ISSN:2229-6093

Kumar J et al, Int. J. Comp. Tech. Appl., Vol 2 (5), 1466-1471

Framework to measure and maintain the quality of software using the concept of Code Readability

[#1]Rambabu P, [*2]Kumar J, [*3]Praneeth S,

Abstract

Present day's software industry is using software metrics to estimate the complexity of software systems to find software cost estimation, software development control, software testing, software assurance and software maintenance. The relationship between a simple set of local code features and human concept of readability can be derived by collecting the data from 120 human annotators. This paper presents the concept of code readability and investigate its relation to software quality and also a Framework has been developed to evaluate proposed metrics and apply to the use of Bug counts, which reduces the complexity of not capturing or missing even the small parts of the meaning of the attributes they are being used to measure.. By predicting the judgment of readability it can be ensured that , the constructed automated readability measure is more effective than a human on average. Hence, this paper strongly satisfies the three measures of software quality: Changes in the code, defect log messages, and automated defect reports.

Keywords

Software Quality, Readability, Snippets, human Annotators, notations, Classifier.

1. INTRODUCTION

Readability can be defined as a human judgment of understanding a text. The critical factor in maintaining the software quality is readability and the readability of a program is related to its maintainability. Where the cost of a software product in the total life cycle the maintenance will consume around 70%. According to Aggarwal in maintenance of the software both the source code readability and documentation readability play a critical role. On other hand some researchers have noted that the act of reading code is the most time-consuming component of all maintenance activities. As of the modern software engineering, maintaining software often means evolving software and modifying existing code. Readability is another important attributes of software systems that gives substantial affect on software maintainability. Maintenance of a less readable source code is more difficult than a source code which has more readable source code. Readability Metrics are a family of software metrics that measure software complexity with taking readability into considerations. There are several uses from this automated readability metric like, helps in writing more readable software to the developers by quickly identifying code that scores poorly and also it can monitor and maintain the readability of a code which support project managers. It can even assist inspections by helping to target effort at parts of a program that may need improvement. . It can serve as a requirement for acceptance. The contributions which included in this paper are:

A. An automatic software readability metric based on local features. Our metric correlates strongly with both human annotators and also external notations of software quality.

B. A survey of 120 human annotators on 100 code snippets that forms the basis for our metric. We are unaware of any published software readability study of comparable size (12,000 human judgments).

C. A discussion of the features involved in that metric and their relation to software engineering and programming language design.

The applications of Readability Metrics indicate the readability of software systems and help in keeping the source code readable and maintainable. Finally, it can be used by other static analyses to rank warnings or otherwise focus developer attention on sections of the code that are less readable and thus more likely to contain bugs.

2. RELATED WORK

Many major projects like Linux, Java , MySQL and some popular compilers has gained incredible visibility and validation as open source model of software .“Many eyes” approach which is a source model had led to fast evolving, and easy to configure software that is being used in production environments by countless commercial enterprises. However, how exactly (if at all) do consumers of open source measure the quality and security of any piece of software to determine if it is a good fit for their stack? Few would disagree that many eyes reviewing code is a very good way to reduce the number of defects. However, no effective yardstick has been available to measure how good the quality really is. In this study, we propose a new technique and framework to measure the quality of software.

This technique leverages technology that automatically analyzes 100% of the paths through a given code base, thus allowing a consistent examination of every possible outcome when running the resulting software. Using this new approach to measuring quality, we aim to give visibility into how various open source projects compare to each other and suggest a new way to make software better. Software has transitioned from being considered as a liability to that of a re-usable asset. This shift in understanding now requires that software be written for maintainability (Troy, **1995**). Of the software quality attributes defined by **ISO-9126**, maintainability is recognized by many researchers as having the largest effect on software quality (Troy, **1995**). At the **1992** Software Engineering Productivity conference, a Hewlett- Packard executive stated that 60 – 80% of their research and development staff were involved with maintaining 40 – 50 million SLOC (Troy, 1995). Glass (**2002**) states that software maintenance consumes from 40 – 80% of the total software cost, with a mean of 60%. Boehm and Basili (**2001**) report a mean of 70%.Spinellis (**2003**) observes that programmers are poor at choosing meaningful identifier names because they find it difficult to concurrently manage the expression of programming constructs along with the managing of natural language description, say to invent identifier names. Slaughter (**2006**) reports that 80% of software quality programs fail within the first year and that these failures are not because of poor measurement techniques but due to cultural resistance on the part of the programmers and their management.

The **techniques** presented in(**2011**) this paper should provide an excellent platform for conducting future readability experiments, especially with respect to unifying even a very large number of judgments into an accurate model of readability.

3. BASIC TECHNIQUES AND PROCEDURES

Some of the major techniques which are used to code readability of software are as follows.
a. Software Quality Measurement.
b. Software Quality Management.
c. Readability Model.
d. Software Verification & Validation.

A. Software Quality Measurement

Historically software quality metrics have been the measurement of exactly their opposite—that is, the frequency of software defects or bugs. The inference was, of course, that quality in software was the absence of bugs. So, for example, measures of error density per thousand lines of code discovered per year or per release were used. Lower values of these measures implied higher build or release quality. For example, a density of two bugs per 1,000 lines of code (LOC) discovered per year was considered pretty good, but this is a very long way from today's Six Sigma goals.

We will start this article by reviewing some of the leading historical quality models and metrics to establish the state of the art in software metrics today and to develop a baseline on which we can build a true set of upstream quality metrics for robust software architecture. Perhaps at this point we should attempt to settle on definitionof ***software architecture*** as well. Most of the leading writers on this topic do not define their subject term, assuming that the reader will construct an intuitive working definition on the metaphor of computer architecture or even its earlier archetype, building architecture.

B. Software Quality Management

a. Software Quality Goals and Objectives – A discussion of how to describe, analyze and evaluate the quality goals and objectives for programs, projects, and products.

b. Software Quality Management (SQM) Systems Documentation – An overview of the various SQM system documents that a company should have in place and their relationship to each other.

c. Overview of Cost of Quality (COQ) – How to define, differentiate, and analyze COQ categories (prevention, appraisal, internal and external failure). · Problem Reporting and Corrective Action Procedures

C. Readability Model

We have shown that there is significant agreement between our groups of annotators on the relative readability of snippets. However, the processes that underlie this correlation are unclear. In this section, we explore the extent to which we can mechanically predict human readability judgments. We endeavor to determine which code features are predictive of readability, and construct a model (i.e., an automated Software readability metric) to analyze other code.

Software Verification & Validation
a. Planning Procedures and Tasks – Overview of various methods for verification and validation, including static analysis, structural analysis, mathematical proof, simulation, and dynamic analysis.

b. Reviews and Inspections – Overview of the various types of reviews and inspections, including desk-checking and inspections.

c. Testing – Overview of the various types of test, including structural integration, black box and regression.

4. DESIGNING & IMPLEMENTATION OF SYSTEM

The Snippet Extractor Eclipse plug-in is a simple and easy-to-use plug-in for storing and using code snippets throughout the Eclipse workbench. **Snippet** is a programming term for a small region of re-usable source code, machine code or text. Ordinarily, these are formally-defined operative units to incorporate into larger programming modules. Snippets are often used to clarify the meaning of an otherwise "cluttered" function, or to minimize the use of repeated code that is common to other functions.
Snippet management is a feature of some text editors, program source code editors, IDE's, and related software. It allows the user to persist and use snippets in the course of routine edit operations.

Annotators do the real work of extracting structured information from unstructured data. We can write our own annotators, use the annotators available here, and annotators will give judgment on quality and also represents feature director for verifying structural format.

Classifier is used to extract the information from annotators and feature director then it converts into human readable format. Co-verity Prevent is an advanced static software analysis tool designed to make software more reliable and secure. It relies on a combination of dataflow analysis, abstraction, and highly efficient search algorithms that can detect over 40 categories of crash-causing defects while achieving 100% path coverage.

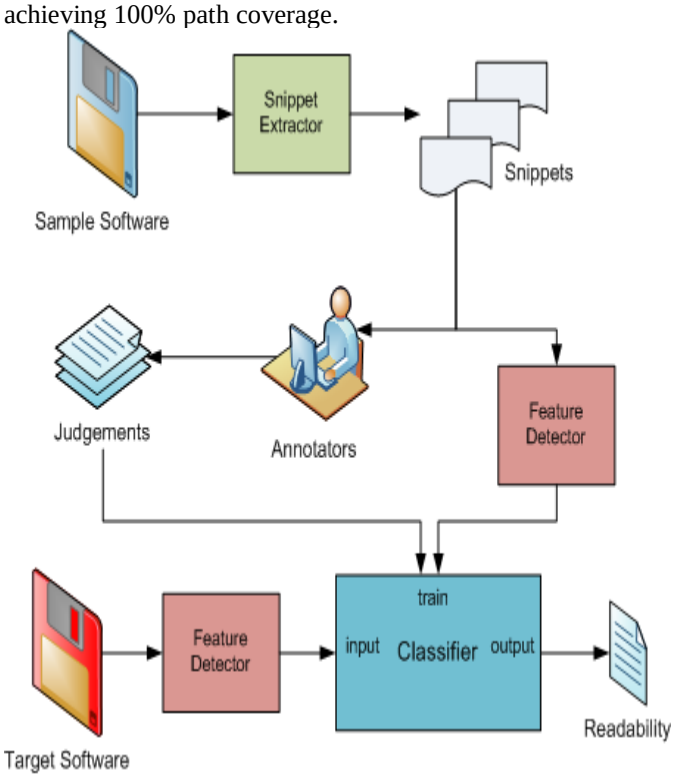

Figure: 1. the complete data set obtained for this study. Our metric for readability is derived from these judgments.

Types of defects detected include memory leaks, buffer overruns, illegal pointer accesses, use after frees, concurrency errors and security vulnerabilities. Co verity Prevent also efficiently detects hard-to-see bugs that span functions and modules. Most importantly, no changes to the code or build are required and the analysis is fast, scaling linearly with the code size.

To measure the readability and to maintain the quality of the code initially we should check the code, so a pseudo code is explained in Fig 2 to check the code and another Pseudo code is displayed in Fig 3 to find the readability of the code.

```
String Check(String fileinputtt, String
forinputt)
throws ClassNotFoundException
{
System.out.println(fileinputtt);
String ser <-- forinputt;
ser <-- ser.replace('.', ' ');
String seri[] <-- ser.split(" ");
String forinput <-- seri[0];
System.out.println("*****" + forinput);
int methodcountt <-- 0;
Class c <-- Class.forName(forinput);
Map methodList <-- new HashMap();
Method[] methods <--
c.getDeclaredMethods();
Constructor[] constructors <--
c.getDeclaredConstructors();
for (int i <-- 0; i < constructors.length;
i++)
{
System.out.println("i am from
constructors"
+ constructors[i].getName());
}
```

Figure 2: The Pseudo Code to Check the Code

```
public CodeRealaiabilityGUI()
{
}
String projectGui11(String s1, String s22)
{
String s = s1;
String s2 = s22;
System.out.println(s);
System.out.println(s2);
initializeComponent(s, s2);
this.setVisible(true);
this.setDefaultCloseOperation(JFrame.E
XIT_ON_CLOSE);
return " ";
}
```

Figure 3: The Pseudo Code for Readability of Code

5. RESULTS

The following are the screen shots of the system.

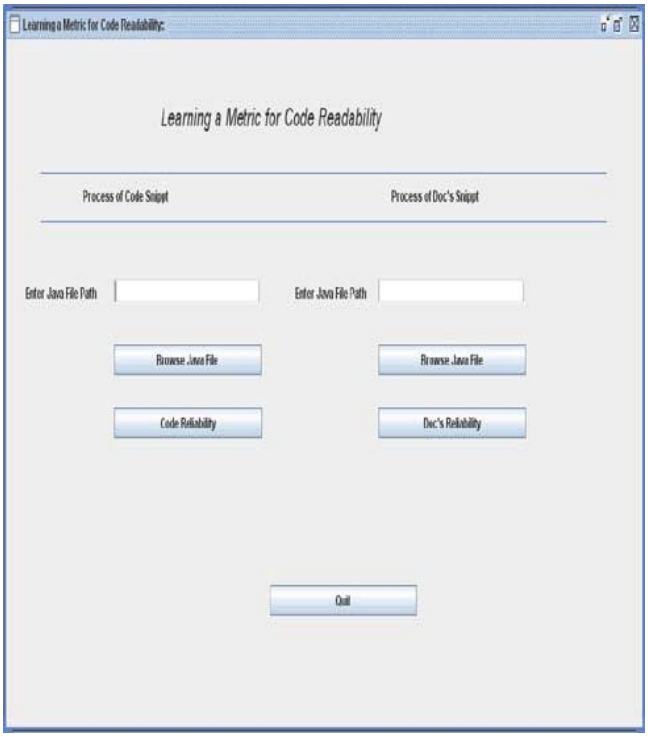

Figure 4: Processing of Code and Doc Snippet's

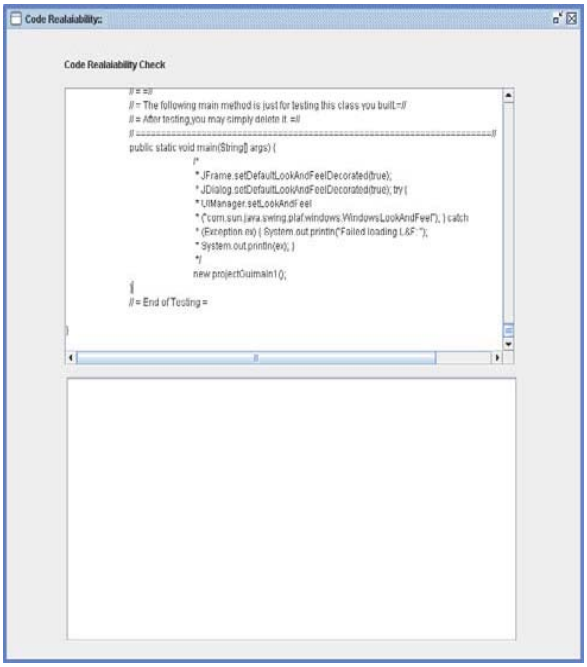

Figure 5: Generating the code readability check

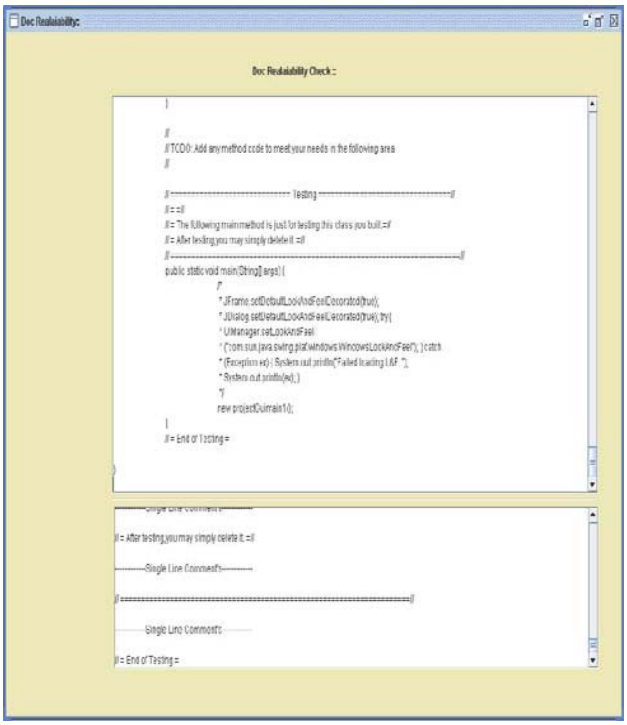

Figure 6: Generating the doc readability check

6. CONCLUSION

In this paper we have presented an automated readability measure for modeling code readability based on the judgments of human annotators. And here, we presented that it is possible to create a metric that agrees with these annotators as much as they agree with each other by only considering a relatively simple set of low-level code features. We have also observed that readability provides a significant Level of correlation with more conventional metrics of software quality, such as defects, code churn, and self reported Stability.

7.REFERENCES

[1] B.B. Bederson, B. Shneiderman, and M. Wattenberg, "Ordered and Quantum Treemaps: Making Effective Use of 2D Space to Display Hierarchies," ACM Trans. Graphics, vol. 21, no. 4, pp. 833-854, 2002.

[2] B. Boehm and V.R. Basili, "Software Defect Reduction Top 10 List," Computer, vol. 34, no. 1, pp. 135-137, Jan. 2001.

[3] R.P.L. Buse and W.R. Weimer, "A Metric for Software Readability," Proc. Int'l Symp. Software Testing and Analysis, pp. 121-130, 2008.

[4] L.W. Cannon, R.A. Elliott, L.W. Kirchhoff, J.H. Miller, J.M. Milner, R.W. Mitze, E.P. Schan, N.O. Whittington, H. Spencer, D. Keppel, and M. Brader, Recommended C Style and Coding Standards: Revision 6.0, Specialized Systems Consultants, June 1990.L.L. Giventer, Statistical Analysis in Public Administration. Jones and Bartlett, 2007.

[5] J. Gosling, B. Joy, and G.L. Steele, The Java Language Specification.Addison-Wesley, 1996.

[6] R. Gunning, The Technique of Clear Writing. McGraw-Hill, 1952.

[7] N.J. Haneef, "Software Documentation and Readability: A Proposed Process Improvement," ACM SIGSOFT Software Eng.Notes, vol. 23, no. 3, pp. 75-77, 1998.

[8] A.E. Hatzimanikatis, C.T. Tsalidis, and D. Christodoulakis,"Measuring the Readability and Maintainability of Hyperdocuments,"J. Software Maintenance, vol. 7, no. 2, pp. 77-90, 1995.

[9] G. Holmes, A. Donkin, and I. Witten, "WEKA: A MachineLearning Workbench," Proc. Australia and New Zealand Conf.Intelligent Information Systems, 1994.

[10] D. Hovemeyer and W. Pugh, "Finding Bugs Is Easy," ACMSIGPLAN Notices, vol. 39, no. 12, pp. 92-106, 2004.

[11] jUnit.org, "jUnit 4.0 Now Available," http://sourceforge.net/forum/forum.php?forum_id=541181, Feb. 2006.

[12] J.P. Kinciad and E.A. Smith, "Derivation and Validation of theAutomated Readability Index for Use with Technical Materials,"Human Factors, vol. 12, pp. 457-464, 1970.

[13] J.C. Knight and E.A. Myers, "Phased Inspections and TheirImplementation," ACM SIGSOFT Software Eng. Notes, vol. 16,no. 3, pp. 29-35, 1991.

[14] R. Kohavi, "A Study of Cross-Validation and Bootstrap forAccuracy Estimation and Model Selection," Proc. Int'l Joint Conf.Artificial Intelligence, vol. 14, no. 2, pp. 1137-1145, 1995.

AUTHORS

Mr. Rambabu Pemula received B.Tech(CSE), M.Tech(SE) from JNTU.He is a research scholar in field of Software Engineering & Data Mining .Presently he is working as Assistant Professor at Nimra Institute of Engineering & Technology, Ongole , Andhra Pradesh, India. He is having 5+ years of teaching experience in the field of Computer Science and Engineering. He can be reached at rpemula@gmail.com

Kumar Jetti received Bachelors degree in Computer science engineering from JNTUK, M.Tech in Computer science engineering from

JNTUK. He is a research scholar in field of Data Mining & Information Security. He is having experience of 4 Years in the field of Computer Science and Engineering, presently working as Assistant Professor in the department of CSE, CMR Group of Institutions, Hyderabad, R.R.Dist., A.P, INDIA. He can be reached at kumarkanna.j@gmail.com

Praneeth Sajjala received B.Tech(CSIT), M.Tech(SE) from JNTU.He is a research scholar in field of Software Engineering & Networking .Presently he is working as a Asst.Prof at Jaya Prakash Narayan College Of Engineering, Mahaboob Nagar, Andhra Pradesh, India. He is having 3+ years of teaching experience. He can be reached at sajjala.praneeth@gmail.com