Predicting Maintainability for Software Applications Early in the Life Cycle

By Cara Cuiule
PRICE Systems, L.L.C.

Abstract
Maintainability is defined as the difficulty of altering a software system's source code, thus it is tied very closely to the concept of software maintenance. The following research is an investigation of the methods used for measuring this characteristic (including the Maintainability Index). Guidance on how maintainability affects maintenance effort will be proposed. This will be followed by a discussion of which metrics could possibly predict maintainability early in the life cycle.

Introduction

As software systems age, it becomes critical to determine how much effort will be needed to perform maintenance. Maintainability has the potential to help measure maintenance effort, but no model describing a relationship between both has been perfected yet. Therefore, the purpose of this paper is twofold: to provide guidance on how to interpolate maintainability for a software system at the beginning of the life cycle, and to discuss its relationship with maintenance effort.

The first section of this paper will give a description of maintenance while the second will define maintainability and its sub characteristics. Next, a brief overview of studies that measure maintainability will be given, followed by a discussion on how maintainability is tied to maintenance effort. Then, measuring maintainability near the beginning of the life cycle will be examined. After that, additional research will be recommended followed by a conclusion.

Part 1: Definition of Maintenance

Maintainability is tied very closely to maintenance, which makes up the last stage of the Software Development Life-Cycle (SDLC) [1]. Maintenance is defined as any change made after the system's initial release [2, p. 2]. There are multiple ways software maintenance effort hours can be predicted, but none of the currently proposed methods are an industry-accepted approach [3, p. 1]. According to experts, a general rule is that this phase of the software life cycle typically costs about 60% of the entire budget of a system's lifetime [4, p. 9].

There are four types of maintenance:

- **Adaptive –** changes made to a system so it interacts properly with external parts such as government laws, hardware or third-party applications [5, p. 36]
- **Corrective –** work done to eliminate bugs within the system
- **Perfective –** adding new features to the system or adjustments to make it more maintainable
- **Preventive –** adjustments made to the system to fix underlying issues [2, p. 4]

By these definitions, maintainability is specifically tied to perfective maintenance.

Part 2: Definition of Maintainability and its Use

The International Organization for Standardization/International Electrotechnical Commission (ISO/IEC) developed international standards that define maintainability as the "degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers" [6]. Maintainability is composed of sub characteristics, which break it down further. The most recent definitions presented by the ISO/IEC are below:

- **Modularity –** how much of an issue it is to change one part of the system based on how it affects the other parts
- **Reusability –** how often parts can be used again in separate components or systems
- **Analyzability –** ease of figuring out how a change will be made and how it will affect the system
- **Modifiability –** ability to change a system without creating bugs or decreasing quality (also tied to modularity and analyzability)
- **Testability –** ability to create and meet standards for testing [6]

There are two types of quality attributes for software: internal and external. Depending on how it is viewed, maintainability can be considered either. It can be considered an internal attribute if it is *only* measured using properties within the system. When additional information outside the system is used to develop a rating, such as staff related characteristics or maintainer guidelines, then it is viewed as an external quality attribute [5, pp. 260-261]. Since the sub characteristic analyzability observes the quality in terms of how it is viewed by the maintainer, we would like to observe maintainability as an external quality attribute for the purposes of our analysis.

If viewed as an external attribute, maintainability can only be measured once the development of the system is nearly finished. This is because of the amount of source code metrics needed to measure external qualities [7]. Briand et al. [8] suggests that it is possible to use design metrics to interpolate maintainability [p. 388]. This will be discussed further in a later section.

Maintainability is calculated by workers in the software industry for multiple reasons. Maintainers working on the code use this quality attribute so they can figure out which modules are weak and need work done [9, p. 44]. This can end up saving effort and cost if a problem is caught early [10, p. 367]. In addition, it is used as a second opinion to help software engineers know if they're doing their job properly [11, p. 31].

Part 3: Notable Methods of Measuring Maintainability

There are multiple ways this metric has been measured throughout the years, using two types of models: those that have a numerical output and those that do not. This study will focus on the former. Similar to maintenance effort prediction, there is no standard way to measure maintainability [10, p. 375]. Most numerical models use some combination of the following: design metrics, source code metrics, and expert opinion. It is important to note that this section is not a comprehensive list of every significant maintainability model, but a brief overview of ones that use the distinct categories of metrics.

The most referenced model that uses source code metrics is called Maintainability Index (MI), discussed in many articles (e.g., [10], [12], and [13]). MI was initially proposed by Oman and Hagemeister, and was originally validated against professional opinion and eleven real systems [13, p.14]. A modified version of this equation is even used in programs such as Microsoft Visual Studios [14]. The original equation was modified several times after its introduction, but generally a score in the range of 65-85 indicates average maintainability. Scores below and above that range suggest low or high maintainability, respectively [12, p. 15]. The most popular deviation of their model is the following:

$$MI = 171 - 5.2 * ln(aveV) - 0.23 * aveV(g') - 16.2 * ln(aveLOC) + 50.0 * sin\sqrt{2.46 * aveCM}$$

Where the independent variables are the averages calculated for every module:

- aveV = Average Halstead Volume
- aveV(g') = Average McCabe's extended Cyclomatic Complexity
- aveLOC = Average Lines of Code (LOC)
- aveCM = Average percent of lines of comments [12, p. 15]

Note that Halstead Volume and Cyclomatic Complexity can be studied further in [15] and [16]. For a more critical review of MI, see [12] and [17, pp. 2-4].

Some researchers have looked at measuring maintainability by assigning source code metrics for its sub characteristics as defined by ISO/IEC. Members of the Software Improvement Group (SIG) created a model that assigned specific aspects of the code to the sub attributes of maintainability defined in ISO/IEC 9126. Some of the inputs were similar to those used in the MI model, such as LOC, the percent of code reused, complexity (also measured by Cyclomatic Complexity), and the percent of code covered by quality unit tests. The creators of this method claimed it was better than MI, since it indicates specifically what is wrong with the system and how can be made more maintainable [17, pp. 5-7]. This model was later tested and verified in [13].

There have also been models using only design related metrics. Design-level metrics such as control flow complexity and fan-out can predict error rate, which is related to maintainability [10, pp. 373-374]. The study detailed in [18] developed a maintainability model for modules between 1-2 KLOC of size using metrics that measure average data flow, fan out, and Cyclomatic Complexity [p. 252]. Lu et al. [19] developed a case study that found a relationship between class level diagrams and their developed maintainability measure. They also briefly

describe other studies that have measured design metrics for object oriented software [pp. 53-54].

There are other studies that have looked at creating a model with the aid of expert opinion. Bakota et al. [20] used multidimensional probability-based distributions tied with expert opinion to develop a model. Their model demonstrates the relation between lower-level attributes and the sub characteristics of maintainability to approximate a maintainability score [p. 6].

There are other models various academic and industry professionals have utilized to estimate maintainability. Many of these are detailed in reviews such as [10] and [21].

Part 4: Methodology/Guidance on How Maintenance Effort May be Affected by Maintainability

If there is a relationship between maintenance effort and maintainability, then drivers of effort could be easily related to maintainability. Unfortunately, very few studies observe the relationship between these two metrics. In theory, as the maintainability of a system or a part of one goes up, the maintenance effort associated with doing work on it goes down. But is this accurate?

As stated before, expert opinion uses maintainability to gauge maintenance effort so that problematic sections of code can be managed early and thus save cost [10, p. 367]. This method assumes that the idea proposed above is true. In addition, the experience of those in the field conclude that a program that has a higher maintainability will be associated with a lower amount of maintenance effort [22, p. 300].

Several studies that used quantitative data make assumptions about this relationship. A handful of these investigations use effort hours as a measurement to gauge maintainability itself. This is true for two experiments, which assumed that lower effort hours meant that a system was more maintainable [23, p. 107], [24, p. 135]. The only experiment that attempted to tie these two characteristics was [25], which created a model where expert opinion generated a number to measure maintainability. In this experiment, maintainability had a positive relationship with effort of design and requirements analysis, but an inverse relationship with the effort of coding a change [pp. 1, 3-4].

For the purposes of this paper, a relationship between these two metrics will be the same as what expert opinion generally surmises; it will be assumed that the higher the maintainability, the less maintenance effort hours may be required to make changes on a given system.

Part 5: Promising Metrics for Estimating Maintainability at the Beginning of the Life Cycle

As stated before, one of the purposes of this paper is to find or propose a method for estimating maintainability at the beginning of the SDLC, such as during the requirements

analysis phase or the design phase. High-level decisions, such as which personnel and programming languages are chosen, must be made before the system is implemented. For further reading on the life cycle, see [1] and [26].

Out of the models summarized in Section 3, the only ones that can be used for predicting maintainability early in the life cycle are the ones that measure maintainability in its design phase. However, it may be possible to suggest other metrics that may be useful for predicting maintainability that can also be determined during the early phases. In this section, the discussion centers on two types of metrics: source code and external.

Based on materials collected for this paper, there are two types of source code metrics that could indicate maintainability very early in the life cycle: size-related and if the programs within the system are written in an object oriented language. These will be described in detail below.

LOC is a common maintainability predictor successfully used by researchers [10, p. 373]. This makes sense since the larger the system is, the more difficult it would be to understand or make a change. One study claims it is the most effective predictor of maintainability for their experiment [23, p. 110]. There are also claims that that size is tied to maintainability mainly through the sub-characteristic of analyzability [17, p. 4]. Similarly, LOC is one of the better predictors of comprehension in an experiment done by Nishizono et al. [27, pp. 3,7]. However, some models did not find a strong correlation between maintainability and size metrics, specifically in [25, p. 2] and [28, p. 11]. Since expert findings generally tend to indicate that size metrics can be used to measure maintainability, we will assume there is a relationship between the two.

Another aspect is the use of an OO language. Previous research done by Dash et al. proposes that using an OO language leads to a more maintainable system [29, p. 209]. Lim et al.'s [24] work found that between two "real-world" systems with the same functionality, the OO system was judged to have a better maintainability, where maintainability was measured by effort and volume. However, they pointed out that their conclusion may be due to the use of superior design practices *along* with the use of an OO language [p. 136].

There are some external factors that might have an impact on effort (and thus maintainability) but sufficient research has not been conducted to support any claims. For example, there have been various environmental computing factors suggested that can impact maintenance effort such as the amount of operating systems, but these were merely theories that were never verified [30, pp. 102-103]. Another external characteristic of a maintenance team that could possibly be beneficial to maintainability is the use of a standard practice such as the Software Maintenance Maturity Model (SMmm), whose framework is meant to be a supplement to the Capability Maturity Model Integration (CMMI) [31, p. 23].

The CMMI guidelines are only aimed at improving development [32]. This could still be beneficial to predicting the maintainability of a system. A study done in [33] found that using the CMMI can generally improve the productivity and quality of an organization and its output by 60% and 50%, respectively [p. 5]. However, not all of the organizations in the study used the CMMI for improving software processes, as CMMI is a general set of guidelines that was used

for other processes as well as those related to hardware. Plus, there was no standard method across the organizations that gave data for measuring productivity and quality [pp. 20, 38]. Based on this study, it could be interpolated that if the developers of a software system use CMMI, then the system should have less faults. Less faults would improve the sub characteristic of testability, therefore making the system easier to maintain.

Since the SMmm is related to the CMMI, it can also be interpolated that organizations that use the former would have better productivity levels and thus lower maintenance effort. However, the effect that SMmm potentially has on quality or productivity has not been validated, although the model's framework has been vetted by multiple organizations [34, p. 21]. However, there has also been research claiming that improving processes (which may be done with the help of models like the two mentioned above) might improve productivity and quality of a system. It is important to note that this study only looked at one organization over a 39-month period [35, pp. 3-4].

There are also other personnel related factors that might influence maintainability. From their experience, Hayes et al. [22] offers recommendations for aspects of creating parts of a system that increase or decrease maintainability. Adhering to good coding and architecture practices, being clear with rules and standards, and focusing effort on the most important modules help to create sections of the system with good maintainability [p. 319]. These suggestions should be applicable to both the development and maintenance phases as code is still being written during both phases, although coding practices would probably have a more significant impact during development. While their advocacy for the use of good coding styles is not very detailed, other research has investigated specific coding practices. It has been proposed that using design patterns may improve quality, but various research on the topic is contradictory [36, p. 3].

Part 6: Further Recommendations

As it stands now, a universal measure of maintainability has not been developed, let alone a model that is useful in predicting maintainability early in the life cycle. In addition, a clear relationship between this metric and maintenance effort must be defined before maintainability can be used as a predictor of effort. However, expert opinion indicates that parts of the code with higher maintainability have less maintenance work done on them. As explained earlier, one study [25] found that maintainability had a positive relationship with two phases of maintenance (design and requirements analysis), and a negative relationship with one (coding). This is only a single study and more research should verify these ideas.

As stated earlier in this work, observing maintainability as an external characteristic means that it can't be measured until development is completed. There are certain aspects of a software system and development/maintenance team that could guide an estimation of maintainability. This includes size, and the use of good coding practices and processes. The use of an object oriented language might improve maintainability, but this might be only when it is used along with efficient design practices. It would be helpful to validate the relationships these attributes

have with maintainability. Only then could a model that predicts maintainability at the early phases of a project be developed.

While it may be unlikely that models attempt to predict maintainability at the first step of the SDLC, studies have shown that it can be predicted at the design stage. Design properties of a system could be discerned only once initial schematics are finished, due to metrics either taken from "design representations" or "design process" [18, p. 250]. It is realistic to expect research to be focused here; it would be useful to either validate current studies or to create improved models.

Part 7: Conclusions

Studying maintainability is challenging because there is no standard way to measure this metric. Thus, academic and industry researchers are using different models to represent it. Some studies used metrics such as the Maintainability Index to represent maintainability while others used maintenance effort hours or the amount of changes made to the system.

Since maintainability is impossible to measure until a system's creation is completed, it is difficult to try to predict maintainability before any design work is completed. Any method that could predict maintainability early in the SDLC might measure it as a function of predicted size, personnel/organizational characteristics (of both developers and maintainers), and the use of an object oriented language with proper design procedures. If this cannot be done, future works should focus on predicting maintainability after the design phase is finished, which has already proven to be possible.

Works Cited

[1] "Software Development Life Cycle." Department of Technology Services, State of Utah, 4 Sept. 2013, https://dts.utah.gov/standard/software-development-life-cycle.

[2] Pigoski, Thomas M. "Software Maintenance." *Guide to the Software Engineering Body of Knowledge: Trial Version SWEBOK: A Project of the Software Engineering Coordinating Committee*, edited by Alain Abran et al., IEEE Computer Society, 2001, pp. 1–16.

[3] Hayes, Jane Huffman, et al. "A Metrics-Based Software Maintenance Effort Model." Software Verification and Validation Lab, http://selab.netlab.uky.edu/homepage/hayesj_sw_maintenance_effort_csmr_04-Revised.pdf.

[4] Bell, G. C. "Estimating Software Maintenance Costs: The O&M Phase." 17 Sept. 2014, washingtoniceaa.com/files/presentations/SOFTWARE_MAINTENANCE_O&M_COST.pdf.

[5] Grubb, Penny, and Armstrong A. Takang. *Software Maintenance: Concepts and Practice*. 2nd ed., World Scientific, 2003.

[6] "ISO/IEC 25010:2011." *ISO - International Organization for Standardization*, Mar. 2011, www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-1:v1:en.

[7] Wüst, Jürgen. "Principles of Quality Modeling with Design Measurement." *SDMetrics: The Software Design Metrics Tool for the UML*, www.sdmetrics.com/QModel.html.

[8] Briand, Lionel C, et al. "Empirical Studies of Object-Oriented Artifacts, Methods, and Processes: State of the Art and Future Directions." *Empirical Software Engineering*, vol. 4, 1999, pp. 387–404. *Semantic Scholar*, 10.1023/A:1009825923070.

[9] Coleman, Don, et al. "Using Metrics to Evaluate Software System Maintainability." *Computer*, vol. 27, no. 8, Aug. 1994, pp. 44–49., 10.1109/2.303623.

[10] Riaz, Mehwish, et al. "A Systematic Review of Software Maintainability Prediction and Metrics." *3rd International Symposium on Empirical Software Engineering and Measurement*, 2009, 10.1109/esem.2009.5314233.

[11] Babu, P. Chitti, and K. C. K. Bharathi. "Assessment of Maintainability Factor." *International Journal of Computer Science Engineering and Information Technology Research (IJCSEITR)*, vol. 3, no. 3, Aug. 2013, www.tjprc.org/publishpapers/2-14-1370415259-5.%20Assessment%20of%20maintainability.full.pdf.

[12] Liso, Aldo. "Software Maintainability Metrics Model: An Improvement in the Coleman-Oman Model." *The Journal of Defense Software Engineering*, Aug. 2001, pp. 15–17. CiteSeerX, 10.1109/2.303623.

[13] Bijlsma, Dennis, et al. "Faster Issue Resolution with Higher Technical Quality of Software." *Software Quality Journal*, vol. 20, no. 2, 2011, pp. 265–285., doi:10.1007/s11219-011-9140-0.

[14] Conorm. "Maintainability Index Range and Meaning." *Code Analysis Team Blog*, Microsoft Developer Network, 20 Nov. 2007, https://blogs.msdn.microsoft.com/codeanalysis/2007/11/20/maintainability-index-range-and-meaning/.

[15] "Complexity Metrics." *Complexity Metrics*, Aivosto, www.aivosto.com/project/help/pm-complexity.html.

[16] "Halstead Metrics." *IBM Knowledge Center*, IBM, http://www.ibm.com/support/knowledgecenter/en/SSSHUF_8.0.1/com.ibm.rational.testrt.studio.doc/topics/csmhalstead.htm.

[17] Heitlager, Ilja, et al. "A Practical Model for Measuring Maintainability." *6th International Conference on the Quality of Information and Communications Technology*, 2007, 10.1109/quatic.2007.8.

[18] Muthanna, S., et al. "A Maintainability Model for Industrial Software Systems Using Design Level Metrics." *Proceedings Seventh Working Conference on Reverse Engineering*, Feb. 2000, pp. 248–256., doi:10.1109/wcre.2000.891476.

[19] Lu, Yao, et al. "Assessing Software Maintainability Based on Class Diagram Design: A Preliminary Case Study." *Lecture Notes on Software Engineering*, vol. 4, no. 1, 2016, pp. 53–58., doi:10.7763/lnse.2016.v4.223.

[20] Bakota, Tibor, et al. "A Probabilistic Software Quality Model." *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, 18 Nov. 2011. *IEEE Xplore*, doi:10.1109/ICSM.2011.6080791.

[21] Saini, Monika, and Mukti Chauhan. "A Roadmap of Software System Maintainability Models." *International Journal of Software and Web Sciences (IJSWS)*, vol. 2, no. 3, Feb. 2013, pp. 69–73., iasir.net/IJSWSpapers/IJSWS12-359.pdf.

[22] Hayes, Jane Huffman, et al. "Observe-Mine-Adopt (OMA): An Agile Way to Enhance Software Maintainability." Journal of Software Maintenance and Evolution: Research and Practice, vol. 15, no. 5, 2003, pp. 297–323., doi:10.1002/smr.287.

[23] Sjøberg, Dag I.K., et al. "Questioning Software Maintenance Metrics: A Comparative Case Study." *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement - ESEM '12*, 2012, pp. 107–110., doi:10.1145/2372251.2372269.

[24] Lim, Joa Sang, et al. "An Empirical Investigation of the Impact of the Object-Oriented Paradigm on the Maintainability of Real-World Mission-Critical Software." *Journal of Systems and Software*, vol. 77, no. 2, 2005, pp. 131–138. *Science Direct*, doi: 10.1016/j.jss.2004.11.004.

[25] Hayes, J.H., and L. Zhao. "Maintainability Prediction: A Regression Analysis of Measures of Evolving Systems." *21st IEEE International Conference on Software Maintenance (ICSM'05)*, 2005, pp. 1–4., doi:10.1109/icsm.2005.59.

[26] Sofia. "Software Development Process – Activities and Steps." Semantic Scholar, 2010, http://pdfs.semanticscholar.org/cca9/2212758621f52cff68454cf152b1e9e2fc6f.pdf?_ga=2.239741248.212595702.1521641029-1633884844.1519243151.

[27] Nishizono, Kazuki, et al. "Source Code Comprehension Strategies and Metrics to Predict Comprehension Effort in Software Maintenance and Evolution Tasks - An Empirical Study with Industry Practitioners." *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, 2011, doi:10.1109/icsm.2011.6080814.

[28] Hegedűs, Péter, et al. "Source Code Metrics and Maintainability: A Case Study." *Communications in Computer and Information Science Software Engineering, Business Continuity, and Education*, 2011, pp. 272–284., doi:10.1007/978-3-642-27207-3_28.

[29] Dash, Yajnaseni, et al. "Maintainability Measurement in Object Oriented Paradigm." International *Journal of Advanced Research in Computer Science*, vol. 3, no. 2, 2012, pp. 207–213., ijarcs.in/index.php/Ijarcs/article/download/1048/1036.

[30] Schneberger, Scott L. "Distributed Computing Environments: Effects on Software Maintenance Difficulty." *Journal of Systems and Software*, vol. 37, no. 2, 1997, pp. 101–116., doi:10.1016/s0164-1212(96)00107-0.

[31] April, Alain, et al. "SMmm Model to Evaluate and Improve the Quality of the Software Maintenance Process." *Eighth European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings.*, doi:10.1109/csmr.2004.1281425.

[32] "About CMMI Institute." *CMMI Institute*, ISACA, cmmiinstitute.com/about-cmmi-institute.

[33] Gibson, Diane L., et al. Software Engineering Institute, 2006, resources.sei.cmu.edu/asset_files/TechnicalReport/2006_005_001_14762.pdf.

[34] April, Alain, et al. "Software Maintenance Maturity Model (SMmm): The Software Maintenance Process Model." *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 17, no. 3, 2005, pp. 197–223., doi:10.1002/smr.311.

[35] Desharnais, Jean-Marc, and Alain April. "Software Maintenance Productivity and Maturity." *Proceedings of the 11th International Conference on Product Focused Software - PROFES '10*, Jan. 2010, pp. 1–6., doi:10.1145/1961258.1961289.

[36] Hegedűs, Péter. "Revealing the Effect of Coding Practices on Software Maintainability." 2013 IEEE International Conference on Software Maintenance, 2013, doi:10.1109/icsm.2013.99.