An Integrated Approach for Measuring Software Quality and Code Readability

Jyothi G & Ch. Veera Babu
CSE Department, St’Ann’s College of Engineering and Technology, Chirala, India
E-mail: mailtojyothig@gmail.com, veeruchinka@gmail.com

Abstract - In this paper, we explore the concept of code readability and investigate its relation to software quality[1]. This is a new approach 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 explore the concept of code readability and investigate its relation to software quality. With data collected from human annotators, we derive associations between a simple set of local code features and human notions of readability. The paper continues with a framework for evaluating proposed metrics, and applies it to two uses of bug counts. Bug counts capture only a small part of the meaning of the attributes they are being used to measure.[4] 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. Furthermore, we show that this metric correlates strongly with three measures of software quality: code changes, automated defect reports, and defect log messages. Finally, we discuss the implications of this study on programming.

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 overall 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 [1]. Other researchers have noted that the act of reading code is the most time-consuming component of all maintenance activities. Furthermore, maintaining software often means evolving software, and modifying existing code is a large part of modern software engineering [1]. Readability is another important attribute of software systems that gives substantial affect on software maintainability [8]. The software systems with less readable source code are recognized as more difficult to maintain than those with more readable source code. Readability Metrics are a family of software metrics that measure software complexity with taking readability into considerations. The applications of Readability Metrics indicate the readability of software systems and help in keeping the source code readable and maintainable.

The main contributions of this paper are:

- An automatic software readability metric based on local features. Our metric correlates strongly with both human annotators and also external notations of software quality.
- 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).
- A discussion of the features involved in that metric and their relation to software engineering and programming language design.

There are a number of possible uses for an automated readability metric. It may help developers to write more readable software by quickly identifying code that scores poorly. It can assist project managers in monitoring and maintaining readability. It can serve as a requirement for acceptance. It can even assist inspections by helping to target effort at parts of a program that may need improvement. 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[10].
II. 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.

III. BASIC TECHNIQUES AND PROCEDURE

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 (V&V)

- 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 desk-checking and inspections.
• Testing – Overview of the various types of test, including structural integration, black box and regression.

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

IV. DESIGNING OF ARCHITECTURE:

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, IDEs, and related software. It allows the user to persist and use snippets in the course of routine edit operations.

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

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.

V. IMPLEMENTATION:

Coverity Prevent[11] 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. Types of defects detected include memory leaks, buffer overruns, illegal pointer accesses, use after frees, concurrency errors and security vulnerabilities. Coverity 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.

VI. CONCLUSION:

We can automatically judge readability about as well as the “average” human can. This notion of readability shows significant correlation with: Version Changes, The output of a bug finder and Self-reported program maturity. We may also learn more about software readability by looking at the predictive power of our model’s features.

In this paper we have presented a technique for modeling code readability based on the judgments of human annotators. In a study involving 120 computer science students, we have shown 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

REFERENCES:


