramter and paramter type match the
                       @param tag in the comments.");
70
71                     paramCount++;
72                     break;
73                 }
74             }
75                 if(!check)
76             {
77                 prop.setProperty(i + "." + j + ".c" + paramCount, "Analyzed method " +
                    methods[j].name() + "'s parameter " + parameter[s].typeName() + " " +
                    parameter[s].name() + ". There is no @param tag present for this
                paramter.");
78
79                 paramCount++;
80             }
81          }
82
83          for(int s=0; s < paramTags.length; s++)
84          {
85                 boolean check = false;
86                 for(int q=0; q < parameter.length; q++)
87                 {
88                     if(paramTags[s].text().startsWith(parameter[q].name()))
89                         check = true;
90                 if(!check)
91                 {
92                     prop.setProperty(i + "." + j + ".c" + paramCount, "Analyzed method " +
                       methods[j].name() + "'s @param tag " + paramTags[s].text() + ". There is
                    no
                       parameter in the method for the this @param tag.");

93                     paramCount++;
94                 }
95             }
96          }
97
98          Tag[] throwsTags = methods[j].tags("@throws");
99          ClassDoc[] exceptions = methods[j].thrownExceptions();
100             int throwsCount=0;
101
102             for(int s=0; s < exceptions.length; s++)
103             {
104                 boolean check = false;
105                 for(int q=0; q < throwsTags.length; q++)
106                 {
107                     if(throwsTags[q].text().startsWith(exceptions[s].name()))
108                         check = true;
109                 if(check)
110                 {
```

```
111                         prop.setProperty(i + "." + j + ".d" + throwsCount, "Analyzed method " +
                            methods[j].name() + "'s exception " + exceptions[s].name() + ". The
                            exception matches the @throws tag in the comments.");
112
113                         throwsCount++;
114                             break;
115                         }
116                     }
117                 if(!check)
118                 {
119                         prop.setProperty(i + "." + j + ".d" + throwsCount, "Analyzed method " +
                            methods[j].name() + "'s exception " + exceptions[s].name() + ". There is
                        no
                            @throws tag present for this exception.");
120
121                             throwsCount++;
122                     }
123             }
124
125             for(int s=0; s < throwsTags.length; s++)
126             {
127                 boolean check = false;
128                 for(int q=0; q < exceptions.length; q++)
129                 {
130                     if(throwsTags[s].text().startsWith(exceptions[q].name()))
131                         check = true;
132                     if(!check)
133                     {
134                         prop.setProperty(i + "." + j + ".d" + throwsCount, "Analyzed method " +
                            methods[j].name() + "'s @throws tag " + throwsTags[s].text() + ". There
                        is
                            no exception in the method for this @throws tag.");
135
136                         throwsCount++;
137                     }
138                 }
139             }
140         }
141         return prop;
142     }
143 }
```

Appendix B
CommentAvgRatio.java

```
1   import com.sun.javadoc.*;
2
3   /**
4    * The CommentAvgRatio class measures the average length of comments for methods in a class.
5    */
6   public class CommentAvgRatio implements ComtorDoclet
7   {
8       Properties prop = new Properties();
9
10      /**
11       * Examine each class and its methods. Calculate the length of each method's comments.
12       *
13       * @param rootDoc  the root of the documentation tree
14       * @returns Properties list
15       */
16      public Properties analyze(RootDoc rootDoc)
17      {
18          prop.setProperty("title", "Comment Average Ratio");
19          prop.setProperty("description", "Calculates the length of each method's comments.");
20          DateFormat dateFormat = new SimpleDateFormat("M/d/yy h:mm a");
21          java.util.Date date = new java.util.Date();
22          prop.setProperty("date", "" + dateFormat.format(date));
23
24          ClassDoc[] classes = rootDoc.classes();
25          MethodDoc[] methods = new MethodDoc[0];
26
27          for(int i=0; i < classes.length; i++)
28          {
29              prop.setProperty("" + i, "class: " + classes[i].name());
30              int total=0;
31
32              methods = classes[i].methods();
33              for(int j=0; j < methods.length; j++)
34              {
35                  Scanner scan = new Scanner(methods[j].getRawCommentText());
36                  int count=0;
37                  while(scan.hasNext())
38                  {
39                      scan.next();
40                      count++;
41                  }
42                  prop.setProperty(i + "." + j, "The length of comments for the method '" +
                    methods[j].name() + "' is" + count + " words.");
43
44                  total+=commentLength;
45              }
46
47              int average = total/methods.length;
48              prop.setProperty(i + "." + methods.length, "The average length of comments for the
class
                '" + classes[i].name() + "' is " + average + " words.");
49          }
50          return prop;
51      }
52  }
```

Appendix C
PercentageMethods.java

```
1   import com.sun.javadoc.*;
2
3   /**
4    * The PercentageMethods calculates the percentage of methods in a class that are documented.
5    */
6   public class PercentageMethods implements ComtorDoclet
7   {
8       Properties prop = new Properties();
9
10      /**
11       * See if rawCommentText for each method has a length more than zero. If so, count it in the
12       * total frequency of commented methods for that class. Calculate the percentage.
13       *
14       * @param rootDoc  the root of the documentation tree
15       * @returns some boolean value
16       */
17      public Properties analyze(RootDoc rootDoc)
18      {
19          prop.setProperty("title", "Percentage Methods");
20          prop.setProperty("description", "Calculates the percentage of commented methods per
class.");
21          DateFormat dateFormat = new SimpleDateFormat("M/d/yy h:mm a");
22          java.util.Date date = new java.util.Date();
23          prop.setProperty("date", "" + dateFormat.format(date));
24
25          int methodsCommented = 0;
26          double percentCommented = 0.0;
27
28          ClassDoc[] classes = rootDoc.classes();
29          MethodDoc[] methods = new MethodDoc[0];
30
31          for(int i=0;i<classes.length;i++)
32          {
33              methods = classes[i].methods();
34              for(int j=0;j<methods.length;j++)
35                  if(methods[j].getRawCommentText().length() > 0)
36                      methodsCommented++;
37
38                  if(0!=methods.length)
39                  {
40                      percentCommented = (double)((1.0*methodsCommented)/methods.length);
41                      prop.setProperty("" + i, Math.round(percentCommented*100) + " percent (" +
                        methodsCommented + "/" + methods.length + ") of class " + classes[i].name() +
"\'s
                        methods are commented.");
42                  }
43                  else
44                      prop.setProperty("" + i, "class: " + classes[i].name() + "has no JavaDoc\'d
                        methods.");
45
46                  methodsCommented = 0;
47                  percentCommented = 0.0;
48              }
49          }
50          return prop;
51      }
52  }
```

Appendix D
ComtorDoclet.java

```
1   import com.sun.javadoc.*;
2
3   /**
4    * ComtorDoclet is an interface class.
5    */
6   public interface comtorDoclet
7   {
8       public Properties analyze(RootDoc root);
9   }
```

Appendix E
ComtorDriver.java

```
1   import com.sun.javadoc.*;
2
3   /**
4    * ComtorDriver is the driver application for the system.  It runs the selected doclets.
5    */
6   public class ComtorDriver
7   {
8       /**
9        * Accepts a property list from the doclets and stores them in a vector.
10       * The vector is passed to the report generator class.
11       *
12       * @param rootDoc the root of the documentation tree
13       * @return boolean value
14       */
15      public static boolean start(RootDoc rootdoc)
16      {
17          try
18          {
19              Vector v = new Vector();
20              Scanner scan = new Scanner(new file("Doclets.txt"));
21              String docletName;
22              while(scan.hasNext())
23              {
24                  docletName = scan.nextLine();
25                  Class c = Class.forName(docletName);
26                  ComtorDoclet cd = (ComtorDoclet) c.newInstance();
27                  Properties list = cd.analyze(rootDoc);
28                  v.addElement(list);
29              }
30
31              GenerateReport report = new GenerateReport();
32              report.generateReport(v);
33              scan.close();
34          }
35          catch(ClassNotFoundException cnfe)
36          {
37              System.err.println(cnfe.toString());
38          }
39          catch(InstantiationException ie)
40          {
41              System.err.println(ie.toString());
42          }
43          catch(IllegalAccessException iae)
44          {
45              System.err.println(iae.toString());
46          }
47          catch(IOException ioe)
48          {
49                  System.err.println(ioe.toString());
50          }
51
52          return true;
53      }
```

```
54  }
```

Appendix F
GenerateReport.java

```
1   /**
2    * The <code>GenerateReport</code> class is a tool to
3    * generate a report from a vector containing property lists.
4    *
5    * @author Joe Brigandi
6    */
7   public class GenerateReport
8   {
9       /**
10       * Takes a vector full of property lists and
11       * generates a report.
12       *
13       * @param v is a Vector of property lists
14       */
15      public void generateReport(Vector v)
16      {
17          try
18          {
19              PrintWriter prt = new PrintWriter(new FileWriter("ComtorReport.html"));
20
21              prt.println("<html>");
22              prt.println("<head>");
23              prt.println("<title></title>");
24              prt.println("<style type=text/css>");
25              prt.println("</style>");
26              prt.println("</head>");
27              prt.println("<body>");
28
29              for(int i=0; i < v.size(); i++)
30              {
31                  Properties list = new Properties();
32                  list = (Properties)v.get(i);
33
34                  String[] arr = new String[0];
35                  arr = list.keySet().toArray(arr);
36                  Arrays.sort(arr);
37
38                  prt.println("<h3>");
39                  prt.println(list.getProperty("title") + "<br />");
40                  prt.println(list.getProperty("description") + "<br />");
41                  prt.println(list.getProperty("date") + "<br />");
42                  prt.println("</h3>");
43                  prt.println("<p>");
44
45                  for(int j=0; j < arr.length-3; j++)
46                  {
47                         if(arr[j] != null)
48                         prt.println(list.getProperty("" + arr[j]) + "<br />");
49                  }
50                  prt.println("</p>");
51              }
52
53              prt.println("</body>");
54              prt.println("</html>");
55
56                  prt.close();
57          }
58
59          catch(Exception ex)
60          {
61              System.err.print("!!!An exception was thrown!!!");
62              System.err.println(ex.toString());
63          }
64      }
65  }
```