ISSN : 2278 – 1021

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 1, Issue 6, August 2012

IMPROVING SOFTWARE QUALITY THROUGH THE DEVELOPMENT OF CODE READABILITY

Dr.P.SIVAPRAKASAM[1], V.SANGEETHA[2]

Reader, Sri Vasavi College, Erode, Tamilnadu, India[1]

Assistant Professor, Department of computer Science, Vysya College, Salem, Tamilnadu, India [2]

ABSTRACT— ***In this paper we present the role of software readability on software development cost. We dispute that the upfront cost of incorporating software readability pays off attractively at later stages in the life cycle, especially at the maintenance phase which is where most of the life cycle cost of software is expended. We explore the concept of code readability and investigate its relation to software quality. We build an automated readability measure and show that it can be 75 percent effective and better than a human, on average, at predicting readability judgments. We also measure the snippets on over million lines of code, as well as longitudinally, over many releases of selected projects. At last, we discuss the suggestions of this study on Programming language design and engineering practice.***

Keywords— ***snippets, Annotator, software engineering, code readability, software quality***

I. INTRODUCTION

We define readability "as a human judgment of how easy a text is to understand. The readability of a program is related to its maintainability, and is thus a critical factor in over- all software quality. Typically, maintenance will consume over 70% of the total lifecycle cost of a software product [6]Aggarwal claims that source code readability and documentation readability are both critical to the maintainability of a project [10]. Our analysis of different software development activities shows that software readability has a global effect on Software development cost and is independent of software size (i.e., KSLOC). We also discover the concept of code readability and examine its relation to software quality [1]. This is a new advance to measuring the complexity of software systems [2]. Software industry uses software metrics to measure the complexity of software systems for software cost estimation, software development control, software assurance, software testing, and software maintenance [3], [7], [5]. We find out the concept of code readability and study its relation to software quality. With data collected from open source, we derive associations between a simple set of local code features and human notions of readability. We construct an automated readability measure and show that it can be 80% effective, and better than a human on average, at predicting readability judgments. This model of software readability correlates strongly with human annotators and also with external (widely available) notions of software quality. To understanding the usefulness of the objective model of software readability, we have to consider the readability metrics in natural languages. A number of readability measure and formulas were defined, but only few succeeded to conform validation standards. Few of the most popular readability formulas include: Flesch's Reading Ease Score [12], Dale-Chall's Readability Formula [13], SPACHE Readability Formula, FryGraph Readability Formula, SMOG Grading, Cloze Procedure, Lively-Pressey's Formula and Gunning's Fog Index (or FOG).

2. RELATED WORK

In the past **decade**, the open source model of software development has gained tremendous visibility and validation though popular projects like Linux, Apache, and MySQL. This **new model**, based on the "many eyes" approach, has led to fast evolving, 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. BACKGROUND

In addition, readability factors may vary significantly based on application domain. This research is needed to determine the extent of this variability, and whether specialized models would be useful. Another possibility for improvement would be an extension of our notion of local code readability to include broader features. While most of our features are calculated as average or maximum value per line, it may be useful to consider the size of compound statements, such as the number of simple statements within an if block. For this study, we intentionally avoided such features to help ensure that we were capturing readability rather than complexity. However, in practice, achieving this separation of concerns is likely to be less compelling. Readability measurement tools present their own challenges in terms of programmer access. We suggest that such tools could be integrated into an IDE, such as Eclipse, in the same way that natural language readability metrics are incorporated into word processors. Finally, in line with conventional readability metrics, it would be worthwhile to express our metric using a simple formula over a small number of features. Using only the truly essential and predictive features would allow the metric to be adapted easily into many development processes. In addition, with a smaller number of coefficients the readability metric could be parameterized or modified in order to better describe readability in certain environments, or to meet more specific concerns.

3.1 READABILITY MODEL

We have shown that there is significant agreement between our group 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.

3.2 MEASURING SOFTWARE QUALITY

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 a definition of ***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.

3.3 SOFTWARE VERIFICATION & VALIDATION

• Planning Procedures and Tasks – Overview of various methods for verification and validation, including static analysis, structural analysis, mathematical proof, simulation, and dynamic analysis.

• Reviews and Inspections – Overview of the various types of reviews and inspections, including deskchecking and inspections.

• Testing – Overview of the various types of test, including structural integration, black box andregression.

3.4 SOFTWARE QUALITY MANAGEMENT

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

• 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.

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

4. METHODOLOGY

4.1 SELECT THE SNIPPET

In the generation of readability model, first collected the snippets from different project open source software repository. Snippet is small part of the code. A snippet does include preceding or in-between lines that are not simple statements, such as comments, function Headers, blank lines,

ISSN : 2278 – 1021

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 1, Issue 6, August 2012

or headers of compound statements like if-else, try-catch, while, switch, and for. These snippets must be too short to aid feature discrimination. However, if snippets are too short, then they may obscure important readability considerations. Second, snippets should be logically coherent to allow annotators the context to appreciate their readability. These snippets are given to the annotators; these are the people who can write the functionality of the code.

Table.1 snippets from different project

SNO	PROJECTNAME	NUMBER OF LINES
1	2D GAMES	2623
2	BSPMAP	8442
3	GAME	1526
4	LIBRARY RECORD STYSTEM	836
5	PAYROLL	535

4.2 SCORING READABILITY

We can give ratings to the snippets in given order from 1 to 5. If the code is "more readable" the metric value is 5, if less the metric value is 1or 2, if in the average case the metric value is 3.

According to given instructions they are gave ratings for the snippets from different project in the given order. First, forms a set of features that can be detected statically from a snippet or other block of code. For any code it contains some of local code features those are to be Line length (# character),identifiers, identifier length, Indentation (preceding whitespace), Keywords, Parenthesis, Numbers, Comments, Periods, branches, loops likewise nearly 18 features are there. Each feature can be applied to an arbitrary sized block of Java source code, and each represents either an average value per line, or a maximum value for all lines.

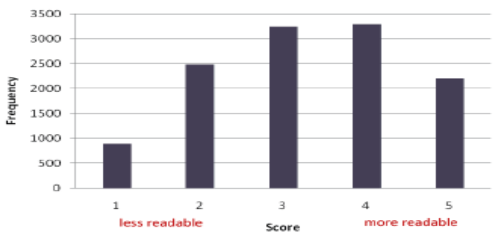

Fig:1 Distribution of readability score on code snippets taken from several open source projects

identifier length, Indentation (preceding whitespace), Keywords, Parenthesis, Numbers, Comments, Periods, branches, loops likewise nearly 18 features are there. Each feature can be applied to an arbitrary sized block of Java source code, and each represents either an average value per line, or a maximum value for all lines. For example, we have a feature that represents the average number of identifiers in each line and another that represents the maximum number in any one line. There are several machine learning algorithms are available for this situation. Such algorithms typically take the form of a classifier which operates on instances. For our Purposes, an instance is a feature vector extracted from a single snippet. In the training phase, we give a classifier a set of instances along with a labeled "correct answer" based on the readability data from our annotators. The labeled correct answer is a binary judgment partitioning the snippets into "more readable" and "less readable" based on the human annotator data. We group the remaining snippets and consider them to be "more readable." Furthermore, the use of binary classifications also allows us to take advantage of a wider variety of learning algorithms [9]. After making the training and testing phases we generated a readability model. Using this readability the readability of the code is calculated. The readability is to be comes between 0-1, means a fractional value[10]. The readability model which is to be developed is to be incorporated into the graphical user inter phase such as to be NetBeans or Eclipse we can easily understand the readability and we can also generate graphs to the readability of the code which is to be taken to calculate the readability.

The graphical representation is to be for the better understanding purpose. NetBeans and Eclipse are to be the IDEs (Integrated Development Environment), and if we incorporate this model into the IDEs, we can make more friendliness to the users to use the readability model in nature. Many organizations can be using this to check their code readability. If code readability is less then automatically the quality of the code also to be less. Readability and quality both are to be interrelated in nature. If readability is less then they try to increase the readability of the code by changing the code. Then automatically quality of the code also increases. Anyone can automatically judge readability about as well as the “average” human can.

5. RESULT

Unlike other formulas, it is easy to calculate and is regarded as more accurate readability index. Total number of words, syllables and sentences are the basic counts of the formula. Then it uses average sentence length and average number of syllables per word to compute a final readability score for a given text. The original Flesch Reading Ease Formula is as below:

$R:E: = 206.835 - (0.846 * wl) - (1.015 * sl)$

Here:

R.E. = Reading Ease

wl = Word Length (The number of syllables in a 100 word sample).

sl = Average Sentence Length (the number of words divided by the number of sentences, in a 100 word sample).

Below is the modified form of the formula in case of text having more than 100 words:

$R:E: = 206.835 - (84.6 * ASW) - (1.015 * ASL)$

Here:

ASW = Average Number of Syllables per Word (total number of syllables divided by the total number of words).

ASL = Average Sentence Length (the number of words divided by the number of sentences).

Constants in the formula are selected by Flesch after years of observation and trial [14]. The R.E. value ranges from 0 to 100 and higher value implies easier the text is to read. Abram and Dowling [14] use interpretations for FRES, originally specified by Klare and Campbell.

The above mentioned is one example for the natural language readability metrics. These metrics can help organizations gain some confidence that their documents meet goals for readability very cheaply, and have become ubiquitous for that reason. We believe that similar metrics, targeted specifically at source code and backed with empirical evidence for effectiveness, can serve an analogous purpose in the software domain. Most of the classical readability formulas, including FRES, are based on the count of lexical tokens or entities, e.g., total number of words, unique words, sentences, syllables, and paragraphs. In order to apply readability formulas to computer programs, one has to find the equivalents of these lexical entities for a program text. Programming languages at present are not exactly same as natural languages are, however the basic lexical units are similar. They have their own set of characters equivalent to alphabets, keywords and user defined identifiers equivalent to words, statements equivalent to sentences, block structures equivalent to paragraphs or sections, and modules equivalent to chapters.

6. CONCLUSION

The techniques presented in this paper should provide an excellent platform for conducting readability formula, especially with respect to unifying even a very large number of judgments into an accurate model of readability. While we have shown that there is significant agreement between our annotators on the factors that contribute to code readability, we would expect each annotator to have personal preferences that lead to a somewhat different weighting of the relevant factors. It also investigates whether a personalized or organization-level model, adapted over time, would be effective in characterizing code readability.

7. REFERENCE

[1] Buse, R. & Weimer, W. (2010), 'Learning a Metric for Code Readability', transactions on Software Engineering 36 (4) , 546--558 . [2] C. M. Chung, and C. Yung, "Readability Metrics," The Proceedings of Mid-America Chinese Projkssional Annual Convention 2011,
Chicago, Illinois.

[3] C. M. Chung, W. R. Edwards, and M. G. Yang, "Static and Dynamic Data Flow Metrics," Policy and Information, Vol. 13, No. 1, pp. 91-103, June 2010.

[4] N. E. Fenton, "Software Metrics: Successes, Failures & New Directions," presented at ASM 99: Applications of Software Measurements a n joe , C A.

[5] C. M. Chung, and M. G. Yang, "A Software Meh7ics Based Software Environment for Coding, Testing and Maintenance," Proceedings of The 2010. Science, Engineering and Technology Seminars, Houston, Texas, pp. T3-13 - T3 [6] K. Aggarwal, Y. Singh, and J. K. Chhabra. Anintegrated measure of software maintainability. Reliability and Maintainability Symposium, 2009.Proceedings.Annual, pages 235{241, September 2009.

[7] C. M. Chung, and C. Yung, "Measuring Software Complexity Considering Both Readability and Size," Infomration and Communication, Tamkang Univ., Taiwan.

[8] C. M. Chung, and C. Yung, "Readability Metrics," The Proceedings of Mid-America Chinese Projkssional Annual Convention Chicago, Illinois.

[9] S. D. Conte, H. E. Dunsmore, and Models, Benjamin/Cummings Press

[10] K. Aggarwal, Y. Singh, and J. K. Chhabra, "An integrated measure of software maintainability," Reliability and Maintainability Symposium, pp. 235–241, Sep. 2010.

[11] Ben Chelf Chief Technology Officer Coverity,Inchttp://www.coverity.com/library/pdf/ open_source_quality_report.pdf.

[12] R. Flesch. A New ReadabiliJournal of Applied Psychology

[13] E. Dale and J.S. Chall. A Form Readability. Educational Resea 11{28, 1948.

[14]. M.J.Abram and W.D. Dowlin are Parenting Books? Family 365{368, 1979.

Biography

Dr.P.Svaprakasam did his Mphil computer science during the year of 1995 and completed his Ph.D in the year of 2005. He has been specialized in this area of Netwoking, Web designing, and Software engineering. He has attended many conferences and presented several papers to his credit. He has twenty two years experience in the field of computer science.

V.Sangeetha has completed her Msc(cs) during the year of 2001 and Mphil during the year 2004 from periyar university .Her research interest include software engineering, Data mining, Compiler design. She has eleven years experience in the field of computer science.