MODEL-BASED FACTORS TO EXTRACT QUALITY INDICATIONS IN SOFTWARE LINES OF CODE

Mohammed Abdullah H. Al-Hagery[1]

[1]Department of Computer Science, Faculty of Computer, Qassim University, KSA
dr_alhagery@Yahoo.com

ABSTRACT

The lifetime of a product depends on its maintainability. It is proved that a lot of cost spent towards maintainability. Researchers experienced difficulties to measure and improve software understandability and maintainability then software quality. In this work a proposed Model-Based Factors (MBF) is created to help developers to increase software understandability and to improve software quality. MBF estimate s/w standardization level determined by its Liens Of Code (LOC). Seventeen software products were used to validate the research results. A standardization set of source code factors was proposed. The proposed set was created and implemented by the MBF to extract the quality indications through source code segments. The MBF results illustrate positive effect of documented source code to increase the level of software quality. This effect will also improve the software readability and reusability, understandability. Furthermore, it will reduce the maintenance cost of long life software.

KEYWORDS

Software Quality, Quality of Code, Code Standardization, Code-Based Factors, software maintainability

1. INTRODUCTION

Maintainability metrics are commonly language dependent, and to compute them requires tools that typically assume access to the full definitions of the software entities [16]. A model for estimating adaptive software maintenance efforts in person hours was described in [13]. It was found that a number of metrics such as the LOC changed, and the number of operators changed is strongly correlated to maintenance efforts [14].

The work presented by Reformat et al. proposed a methodology applied to the process software maintenance data-representing evaluation of maintainability of software objects performed independently by three programmers [14]. Heitlager et al. discussed several problems with the maintainability index (MI), and they identified a number of requirements to fulfill by a maintainability model to be usable in practice. They sketched a maintainability model that alleviates most of these problems, and discussed their experiences with using such as system for IT management consultancy activities [15]. Bertoa et al. reported that they presented a set of measures to assess the maintainability of software components. Furthermore, they described the process followed to obtain and validate them. Such a process can be maintained for defining and validating measures for other quality characteristics [9]. The software system maintainability can be measured in different ways. During the previous studies, maintainability has been defined as "time required to make changes" and "time to understand, develop, and implement modification" [11], [12]. As well as, Yuming and Hareton measured the maintainability of a software system as the number of changes made to code during a maintenance period. They employed a novel exploratory modeling technique, multiple adaptive regression splines (MARS), for building maintainability prediction models using the metric data collected from two different object-oriented systems [10].

The objective of this paper is to define and create Code-Based quality indicators that can assist maintainability of long time software in future. Source code quality makes the software available for enhancement and modification later depends on customer requirements.

This paper organized as follows. After this introduction, Section 2 briefly presents some fundamentals and concepts of measuring software characteristics. Section 3 provides a summary discussion about both software modification and then section 4 discusses basics of source code quality and after that; some details of standard code are given in section 5. Section 6 describes the research methodology that includes the proposed factors, Quality indicators extraction, and the experiments performed to validate the proposed model are described in Section 6.2. Then section 7 discusses the research results. Section 8 is giving some conclusions and section 9 finishes by giving a future work headlines.

2. MEASURING S/W CHARACTERISTICS

A measure relates a defined measurement approach and a measurement scale. A measurement approach is the logical sequence of operations, described generically, used in quantifying an attribute with respect to a specified scale [5]. A measure is expressed in units, and can be defined for more than one attribute. Examples of measures for software component attributes include the number of provided interfaces, the ratio of methods per required interface, or the throughput of video frames emitted per input video frame (they correspond, respectively, to possible measures for the aforementioned attributes size, interface complexity, and performance)[9].

Measuring Software characteristics can be classified into three types; derived measures, base measures, and indicators. Base measures do not depend upon any other measure (e.g., the number of tables in the manuals).

A derived measure is derived from other base or derived measures (e.g., the ratio of methods per interface)[9]. An indicator is a measure that is derived from other measures using an analysis model according to a decision criteria to obtain a measurement result that satisfies an information need (e.g., the size of a sub-system is “medium” if it has more than 30 assemblies, provides more than 45 interfaces, and its manuals have more than 7,000 LOC. The act of measuring software is a measurement, which can be defined as the set of operations that aims at determining a value of a measurement result, for a given attribute of an entity, using a measurement approach.

Accurate software metrics-based maintainability prediction can not only enable developers to better identify the determinants of software quality and thus help them to improve design or coding, it can also provide managers with useful information to help them plan the use of valuable resources[10].

The term metric is not present in the measurement terminology of any other engineering disciplines, at least with the meaning it is commonly used in software measurement [9]. Therefore, the use of the term “software metric” seems to be imprecise, while the term “software measure” seems to be more appropriate to represent this concept. Accordingly, the term measure will be used in the following. This is also consistent with ISO/IEC and IEEE Computer Society positions which, in order to ensure both consensus and consistency with other fields of sciences, made a decision in the year 2002 to align their terminologies on measurement with the internationally accepted standards in this field. In particular, ISO-JTC1-SC7 is trying to follow as much as possible the ISO international vocabulary of basic and general terms on metrology [4].

A number of software metrics measuring maintainability has been proposed by means of theoretical and empirical studies. However, component based system presents a unique maintenance challenges. Unlike the traditional software systems, one cannot be done by viewing or changing the source codes of the component, but are restricted to reconfiguring and reintegrating components [6].

3. SOFTWARE MODIFICATION

Software modification includes four types; Corrective, adaptive, perfective, preventive. Mostly, corrective process in information systems is performed as 60% and the other three types around 40% only as shown in Figure 1. Firstly, corrective process focuses on changes made to a system to repair flaws in its design, coding, or implementation. Secondly, adaptive process concentrates on changes made to a system to evolve its functionality to changing business needs or technologies. Thirdly, perfective process can include changes made to a system to add new features or to improve performance. Finally, preventive process means changes made to a system to avoid possible future problems. Figure 1 illustrates that the most modification works are correctives [8].

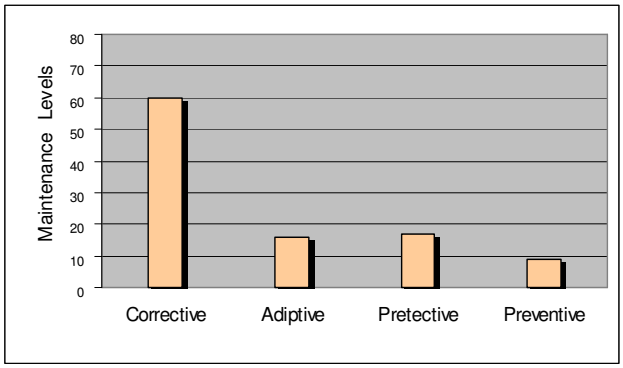

Figure 1. Software maintenance types

Many software organizations allocate 70% of information systems budget to systems modification. Methods for improving usability are inspections, automated audits of comments, test path analysis programs, dual modification of source code, modularity, and structured program logic flow, and use of pseudo-code documentation. Modification can be measured direct or indirect. Direct by calculating number of failures, time between each failure, and type of failure [8].

4. QUALITY OF SOURCE CODE

In general, there is a lack of consensus about how to define and categorize software product quality characteristics [9].There are two main types of software quality, Quality of process and quality of products. Quality of system documentation includes quality of external documentation and quality of internal documentation. Internal documentation is focusing on LOC.

The development of high-quality software must satisfy both the users' requirements and the software firm's budget [3]. Program restructuring is a key method for improving the quality of ill-structured programs, thereby increasing the understandability and reducing the modification cost [7]. Quality is one of the most sought after dimensions of the business software applications that organizations depend on today. Despite this high demand for quality, very few

studies have been done that evaluate the ongoing quality of software applications during the modification portion of the system life-cycle [2].

The development of different types of systems is challenging, because system engineers have to deal with a large number of quality requirements such as safety, security, availability, reliability, maintainability, performance and temporal correctness requirements. The fulfillment of these runtime observable quality requirements is important for customer satisfaction and project success. Often they are more important than the functional requirements. Consequently, a rigorous assessment, evaluation and analysis of the system and its models are necessary and there is a growing need to predict and evaluate these quality properties in the development process [1]. The quality metrics include adaptability, complexity of interfaces and integration test coverage, end to end test converge, reliability, and customer satisfaction [18]. Quality is measuring objectively as number of failures and defects per month [2] and quality supported by a standard implementation of code, which, will result in quality software modification. On the other hand, Kanellopoulos et al., proposed a methodology for source code quality and static behaviour evaluation of a software system, based on the standard ISO/IEC-9126. It uses elements automatically derived from source code enhanced with expert knowledge in the form of quality characteristic rankings [21].

5. STANDARD CODE

Writing standard code increase its quality, There are some reasons for writing standard code; large s/w projects are generally undertaken by correspondingly large teams of developers, readable s/w code is easier to modify, generate a consistent project wide coding style, enable to apply quality measures to the resultant s/w, support the reuse of s/w project resources, and allows developers movement from one project to another without requiring re-learning.

On the other hand, writing standard code increase code understandability, reusability and reduce the modification time. Washizaki et al., proposed a metrics suite for measuring a component's understandably, adaptability, portability and reusability based on confidence intervals that were set by statistical analysis of Java Bean components. Reusability metrics were based on a reusability model. However, these metrics do not consider architectural and application domain constraints which are important factors affecting the overall measurements [20].

The following two examples in Figure 2 and 3 present the difference between understanding of documented and undocumented code. Figure 2 shows undocumented function, there is difficulty for understanding the idea and the objective of that function, also it can not be modified easily, for doing this modification; it takes more effort than the required effort for doing the same task for the function in Figure 3. So, the code of Figure 2 is difficult to understand. In [17], Aggarwal et al., defined readability of source code as the ratio between LOC and number of commented lines.

Reuse allows us to efficiently create reuse of software components improves overall software quality, reduce software costs, and deliver software with fewer defects. Reuse allows us to efficiently create software systems from existing software artifacts rather than building software systems from scratch [22]. Reuse has been shown to increase productivity and improve quality while reducing effort and time[2]. Standard code affects positively on the following; software readability and reusability, understandability and maintainability. Reuse and reusability can also be used as quality factors for software development and maintenance. From the user's point of view there are four quality factors: maintainability, reliability, reusability and extendibility, are proposed to strongly contribute to the quality of the software product.

Understandability of the code presented in Figure 2 is lower than understandability of the code presented in Figure 3 so, the modification effort of the source code in figure 2 is higher than the modification effort of the source code in Figure 3. There is an inverse relationship between both of code understandability, maintenance and reusability.

Example1:

```
int x1( int i, char s )
{
        if( s = = "m" )
                if( I < 1000 )
                        return  0;
                else
                        if( I < 20000 )
                                return 1000;
                        else
                                return 2400;
        else
                return 2600;
}
```

Figure 2. Undocumented code

The more documentation process within the code the more quality of the product code, and this in turn gives greater opportunity for code re-use and development in the future.

Examplel 2:

```
//Function for calculating the value of tax
  int tax( int anEarning, char aStatus )
  {
     if( aStatus == 'm' )
        if( anEarning < 2000 )
           return  0;  // no tax for married, < $1000
              else
                 if( anEarning < 20000 )
                    return 500;  // married, $2000 -$20000
                 else
                    return 2400; // married, >=$20000
  // If not "married", apply single tax rate of $2600 regardless
                 else
                 return 2600;
    }  //end of  function tax
```

Figure 3. A documented code

6. RESEARCH METHODOLOGY

The research methodology includes the following steps; creation of the proposed factors and extraction of quality level indicators.

6.1. Proposed Factors

Implementation of the proposed factors is increasing code understandability, this will reduce the modification time of software. The proposed factors presented in Table 1, classified into three groups; general factors, class factors and method factors. Each factor can be assigned to any of the following values{0, 1, 2, 3, 4} where 0 indicates that the factor effect is absent, 1, 2 means factor effect is very low and low respectively, 3 means factor effect is medium and 4 means factor effect is completely satisfied.

Assigning a value y to any factor x obtained depends on conditions of formula (1):

$$\textbf{Factor(i)} = F,\ \begin{cases} F=0: & \text{if fi satisfaction within S/W code} < 10\% \\ F=1: & \text{if fi satisfaction within S/W code between } 10 - 25\% \\ F=2: & \text{if fi satisfaction within S/W code between } 25 - 50\% \\ F=3: & \text{if fi satisfaction within S/W code between } 50 - 75\% \\ F=4: & \text{if fi satisfaction within S/W code between } 75 - 100\% \end{cases} \quad (1)$$

The proposed factors in Table 1 include 20 factors for writing a high quality code to reduce the modification effort, these factors will be measured based on five levels of documentation, as shown in Table 1.

Table 1. Code Based Factors

Factor type	Index	Factor name	Levels rank (LR)				
			0	*1*	*2*	*3*	*4*
General factors	1	Variables scope and role are defined clearly			√		
	2	Code describes what is being done		√			
	3	Understand the code by reading the comments			√		
	4	Preface comments defined clearly		√			
	5	Use nouns or noun phrases for naming	√				
	6	Use alignment to enhances readability			√		
	7	End of lines comments				√	
Method's factors	1	Maximum one screen					√
	2	The meaning of return values		√			
	3	Use verbs for Function names, Get, Find, …	√				
	4	Write methods as general as possible					√
	5	The purpose of each method/function		√			
	6	Variables declarations should be left aligned				√	
	7	Method size at most one screen					√
Class factors	1	Use correct spelling in names			√		
	2	Classes should have one role only		√			
	3	Max class length between 200-300 LOC		√			
	4	Class purpose is defined clearly	√				
	5	Avoid using names that differ only by letter			√		
	6	Use nouns or noun phrases for class names		√			
		Total Satisfaction = 20	3	7	5	2	3

6.2 Quality indicators extraction

The MBF was proposed in the Formula (2). It was created based on five levels of documentation with the ranks {0, 1, 2, 3, 4}. They are used as a primary value to discover quality indicators in source code. These indicators contribute increasing software understandability and reusability of software. The maximum value of these levels is $20 \times 4 = 80$

in case of each factor satisfied completely, and their minimum value is 0 in case of each factor is satisfying nothing(undocumented code, no quality indicators). The final value of MBF is multiplied by 1.25 to change the values to degrees of percentage.

$$MBF = \left(\sum_{J=0}^{R} \left(\sum_{i=1}^{N} G_i + \sum_{m=1}^{K} M_m + \sum_{p=1}^{Q} C_p \right) \times LR_J \right) \times C \qquad (2)$$

Where, $R=4,\ n=7,\ k=7,\ q=6,\ C=1.25$

The contents of Table.2 illustrate the results of MBF when applied on seventeen software products. The final calculations performed after the measurement of the factors of Table.1, as shown in the following example,

MBF = (3×0+7×1+5×2+2×3+3×4) ×C
= 35×1.25
= 43.75

7. RESULTS DISCUSSION

After the application of the Formula (1) and use its results in the MBF presented in formula (2), the contents of Table 2 are formed for 17 software projects. These projects are small projects, it was used as a sample in the process of the test, some of these projects are documented in full and some are documented in part, and some other not documented. The purpose of the diversity of this sample of projects is to cover many different situations of the projects according to the degree of documentation for each project, and to disseminate the idea of the proposed model. Table 2 also shows the projects sample used in this research work, and at the same table, the values of each documentation level (F1, F2, F3, F4, F5) are displayed, their values were chosen based on the internal documentation of each project, depending on the Formula(1). These values show to which extent there is a variation and differences in the internal documentation of these projects.

Table 2. Code Based Factors calculations

Project	*F1*	*F2*	*F3*	*F4*	*F5*	***D-Code-Level***	***Quality-Level***
P1	4	0	0	1	15	79	19.8
P2	3	4	7	6	0	45	11.3
P3	3	2	5	6	4	58	14.5
P4	11	7	0	1	1	18	4.5
P5	9	6	2	0	3	28	7
P6	0	4	0	1	15	84	21
P7	12	2	3	0	3	25	6.3
P8	10	5	3	2	0	21	5.3
P9	8	4	3	3	2	34	8.5
P10	0	0	0	0	20	100	25
P11	0	1	10	2	7	69	17.3
P12	0	0	2	3	14	86	21.5
P13	1	3	3	4	9	71	17.8
P14	1	3	3	5	8	70	17.5
P15	0	0	0	20	0	75	18.8
P16	0	2	18	0	0	48	12
P17	19	1	0	0	0	1	0.3

On the other hand, these values give strong indicators to help determine if there is the feasibility of future development of these projects or not. For example, the projects results obtained in the last column in Table 2 with high values, especially those projects that begin with 20 and less than or equal to 25 easy to develop in future as illustrated in Figure 4, at less time and less cost.

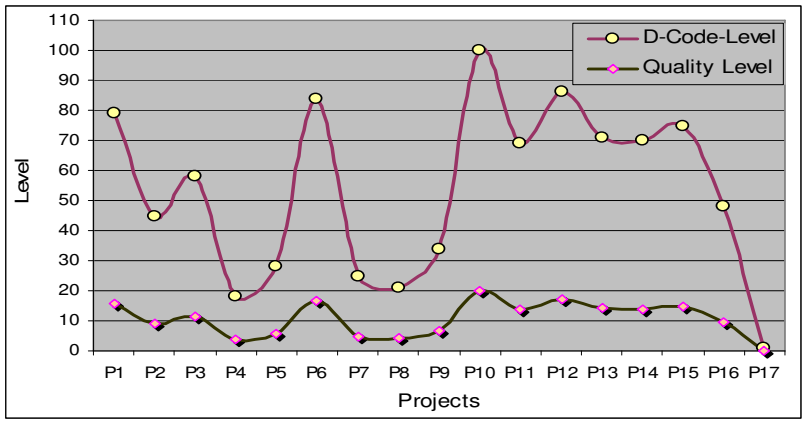

Figure 4. The relation between documentation of code and S/W quality

8. CONCLUSION

Through the discussion of the above results, found that the software documentation process according to international standards increases quality of software systems. In addition to that, it is easy to beneficiary organizations and developers to reuse the source code for these software systems and the possibility of understanding, improvement and future development at a lower cost and better results.

It can be concluded that both the proposed model and the obtained results give clear indications of the importance of documenting the source code. The results affect positively during the decision making through the maintenance of long life software. Well-documented software reduces the cost of this type of software and transition of artefacts to another team members or the development team itself.

9. FUTURE WORK

Many further points can be added to expand this work. First, increasing the number of used projects for the test. Second, new factors or criteria can be added for measuring software maintainability then designing a software tool to generate a standard code style. The utility input is undocumented source code and its output is full documented source code.

REFERENCES

[1] Clements, P.C. & Kazman, R. & Klein, M, (2001) "Evaluating Software Architectures: Methods and Case Studies", Addison-Wesley Longman.

[2] Ghods, M. & Nelson, K. M, (1998) "Contributors to quality during software maintenance Decision Support Systems", Vol, 23, p.361–369.

[3] DeMillo, R. A. & Liption, R. J & Perlis. A. J, (1981) "Software Project Forecasting", MIT Press, Software Metrics, Cambridge, MA.

[4] ISO VIM (Second ed.), (1993) "International Vocabulary of Basic and General Terms in Metrology", International Standards Organization, Geneva, Switzerland.

[5] ISO/IEC 15939, (2002) "Software Engineering–Software Measurement Process".

[6] Voas. J, (1998) "Maintaining component based systems", IEEE Software 15 (4), p. 22–27.

[7] Lung, C & Xu, X & Zaman, M & Srinivasan, A, (2006) "Program restructuring using clustering techniques", The Journal of Systems and Software 79, p. 1261–1279.

[8] Hoffer, J. A & George, J. F & Valacich, J. S, (2008) "Modern Systems Analysis and Design, chapter 16", fifth Edition.

[9] Manuel, F. B & Jose, M. T. & Antonio, V, (2006) "Measuring the usability of software components", The Journal of Systems and Software 79, 427–439.

[10] Zhou, Y & Leung, H, (2007) "Predicting object-oriented software maintainability using multivariate adaptive regression splines", The Journal of Systems and Software 80, p. 1349–1361.

[11] Gibson, V. R. & Senn, J. A, (1989) "System structure and software maintenance performance", Communication of ACM 32 (3), p. 347–358.

[12] Rising, L. S, (1992) "Information hiding metrics for modular programming languages", PhD dissertation, Arizona State University.

[13] Hayes, J. H. & Patel, S. C & Zhao, L, (2004) "A Metrics- Based Software Maintenance Effort Model", In Proc Of the Eighth European Working Conference on Software Maintenance and Reengineering (CSMR'04), p. 254-260.

[14] Reformat, M & Kapoor, A and Pizzi, N. J, (2006) "Software Maintenance: Similarity and Inclusion of Rules in Knowledge Extraction", Proceedings of the 18th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'06).

[15] Heitlager, I & Kuipers, T & Visser, J, (2007) "A Practical Model for Measuring Maintainability, ", Sixth International Conference on the Quality of Information and Communications Technology, IEEE.

[16] Hindle, A & Godfrey, M. W & Holt, R. C, (2009) "Reading beside the lines: Using indentation to rank revisions - by complexity", Software Architecture Group, University of Waterloo, Waterloo-Ontario, Canada Science of Computer Programming, 74, p. 414-429.

[17] Aggarwal, K. K. & Singh, Y. & Chhabra, J. K, (2002) "An Integrated Measure of Software Maintainability", In Proceedings of Annual Reliability and Maintainability Symposium, IEEE.

[18] Mahmood, S & Lai, R & Kim, Y. S & Kim, J. H & Park, S. C & Oh, H. S, (2005) "A survey of component based system quality assurance and assessment", Journal of Science Direct, Information and Software Technology 47, p. 693–707.

[19] Kim, Y. & Stohr, E. A, (1998) "Software reuse: survey and research directions, Journal of Management Information Systems", forthcoming.

[20] Washizaki, H. & Yamamoto, H. & Fukazawa, Y, (2003) "A metrics suite for measuring reusability of software components, In Proceedings of Nineth International Software Metrics", Symposium (METRICS'03), Sydney, Australia.

[21] Kanellopoulos. Y, Antonellis. P, Antoniou. D, Makris.C, Theodoridis. E, Tjortjis. C, and Tsirakis.N, (2010) "Code Quality Evaluation Methodology Using The Iso/Iec 9126 Standard", International Journal of Software Engineering & Applications (IJSEA), Vol.1, No.3.

[22] P.Shireesha and .S.S.V.N.Sharma, (2010) "Building Reusable Software Component For Optimization Check in ABAP Coding", International Journal of Software Engineering & Applications (IJSEA), Vol.1, No.3.

Author

Mohammed A. H. Al-Hagery, Ph.D

Al-Hagery received his B.Sc in Computer Science from the University of Technology in Baghdad Iraq in 1994. He got his MSc in Computer Science from the University of Science and Technology Yemen in 1998. Al-Hagery finished his Ph.D. In Computer Science, field of Software Engineering from the Faculty of Computer Science and Information Technology, University of Putra Malysia (UPM) in 2004. From 2004 to 2007 he was the head of Comp Science Department at the Faculty of Science and Engineering, USTY, Sana'a. From 2007 to this date, he is a staff member at the Faculty of Computer, Qassim University in KSA. He published more than 10 papers in international journals.