Automated Evaluation of Source Code Documentation: Interim Status Report

Ben Hirsch
Jesse M. Heines

University of Massachusetts Lowell
Dept. of Computer Science
One University Ave., Lowell, MA 01854
01-1-978-934-3634

{bhirsch,heines}@cs.uml.edu

ABSTRACT

The poor quality of students' source code documentation is a major impediment to software development and maintenance that receives little attention in programming courses. While students are almost always required to document their source code, they typically get as much credit for trivial, useless comments such as

```
int n ;  // declare an integer variable named n
```

as they do for meaningful, truly helpful ones.

We have attempted to develop a Web-based application that can analyze source code documentation, assign it quantitative measures, and provide comprehensive feedback on how to improve it. This paper reports on our efforts to date.

Acknowledgement: Seed funding for this work was provided by the ACM SIG SCE via a Special Projects Grant.

Categories and Subject Descriptors:

D.2.4 Software/Program Verification.

General Terms: Documentation, Standardization.

Keywords: Programming, Documentation, Source Code.

1. THE PROBLEM

In November 1999, the Department of Education Office of Student Financial Assistance issued a notice of its desire to procure "independent verification and validation services" that included the ability to "evaluate source code documentation to ensure accuracy and completeness" (FBO Archive, 1999). At least one company claims on its Web site to be able to provide this service (NITC, 2003), but the authors' efforts to contact that company have not been successful. Unfortunately, while few would argue that quality source code documentation is valuable, few (if any) guidelines exist on how to produce or evaluate it.

ITiCSE '04, June 28-30, 2004, Leeds, UK.

Thus, source code documentation remains a critical — yet often neglected — facet of computer program development at all levels.

The perplexing coincidence in this situation is that tools for producing well-formatted source code documentation are readily available. Javadoc (Sun, 2000) and Doxygen (van Heesch, 2003; Sandberg, 2003) allow programmers to generate beautifully formatted Web pages that provide comprehensive source code documentation in standardized formats with relatively little effort. Given these tools, students can *and should* be taught and required to produce such documentation for all assignments.

We believe that students' performance in a computer science program will be enhanced if they are required to write thorough and meaningful documentation. If we can develop a measurable standard for documentation and assign a numerical value to students' programs that accurately represents conformance that standard, we feel that we can help students learn to write documentation and consequently better understand their code.

1.1 Going Beyond the Obvious

Writing documentation in a standard format is not enough. Documentation must also be useful. For example, documentation should not simply state the obvious:

```
k++ ;  // add 1 to k     (a useless comment that
                          documents the obvious)
```

This type of comment is #2 on Green's (2003) satirical list of 12 guidelines for documenting *unmaintainable* code. Interestingly enough, the authors have found that Green's guidelines are actually some of the most useful in this arena if one reads the statements as admonitions of what *not* to do. Kabutz (2002) finds most source code documentation so useless that "it would never occur to me to read the comments that geeks like you and I had written." He admits, however, that comments are useful "when they provide information that I could not glean from the [code]," and he even concedes that in some situations they're "absolutely essential."

Kabutz (*op cit.*) humorously reports that his grades went up considerably when he gave in to instructors' demands that he document his code, even though his comments looked as follows (*N.B.* all spelling errors are intentional!):

```
int i;   /* Conter variabble for "for" loop. */
int t;   /* Toatl of additions for calculaton */
int d;   /* Indicidual number for calclatuion */
```

```
/* "for" loop */
for (i=0; i<100; i++) { /* inc i by one up to hunderd */
  d = f();       /* get ghe calue for d */
  t = t + d;     /* ad it to t */
}
return t; /* return the variable t */
```

Thus, even though the above code has a sufficient level of documentation at first glance, closer inspection reveals that it is obviously not meaningful documentation that enhances the software's quality or provides information of any value to those who must maintain it.

1.2 Project Goals

This project explores the development of algorithms for analyzing Java source code documentation to assess not only the presence of comments, but also the *essence* of those comments. We propose to explore the creation of an application that assesses not only simple documentation and code formatting characteristics such as:

- reasonable white space and indentation for readability
- reasonable in-line comments explaining the purposes of discrete sections (blocks of about 5-15 related lines)
- ratio of lines and/or characters of comments to source code
- presence of author name and revision date near the top of the code

but also more sophisticated characteristics such as:

- helpful summary documentation for all classes
- comments beyond the trivial for all variables
- explanations of the purpose, parameters, pre and post conditions, and return value for all functions (methods)

Our overall goal is to produce a public Web-based application that is freely available for use by students everywhere. The application will allow programs to be uploaded for analysis, or the entire application can be downloaded to run locally (assuming one has an appropriate server to host it). The results of the analysis will be presented via a detailed Web page that shows the shortcomings in a program's source code documentation and provides specific suggestions on how to address them. While the Web implementation part of the project is relatively straightforward, the algorithms for performing the analyses are not. The development of these algorithms is the central part of this project.

2. INITIAL EXPLORATIONS

Our initial approach was to attempt to simply extract all program comments (using regular expressions under Perl) and check to see what percentage of the source code consisted of comments. While this provides some useful information, it is not a satisfactory metric on its own. Source code must not only contain a reasonable number of comments, those comments must be placed at appropriate locations. These experiments led us to realize that a certain level of source code parsing would need to be done and that we would have to be able to associate comments with specific statements or code blocks.

We also looked at an open-source tool developed by IBM for producing readability statistics of source code comments (Zlatanov, 2000). This tool provides the Fog, Flesch, and Flesch-Kincaid indices for source code comments, but again we felt that while useful, this information did not satisfy the basic goals of our project.

We next experimented with JavaML (Badros, 2000), a plug-in for IBM's open-source Java compiler, Jikes (IBM, 2003). JavaML produces output in the form of an XML file that represents the entire source file as a tree. Unfortunately, however, JavaML does not handle comments in a predictable manner, so it turned out to be of little use for our project.

Fourth, we considered using Doxygen (van Heesch, 2003), an open-source program similar to Sun's Javadoc (Sun Microsystems, 2003). While Javadoc can only produce output from Java programs, Doxygen can produce output in multiple formats from programs written in a variety of computer languages. Doxygen's capabilities seem to imply that it could do the same things as JavaML, but with much greater consciousness of the source code comments. Unfortunately, we discovered that Doxygen uses a fairly rudimentary parsing technique and is rather inflexible about where it permits comments to be placed.

3. DISCOVERING CHECKSTYLE

We then discovered Checkstyle, a open source program designed to provide comprehensive *and extensible* analysis of source code programming style (Burn, 2003). Checkstyle was originally designed to ensure that Java source code conform to Sun's official coding conventions. However, Checkstyle has a modular design, consisting of a number of different "checks." Checks can be written to verify virtually anything with respect to the source code, such as hierarchical indentation, spacing between tokens, placement of parentheses, use of curly braces and brackets, etc.

Checkstyle creates what it calls an Abstract Syntax Tree (AST), based on ANTLR, a language tool that provides a framework for constructing recognizers, compilers, and transistors from grammatical descriptions containing Java, C#, or C++ actions (Parr, 2003).

The AST represents a source file's entire contents. Each code block is a subtree of the main AST. Most importantly for our project, Checkstyle handles source code comments consistently. While comments are not accessible through the AST, they are associated with specific program lines. This allows a check to be written that, for example, can identify a class declaration and ensure that a comment exists before, after, or on the same line as that declaration. Furthermore, Checkstyle provides support for Javadoc, allowing us to automatically parse Javadoc tags and determine the presence of information such as specification of the author and version.

The code below shows a simple check that reports where class declarations occur.

```
 1  public class ReportClass extends Check
 2  {
 3    public int[] getDefaultTokens() {
 4      return new int[] { TokenTypes.CLASS_DEF } ;
 5    }
 6
 7    public void visitToken( DetailAST ast ) {
 8      log( ast.getLineNo(),
 9           "Class at: " + ast.getLineNo()
10      ) ;
11    }
12  }
```

Checks are contained within Java classes that inherit from Checkstyle's superclass, `Check`, (`extends Check` at line 1). The `getDefaultTokens()` method at line 3 is used to tell Checkstyle for which tokens we wish our `visitToken()` method at line 7 to be called. In

this case, we are asking Checkstyle to report any CLASS_DEF instances. When a CLASS_DEF token is found, its contents are used to generate an Abstract Syntax Tree (AST) which is then passed to the check's `visitToken()` method. Within `visitToken()`, we use Checkstyle's `log()` method to let Checkstyle know what we want reported back to the user. In this case, we are simply reporting the line number at which each class definition occurs.

4. PROJECT TASKS

We are currently writing checks for Checkstyle to ensure that comments exist for every class, variable, and method declaration. We are also requiring that the author of every class be specified using the `@author` Javadoc tag. A sample check that tests for the existence of class and class-level variable declarations that do not have comments on their the same line or on the immediately preceding or following line is provided in the appendix.

Once a sufficient number of checks are written, we intend to create a Web-based interface that will allow users to upload their programs to be validated. Checkstyle will run on the server side, using the checks we have written as well as some that are supplied with the Checkstyle distribution kit. Checkstyle's output will be captured, filtered, augmented, and formatted to provide users with meaningful feedback on their source code documentation, pointing out shortcomings and making suggestions for improvement.

Ideally, we would like our validator to analyze comments further and make at least a rudimentary assessment of their usefulness. At a minimum, we believe that we can implement checks that test for a minimum readability score on one or more of the major readability scales, Gunning Fox Index, Flesch Reading Ease, and Flesch-Kincaid grade level.

While it may appear that AI techniques must be employed to evaluate higher levels of usefulness, we believe that much can be done with far simpler techniques. Wilson *et al*. (1997) have demonstrated powerful abilities to analyze the quality of software requirement specifications by searching for key words, weak phrases, etc., that can serve as "quality indicators." Although their problem domain was quite different from the one on which this proposal focuses, their technique is highly relevant.

5. FUTURE WORK

We plan to expand the DocValidator to enhance its ability to recognize and evaluate the usefulness of source code comments. While requiring placement of comments in critical places that meet certain readability scores is certainly important, it is also essential that students learn to use meaningful comments. Most students new to Computer Science are able to write comments indicating on the most fundamental level what each statement does. However, students must also understand and articulate what their code actually accomplishes.

To achieve this level of analysis, we must discover precisely what makes documentation acceptable or unacceptable. We need to determine exactly what level of documentation should be required of Computer Science students.

For more advanced evaluation of source code comments, considerable work must be put into AI and/or natural language parsing techniques.

6. REFERENCES

[1] Badros, Greg J. (2000). JavaML. Posted at http://www.cs.washington.edu/homes/gjb/JavaML/ (accessed November 6, 2003).

[2] Brameld, Walter (2001). The MultipartFormData Java Class. users.boone.net/wbrameld/multipartformdata.

[3] Burn, Oliver (2003). Checkstyle. SourceForge.net. Posted at http://checkstyle.sourceforge.net/ (accessed October 16, 2003).

[4] Green, Roedy (2003). *How To Write Unmaintainable Code Documentation*. Posted at http://mindprod.com/unmaindocumentation.html (accessed May 1, 2003).

[5] IBM (2003). The Jikes Homepage. Posted at http://www-124.ibm.com/developerworks/oss/jikes/ (accessed November 6, 2003).

[6] Kabutz, Heinz M. (2002). Why I don't read your comments. *The Java™ Specialists' Newsletter*, No. 39. Posted at http://www.javaspecialists.co.za/archive/Issue039.html (accessed May 1, 2003)

[7] Parr, Terrence (2003). ANTLR. Posted at http://www.antlr.org/ (accessed November 6, 2003).

[8] Sandberg, Albert (2003). *C++ Coding Style*. Posted at http://www.flipcode.com/articles/article_codingstyle-pf.shtml (accessed May 1, 2003).

[9] Sun Microsystems (2000). *How to Write Doc Comments for the Javadoc™ Tool*. Posted at http://java.sun.com/j2se/javadoc/writingdoccomments/index.html (accessed May 1, 2003).

[10] van Heesch, Dimitri (2003). Doxygen documentation system Posted at http://www.stack.nl/~dimitri/doxygen (accessed May 1, 2003).

[11] Wilson, W.M, Rosenberg, L.H., and Hyatt, L.E. (1997). *Automated analysis of requirement specifications*. Proceedings of the 19th International Conference on Software Engineering, Boston, MA, pp. 161-171.

[12] Zlatanov, Teodor (2000). Parsing with Perl modules. IBM developerWorks. Posted at http://www-106.ibm.com/developerworks/library/l-perl-parsing/ (accessed November 6, 2003).

7. APPENDIX

The code on the following pages tests for the existence of class and class-level variable declarations that do not have comments on their the same line or on the immediately preceding or following line. It is included to provide a detailed view of how this critical component of the DocValidator will be implemented. The code is not perfect; we already know of at least one simple way to foil its comment detection scheme! However, it is provided as a more comprehensive example of the types of extensions we plan to add to Checkstyle's basic capabilities and how they will be coded.

7.1 The DocValidator Class

```
/*  File:  DocValidator.java
 *  Ben Hirsch, UMass Lowell Computer Science, bhirsch@cs.uml.edu
 *  Jesse M. Heines, UMass Lowell Computer Science, heines@cs.uml.edu
 *  Copyright (c) 2003 by Jesse M. Heines.  All rights reserved, but may be freely
 *    copied or extracted from for educational purposes with credit to the authors.
 *  updated by JMH on November 09, 2003 at 09:02 AM
 *
 *  Note: To work with this class in NetBeans, file checkstyle.all-3.1.jar must
 *  be mounted, which puts it on the classpath.
 */

package edu.uml.cs.checks ;

import com.puppycrawl.tools.checkstyle.api.* ;  // the Checkstyle API
import java.util.Map ;                          // type of returned comments

/** The contents of our check. This code is based on the discussion and samples
 *  posted at http://checkstyle.sourceforge.net/writingchecks.html. At this
 *  point our check tests for class and class-level variable declarations that
 *  do not have comments on their the same line or on the immediately preceding
 *  or following line.
 *  @author Ben Hirsch, bhirsch@cs.uml.edu
 *  @author <br/>Jesse M. Heines, heines@cs.uml.edu
 *  @version 0.2, November 9, 2003
 */
public class DocValidator extends Check
{
  /** This method lets the TreeWalker know what TokenTypes we want to catch.
   *  @return array of static ints from class TokenTypes
   */
  public int[] getDefaultTokens()
  {
    return new int[] { TokenTypes.CLASS_DEF, TokenTypes.VARIABLE_DEF } ;
  }

  /** This method is called each time one of the tokens we are interested in is
   *  found. At present, we are only checking class and variable definitions.
   *  @param ast the Abstract Syntax Tree passed by the Checkstyle API
   *  @return void
   */
  public void visitToken( DetailAST ast )
  {
    // Simple check which will be true if a comment is not present on,
    // before, or after the line of the token
    if ( ! lineHasComment( ast.getLineNo() - 1 ) &&
         ! lineHasComment( ast.getLineNo() ) &&
         ! lineHasComment( ast.getLineNo() + 1 ) )
    {
      // if it's a class, so log the lack of a comment on a class
      if ( ast.getType() == TokenTypes.CLASS_DEF )
        log( ast.getLineNo(), "Class defined without a comment" ) ;
      // if it's a class-level variable, so log the lack of a comment on that
      // variable (this check does not yet handle local variables)
      else if ( ast.getType() == TokenTypes.VARIABLE_DEF &&
                ast.getParent().getParent().getType() == TokenTypes.CLASS_DEF )
        log( ast.getLineNo(), "Class variable defined without a comment" ) ;
    }
  }

  /** This method is a simple test to determine whether a line contains a
   *  comment.
   *  @param nLineNo number of source code line to check for a comment
   *  @return true if the line contains either a C-style or C++-style comment.
   */
  private boolean lineHasComment( int nLineNo )
  {
    FileContents fcSource = getFileContents() ;   // supplied by Checkstyle API

    Map CppComments = fcSource.getCppComments() ; // supplied by Checkstyle API
    Map CComments = fcSource.getCComments() ;     // supplied by Checkstyle API

    return ( CComments.containsKey( new Integer( nLineNo ) ) ||
             CppComments.containsKey( new Integer( nLineNo ) ) ) ;
  }
}
```

Checks are classes derived from Checkstyle's `Check` class. The method `getDefaultTokens()` at line 31 is used to tell Checkstyle which tokens we want to trigger calls to our `visitToken()` method. At line 33, we tell Checkstyle to call `visitToken()` for each class definition (`CLASS_DEF`) and variable definition (`VARIABLE_DEF`).

The `visitToken()` method at line 41, called each time a class or variable definition is found, calls our `lineHasComment()` method to check whether there is a comment on the line containing the definition or on the line before or after it. If not, `visitToken()` calls Checkstyle's `log()` method to report the lack of that documentation.

Method `lineHasComment()`at line 65 uses Checkstyle's `Ccomments` and `CppComments` collections (Java `Map` objects) to examine the program's actual comments. This method returns `true` if the line passed as an argument contains a comment, and `false` otherwise.

7.2 Compiling the `DocValidator` Class

```
> javac -d . -classpath .;C:\Progra~1\Checkstyle-3.1\checkstyle-all-3.1.jar DocValidator.java
```

7.3 Creating a `jar` File from the `DocValidator.class` File

This step is required because our check must be contained with a jar file.

```
> jar -cvf DocValidator_checks.jar edu\uml\cs\checks\DocValidator.class
```

7.4 The Checkstyle Configuration XML File

This file controls which checks are run when Checkstyle is executed.

```
 1  <?xml version="1.0"?>
 2  <!--
 3    File:  DocValidator_checks.xml
 4    Jesse M. Heines, UMass Lowell Computer Science, heines@cs.uml.edu
 5    Copyright (c) 2003 by Jesse M. Heines.  All rights reserved, but may be freely
 6      copied or extracted from for educational purposes with credit to the author.
 7    updated by JMH on November 09, 2003 at 10:14 AM
 8  -->
 9  <!DOCTYPE module PUBLIC
10      "-//Puppy Crawl//DTD Check Configuration 1.1//EN"
11      "http://www.puppycrawl.com/dtds/configuration_1_1.dtd">
12
13  <module name="Checker">
14    <module name="TreeWalker">
15      <module name="edu.uml.cs.checks.DocValidator"></module>
16    </module>
17  </module>
```

7.5 Demo File for Testing the Check

Note: At present, there must be a blank line between class variable declarations.

```
1  public class demo
2  {
3    int a ;
4
5    /** comment */
6    int b ;
7  }
```

7.6 Running the `DocValidator` Check

```
> java -cp .;DocValidator_checks.jar;C:\Progra~1\Checkstyle-3.1\checkstyle-all-3.1.jar
    com.puppycrawl.tools.checkstyle.Main -c DocValidator_checks.xml demo.java
Starting audit...
demo.java:1: Class defined without a comment
demo.java:3: Class variable defined without a comment
Audit done.
```

A future stage of our project will run Checkstyle from a Web page on the server side, and then capture the program's raw output and return it to the user in a more desirable format.