Software Maintainability

A Brief Overview & Application to the LFEV 2015

Adam I. Cornwell
Lafayette College
Department of Electrical & Computer Engineering
Easton, Pennsylvania
iscimena@lafayette.edu

***Abstract* — This paper gives a brief introduction on what software maintainability is, the various methods which have been devised to quantify and measure software maintainability, its relevance to the ECE 492 Senior Design course, and some practical implementation guidelines.**

Keywords — Software Maintainability; Halstead Metrics; McCabe's Cyclomatic complexity; radon

I. Introduction

Software Maintainability is an important concept in the upkeep of long-term software systems. According to the IEEE, software maintainability is defined as "the ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment [1]." Software Maintainability can be measured using various devised methods, although none have been conclusively shown to work in a large variety of software systems [6]. Software Maintenance as an actual action is divided into several different categories by the IEEE. To go along with putting Maintenance into actual practice, there exist maintenance and configuration guidelines, two of which will be discussed in this paper. Finally, the VSCADA Software team of the LFEV 2015 project has Python software written which was analyzed with an automated Maintenance analysis program. The results, along with the limitations, will be mentioned and discussed here.

II. Measurement Approaches

Several different software maintainability measurement approaches have been devised, but many include a mixture of the following [2].

For "lexical level" approaches which base complexity on program code, the following measurands are typical: individual and average lines of code; number of commented lines, executable statements, blank lines, tokens, and data declarations; source code readability, or the ratio of LOC to commented LOC; Halstead Metrics, including Halstead Length, Volume, and Effort; McCabe's Cyclomatic Complexity; the control structure nesting level; and the number of knots, or the number of times the control flow crosses. The final measurand is not as useful with object oriented programming but can still be of some use.

"Psychological complexity" approaches measure difficulty and are based on understandability and the user [3]. The nature of the file and the difficulty experienced by those working with the file are what contribute to these kinds of measures. Obviously because a user is required to interact with the code and attempt to maintain it, a downside is that the code must already be in production and use for useful metrics to be obtained. This is in contrast to the lexical level algorithmic approaches which can run solely on the code itself and so can be useful earlier in a project's cycle.

A final consideration which can be used to measure maintainability is the existence and understandability of software documentation. This belongs more in the psychological complexity category due to its dependence on the user and their understanding.

III. Halstead Metrics Overview

Halstead metrics are primarily used to estimate the number of errors in a program [4]. The three

most important metrics measured are Implementation Length, Volume, and Effort. Volume and Effort have been correlated with maintainability, although there is criticism due to the small sample sizes [6]. Implementation length is computed as follows.

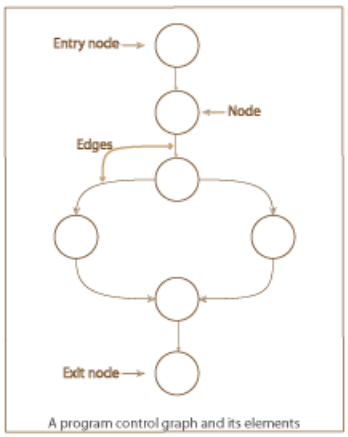
- n1 = number of unique or distinct operators appearing in a program.
- n2 = number of unique or distinct operands
- N1 = the total number of operators
- N2 = the total number of operands
- Program Length: N = N1 + N2

Once N has been computed, it is used to calculate Volume, V = N * log2(n1 + n2), and Effort as follows.

- Difficulty: D = (n1/2) * (N2/n2)
- Effort: E = D * V

IV. McCabe's Cyclomatic Complexity Overview

McCabe's Cyclomatic Complexity metric is used to determine an upper bound on the model for estimating the number of remaining defects [5]. Complexity is equated with the number of decisions a program makes so that overly-complex sections can be recoded. McCabe's metric provides the upper bound on module complexity. The graph and formula below define Cyclomatic Complexity.



- e = number of edges
- n = number of nodes
- p = number of modules
- Cyclomatic Complexity: v(G) = e – n + 2×p

V. VSCADA Generated Maintainability Results

The Python package "radon" was used on the VSCADA team's Python modules to estimate maintainability based on lexical approaches. A maintainability "grade" of A-F was given to modules based on Cyclomatic Complexity. The results were close to expectations since there were not a lot of control loops in the written code; roughly 90% of the files received an A ranking, close to the remaining 10% received a B, and three files received a C. Another measurand simply titled the "maintainability index" was also calculated on the same modules. This time the "grades" were given as A-C, with A representing a score of 20-100, B representing 10-19, and C representing 0-9. Surprisingly, all files received an A despite full team knowledge of the lack of built-in maintainability features. This represents a good example of the limits of automated maintainability metrics.

VI. Maintenance Methods

The three basic types of Software maintenance methods as defined by the IEEE include Corrective maintenance, which is done to fix flaws in the original source code or specifications; Adaptive maintenance, or software maintenance activity intended to adjust software to comply with changes in the technological environment including version upgrades, conversions, recompiles, and the re-assembly and restructuring of code; and Perfective maintenance, which is done for the purpose of expanding and improving the functionality of an existing software system [7]. Although these activities are what software maintainability metrics are supposed to be used for in determining the time taken while performing each one, most studies done which attempt to formulate a maintainability measurement system fail to mention the actual maintenance method which the system is attempting to measure [6].

VII. Practical Guidelines

NASA's Jet Propulsion Laboratory gives some helpful practical guidelines which can help with

software maintainability which will be mentioned here [8]. These include planning early and accounting for future modifications; using a modular design so that there is only one overall task for each function; using an object-oriented design if not already doing so; being sure to follow uniform conventions, including naming conventions, coding standards, comments, style, and documentation standards; using common tool sets throughout the project; and using configuration management.

In addition to this, Microsoft's MSDN Library includes some helpful guidelines on project configuration which should also help with maintainability [9]. These include not configuring everything, since if having something configured incorrectly would have a major system impact then it might be better to leave it constant; having as few separate configuration files as necessary in order to help with complexity; giving default values to optional configurable items and separating them from the required items, which helps with complexity and input error; and keeping thorough documentation which covers all configurable setting relationships, if any. The documentation should be kept in the configuration resource itself if possible, as this can help in the future when the location of original separate documentation may be lost or unknown.

VIII. Conclusion

In conclusion, this paper has covered what Software Maintainability is; different ways it can be measured; an overview of two of its important measurement methods, Halstead Metrics and McCabe's Cyclomatic Complexity; VSCADA maintainability metric results; an overview of the actual Software Maintenance methods used; and some practical guidelines which can be followed to help keep software maintainable. It is hoped that this information can be used by both current and future VSCADA teams as part of the ongoing LFEV senior project.

IX. References

[1] IEEE Std 610.121990, IEEE Standard Glossary of Software Engineering Terminology

[2] Rikard Land, "Measurements of Software Maintainability," Uppsala University, ARTES Graduate Student Conference, 2002.

[3] David L. Lanning and Taghi M. Khoshgoftaar, "Modeling the Relationship Between-Source Code Complexity and Maintenance Difficulty," *IEEE Computer Society Press*, Volume 27 Issue 9, pp 35-40, Sep. 1994.

[4] "Halstead's software metric interpreted." (n.d.). Retrieved May 15, 2015, from http://asetechs.com/NewSite2014/Products/Interpreting_Halstead_metrics.htm#BOTTOM

[5] "McCabe's Cyclomatic complexity metric interpreted." (n.d.). Retrieved May 15, 2015, from http://asetechs.com/NewSite2014/Products/Interpreting_McCabe_metrics.htm

[6] Mehwish Riaz, Emilia Mendes, and Ewan Tempero , "A Systematic Review of Software Maintainability Prediction and Metrics," *Empirical Software Engineering and Measurement*, ESEM 2009. 3rd International Symposium on Empirical Software Engineering and Measurement, pp 367-377, Oct. 2009

[7] Evelyn J. Barry, Chris F. Kemerer, and Sandra A. Slaughter, "Toward a Detailed Classification Scheme for Software Maintenance Activities," *Proceedings of the 1999 Americas Conference*, pp 726-728, 1999

[8] NASA JPL, "Software Design for Maintainability," Johnson Space Center

[9] Terry Young, "Manageability, Maintainability, and Supportability," MSDN Library, Dec. 2007, Retrieved May 15, 2015, from https://msdn.microsoft.com/en-us/library/bb896744.aspx#anchor5