Software metrics and Maintainability Relationship with CK Matrix

Sandeep Srivastava,
Asst. Prof., CS & IT Department
Sobhasaria Engineering College, Sikar, Rajasthan

Abstract:-This paper presents the relation between software metrics and maintainability and the metrics which characterise the ease of the maintenance process when applied to a specific product. The criteria of maintainability and the methods through which these criteria are understood and interpreted by software programmers are analysed. Surveys and examples that show whether software metrics and maintainability are correlated are also presented. In this paper, software metrics which have been proposed for maintainability were used and applied to program languages. The aim of this paper is to determine up to what point and in which cases can we rely on software metrics in order to define the maintainability of a software product.

Keywords – Software Quality, OO metric, Software quality measurement .

I. INTRODUCTION

What is meant by maintainability of software is the degree, to which it can be understood, corrected, adapted and/or enhanced [1]. Of course, in an ideal world, where the principles and instructions of software quality assurance are followed to the letter, maintainability would not be so essential to software development. Moreover, the decrease of the cost of maintenance is a basic aim of software quality programs. However, in reality, the principles of software quality assurance are not always applied as stringently as needed, or even if they are, the role of maintainability is still essential.

Although maintainability is one of those quality factors which is normally only of indirect interest to the customer, figures gathered over the past few decades have indicated that software developers can spend as much as 75 per cent of their project budgets on software maintenance. In other words, software maintenance is the most costly phase of the software life cycle. Furthermore, it is true that software requirements do change, but the impact of change varies with the time at which it is introduced. When changes are requested during software maintenance, the impact cost may be greater than 10 times the impact cost derived from a change requested during software design, i.e. the cost to maintain one line of source code may be more than 10 times the cost of the initial development of that line.

As a result, a software company must contrive various ways to measure the ease of the maintenance process, not only to reduce the cost of maintainability but to ascertain whether the maintenance of a specific software product is worthwhile or not. Several software metrics relative to the criteria of maintainability have being proposed and their primary goal is to determine the degree of maintainability of a software product.

The aim of this paper is to twofold; firstly, it presents the relation between software metrics and maintainability and, secondly, it proposes in which cases, in which way and under which circumstances a software production company can rely on the software metrics results to perceive the maintainability of their products. Finally, examples derived from our surveys on programmers currently participating in maintenance projects are presented.

In the next chapter, the different types and opinions of maintainability and the approaches for measuring this quality characteristic are presented. In chapter 3, the use of internal software metrics to improve maintainability is analysed. Finally, in chapter 4 the results of our surveys for measuring maintainability are presented.

II. Maintainability Towards Software Evolution

2.1 The types of maintenance:-

Software maintenance accounts for more effort than any other software engineering activity. Although it has until very recently been the neglected phase in the software engineering process, maintainability is a key goal that guides its steps. Four types of maintenance are performed on software: a)Corrective maintenance acts to correct errors that are uncovered after software is in use, b) Adaptive maintenance is applied when changes in the external environment precipitate modifications to software, c) Perfective maintenance incorporates enhancements that are requested by customers and d) preventive maintenance improves future maintainability and provides a basis for future enhancement.

According to the Factor–Criteria–Metrics model, the maintainability factor is comprised of consistency, simplicity, conciseness, self-descriptiveness and modularity [2]. According to the IEEE standard for quality metrics, maintainability can be decomposed to correct ability, testability and expandability [3]. Finally, according to the ISO–9126 standard, maintainability consists of analyzability, changeability, stability and testability [4].

2.2 Opinions about handling the maintenance process:-

As more programs are developed, the amount of effort and resources expended on software maintenance is growing. The maintainability of software is affected by many factors, such as the availability of qualified software staff, the ease of system handling, the use of standardised programming languages etc. Inadvertent carelessness in design, implementation and testing has an obvious negative impact on the ability to maintain the resultant software [5]. Additionally, some software organisations may become maintenance-bound, unable to undertake the implementation of new projects, because all their resources are dedicated to the maintenance of old programs.

The programmers' opinion regarding the level of maintainability usually varies. However, generally, according to their opinion a program with a high level of maintainability should consist of modules with strong cohesion and loose coupling, readable, clear, simple, well–structured and sufficiently commented code, having a strictly concurrent style and well–conceived terminology of their variables. Furthermore, the implemented routines should be of a reasonable size (preferably fewer than 80 lines of code), with limited fan–in and fan–out. Straightforward logic, natural expressions and conventional language should be followed [6]. Finally, the declaration and the implementation part of each routine must be strictly separated.

According to the program managers' opinion of maintainability, the implemented programs should consist of documented and introspective modules. Moreover, a framework for the versioning of these modules should be used. Program managers always aim at the limitation of the effort spent during the maintenance process. They also focus on the high reusability of one program, in order to increase the feasibility of the ease with which software modules in this program can be moved to another.

Maintainability is a quality factor that is normally of indirect interest to the customer. However, very few customers indeed include directives about maintainability in the directions they give a software developer, apart, of course, from those developers who intend to carry out the maintenance of a system themselves. As Ince says: *"Paradoxically, one of the indicators that a software developer has that a good system has been developed is to receive numerous requests from a customer for modifications which arise from changes in requirements. Often these are framed in terms of new functions"* [7]. And as we already found, customers may revise their opinion of the quality of a product very often [8]. Nowadays, because of the high demand of the customers from successful software systems and external circumstances change, a high level of modification can be attributed to changes in requirements.

2.3 Approaches for measuring maintainability:-

There are two broad approaches for measuring maintainability, reflecting external and internal views of the attribute. Maintainability is an external quality factor, as it clearly depends not only on the product itself but also on the programmer performing the maintenance. The external and more direct approach to measuring maintainability is to define measures of the maintenance process and then collect the opinion of the programmers who participate in this process. Unfortunately, the external approach is time consuming and depends on conducting a survey, which may be of a high cost. Additionally, this kind of surveys may require the improvement of the accuracy and the interpretation of their input and their derived results [9]. The alternative internal approach is to use internal metrics and hope that these are predictive of the programmers' opinion of the maintainability of a software product. Furthermore, in this approach, the measures can be gathered earlier and easier.

During the software maintenance process the structure of the system usually degrades. Random patches applied by members of the maintenance department often result in a low quality system structure. Gradually, the system becomes more and more difficult to maintain. Software metrics can also be used control the degradation of a system in order to keep the quality of the maintenance process at a high level.

III. USE OF METRICS TO IMPROVE SOFTWARE MAINTAINABILITY

Software maintainability is a difficult factor to quantify. However, it can be measured indirectly by considering measures of design structure and software metrics. It is also claimed that logical complexity and program structure have a strong correlation to the maintainability of the resultant software [10,11]. Moreover, as Fenton says *"Good internal structure provides good external quality"*. But what must be determined is up to what point and in which cases can we rely on software metrics in order to define the maintainability of a software product [12].

Considering the different opinions of implementing a product with a high level of maintainability, it is noticed that software metrics can estimate only the programmers' opinion of maintainability. It is already observed that although software metrics are intertwined with the customers' opinion of user–oriented quality characteristics, satisfaction of internal quality standards does not guarantee a priori success in fulfilling the customers' demand on quality. They can only detect possible causes of low product quality [13]. However, in this case, when comparing the score of software metrics with the programmers' opinion of maintainability, it is concluded that there are highly correlated. In other words, in most cases, not only can software metrics define the modules with low level of maintainability, but they can also predict the high level of maintainability of a product without needing to conduct a survey in order to directly measure the opinion of this quality factor.

The selection of the appropriate internal metrics for measuring the maintainability of one specific product depends on the nature of this product and the programming language used during its implementation. Traditional, broadly used metrics, like Halstead's software science metrics [14], cyclomatic complexity [15], Tsai's data structure complexity metrics [16], lines of code, lines of comments, fan–in, fan–out, etc. can always be applied in order to estimate the level of the maintainability of a software product. Additionally, for programs implemented in a specific type of programming languages, metrics that can be applied only to this type can also be used in order to have a more reliable and acceptable measurement. For example, in the case of object–oriented programming languages, the metrics that can also be used are: weighted methods per class, lack of cohesion of methods, coupling between objects, depth of inheritance tree, number of children, etc. [17].

Furthermore, some of the proposed metrics allow one to focus on just the issues of interest. For example, they directly evaluate the size, the sum of nested levels, the fan–in and the fan–out of a routine, which are all essential criteria in order to form an opinion of the maintainability of this routine. Moreover, software metrics are also aimed to being acceptable indicators not only in order to estimate the structure, the cohesion and the coupling of the modules of a software program, but also for the estimation of the readability, the clearness, the sufficiency of the comments and the simplicity of their code.

The use of internal software metrics is meaningful only when they are applied to large fragments of code. Measurements must not be restricted only to small parts of a software program in order to draw useful conclusions. However, using a framework of metrics allows the programmers to locate modules and routines with a low level of maintainability. The use of metrics assists the programmers to inspect their code and make the necessary corrections and improvements during the implementation phase. Although it is true that internal measurements may delay the duration of this phase, this is much more preferable than the confrontation of the effort of the maintenance process. In other words, the achievement of acceptable score in software metrics must be a basic constraint in order to finish the implementation of a product.

The boundaries that must be set to the score of each metric used in the framework for measuring maintainability depend on the kind of application that is going to be produced, the programming language that is going to be used, the profile and the objectives of the software company, the programmers' experience, etc. For example, in order to produce a parser or a grammar checker, it is expected that some of the modules of the product will have a very big cyclomatic complexity and sum of nested levels. This is not in and of itself distressing, although in other cases, it would prohibit the transition to the following phase of the life cycle and would even force the programmers to rewrite these modules.

Furthermore, the different criteria that maintainability is comprised of may sometimes conflict. Simplicity and conciseness may antagonise analysability, changeability and expandability. For example, the routines A and B presented in figure 1 and written in Object Pascal are of the same underlying purpose.

```
Routine A
Procedure TChoiceForm.ListView1Change(Sender: TObject; Item: TlistItem; Change: TitemChange);
Begin
        If  ListViewChoices.Selected <> nil then
             Begin
                  DelBtn.Enabled:=True;     MoveUp.Enabled:=True;     MoveDown.Enabled:=True;
             End
        Else
             Begin
                  DelBtn.Enabled:=False;    MoveUp.Enabled:=False;    MoveDown.Enabled:=False;
             End;
End;
```

```
Routine B
Procedure TChoiceForm.ListView1Change(Sender: TObject; Item: TListItem; Change: TitemChange);
Var HasSelection : boolean;
Begin
        If ListViewChoices.Selected <> nil then HasSelection:=true
        Else HasSelection:=false;
   DelBtn.Enabled:=HasSelection;  MoveUp.Enabled:=HasSelection;  MoveDown.Enabled:=HasSelection;
End;
```

Figure 1: Example with similar routines

According to Halstead metrics, routine B is better than routine A, because it has a higher program level (L_A=0.2 and L_B=0.33) and a higher language level (l_A=4.45 and l_B=8.98). Furthermore, routine B has fewer lines of code than routine A. This is, of course, a very small part of code from which to make acceptable judgements. However, it is obvious that although routine B is simpler and more concise, routine A is easier to be analysed, changed or expanded. The final choice must be made in accordance with which criterion of maintainability is mainly focused on.

A software company that wants to apply a framework of software metrics in order to control the level of the maintainability of its products must firstly choose a set of appropriate metrics for each product. Secondly, it must arrange the boundaries of each metric that characterise a code as accepted, as required for corrections or as rejected. Then, after conducting the internal measurements, if any module or part of the program fails according to the predetermined metrics limits, it must be inspected by the programmers in order to decide the appropriate step. These emergent interventions prevent the programming team from facing serious problems and making great efforts during the maintenance phase. Moreover, if the metrics' results arise great deviation from the accepted level in the majority of the modules, the software product is preferable to be re–designed. Therefore, project managers must be particularly mindful when they determine the constrains of each metric used in the applied framework.

IV. RESULTS FROM SURVEYS

In order to examine the correlation between software metrics and maintainability, surveys on programmers currently participating in maintenance projects where conducted. 5 big projects, each of which consisted of 10 to 20 modules, were measured according to their maintainability level with both approaches; directly, where evaluation responds were gathered from 50 programmers, and indirectly using internal metrics. Each module of each project was measured separately.

In order to measure the programmers' opinion of the maintainability of the projects (external measurement), an appropriate questionnaire was created, which included questions (in multiple–choice format) for every single factor of maintainability. In other words, the programmers evaluated the modules according to their consistency, simplicity, conciseness, expandability, correctability, etc. Moreover, internal measurements were automatically conducted by using the software measurement and metrics environment ATHENA [18]. This environment provides completely automated measurements and therefore, the collection of raw data was effortless, as it should be in any enterprise that collects developer-oriented measurements.
From the results of these surveys it was obvious that software metrics' score and programmers' opinion of maintainability were highly correlated. In detail, modules with high internal measurements scored equally high in external measurements. Additionally, modules that failed in software metrics scored pure according to the programmers' opinion of maintainability. Examples of modules of each case are presented in figures 2 and 3, where the lines of code (LOC), the program level (L), the language level (l) and the cyclomatic complexity (Vg) of each routine of the modules are presented. However, a large number of different metrics were used in these surveys. Because of the different nature of the 5 projects, some of the chosen metrics could not be applied in all of them. The histogram next to each module presents the programmers' opinion of maintainability, according to their responses to the questionnaires, where the horizontal bar
represents their opinion's score and the vertical bar represents the number of programmers.

Module 'Correct'				
Routine	**LOC**	**L**	**l**	**Vg**
Correct	23	0.26	3.59	3
LowerGRword	17	0.23	1.98	2
Find	17	0.42	8.54	1

Module 'Menu'				
Routine	**LOC**	**L**	**L**	**Vg**
Executive	28	0.04	1.58	4
Getvmn	98	0.01	0.43	24
MenuWidth	12	0.09	1.19	2
Dimension	9	0.07	1.04	2
Light	9	0.15	4.02	1
Vlight	8	0.05	1.51	2
SelectItem	64	0.02	0.90	11

Figure 2: Example of module with high score

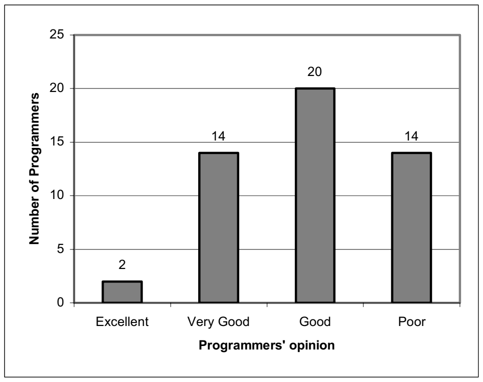

Figure 3: Example of module with low score

Furthermore, it was also observed that some modules that failed in metrics scored with a great variation according to the programmers' opinion of maintainability. In other words, low internal measurements do not necessarily imply low external measurements. This variation was observed not only because of the experience of the programmers in the programming language that these particular modules were implemented or because of their personal capabilities, but mainly because of the nature of the application these modules belong. Figure 4 presents a module of this type. This observation is in conflict with the relation between software metrics and the customers' opinion of user–oriented quality characteristics. In the case of customers, metrics can only detect possible causes for low product quality and satisfaction of internal quality standards does not guarantee success in fulfilling the customers' demand on quality.

Programmers should keep the metrics score of their programs within acceptable limits. Generally, the higher the score they achieve, the less effort will be spent during the maintenance process. This conclusion was also observed when measuring different versions of the same product, where the subsequent ones scored higher in internal measurements than their former.

Module 'Userdict'				
Routine	**LOC**	**L**	**I**	**Vg**
Hash	12	0.04	0.36	3
OpenUserDict	5	0.33	2.18	1
CloseUserDict	5	0.66	2.15	1
CreateUserDict	16	0.07	1.55	4
Lookup	17	0.05	1.02	7
AddInUserDict	31	0.03	0.86	13
FindInUserDict	7	0.40	2.02	2

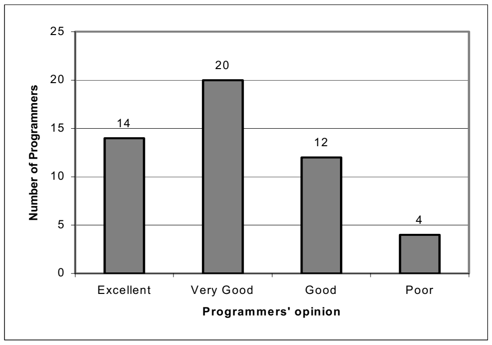

Figure 4: Example of module with low internal score and high external score

However, if a module fails in internal measurements it is preferable not to reject it automatically although it usually fails in external measurements about maintainability, too. On the contrary, as low score in metrics sometimes is expected, the programmers should inspect carefully their code and decide whether it needs to be corrected (or even re-written) or not.

V. CONCLUSION

Software metrics provide an easy and inexpensive way to detect and correct possible causes for low product quality according to the maintainability factor as this will be perceived by the programmers. Setting up measurement programs and metric standards will help in preventing failures before the maintenance process and reduce the requisite effort during that phase. Internal metrics are highly correlated with the programmers' opinion of maintainability. However, dissatisfaction with internal quality standards may not necessarily result in low level of maintainability although it is usually expected. In that case, it is preferable that, despite what internal measurements indicate, the final judge for the maintainability of the produced software is the programmer.

REFERENCES

[1] Pressman, R, '*Software Engineering. A Practitioner's Approach*', Fourth Edition, McGraw Hill, isbn: 0-07-709411-5, 1997.

[2] McCall, A, Richards, K, Walters, F, '*Factors in Software Quality*', US Rome Air Development Centre Reports NTIS AD/A-049 014, 015, 0,55, 1977.

[3] IEEE, ‘*Standards for a Software Quality Metrics Methodology*’, P-1061/D20, IEEE Press, New York, 1989.

[4] ISO9126, ‘*Software Product Evaluation - Quality Characteristics and Guidelines for their Use*’, ISO/IEC Standard ISO-9126, 1991.

[5] Kopetz, H, ‘*Software Reliability*’, Springer–Verlag, p.93, 1979.

[6] Kernighan, B, and Pike, P, ‘*The Practice of Programming*’, Addison Wesley, isbn: 0-201-61586-X, 1999.

[7] Ince, D, ‘*ISO 9001 and Software Quality Assurance*’, McGraw Hill, isbn: 0-07-707885-3, 1994.

[8] Stavrinoudis, D, Xenos, M., Peppas, P, and Christodoulakis, D, ‘*Measuring User's Perception and Opinion of Software Quality*’, 6th European Conference on Software Quality, EOQ-SC, Vienna, pp. 229-237, 1999.

[9] Xenos, M, and Christodoulakis, D, ‘*Software Quality: The user's point of view*’, International Conference on Software Quality and Productivity, Hong-Kong, Sponsored by IFIP, Published by Chapman and Hall Publications, ‘SOFTWARE QUALITY AND PRODUCTIVITY: Theory, practice, education and training’, Edited by Matthew Lee, Ben-Zion Barta and Peter Juliff, isbn: 0-412-629607, pp 266-272, 1994.

[10] Kafura, D, and Reddy, R, ‘*The Use of Software Complexity Metrics in Software Maintenance*’, IEEE Trans. Software Engineering, vol. SE-13, no. 3, pp. 335-343, 1987.

[11] Rombach, D, ‘*A Controlled Experiment on the Impact of Software Structure on Maintainability*’, IEEE Trans. Software Engineering, vol. SE-13, no. 3, pp. 344-354, 1987.

[12] Fenton, N, Pfleeger, S, ‘*Software Metrics A Rigorous & Practical Approach*’, Second Edition, Thomson Computer Press, isbn: 1-85032-275-9, 1997.

[13] Xenos, M, Stavrinoudis, D, and Christodoulakis, D*,* ‘*The Correlation Between Developer-oriented and User-oriented Software Quality Measurements (A Case Study)*’, 5th European Conference on Software Quality, EOQ-SC, Dublin, pp. 267-275, 1996.

[14] Halstead, H, ‘*Elements of Software Science*’, Elsevier Publications, N-Holland, 1975.

[15] McCabe, J, “*A complexity measure*”, IEEE Transactions of Software Engineering, SE-2(4), 1976.

[16] Tsai, T, Lopez, A, Rodreguez, V, Volovik, D, “*An Approach to Measuring Data Structure Complexity*”, COMPSAC86, pp 240-246, 1986.

[17] Hudli, R, Hoskins, C, Hudli, A, ‘*Software Metrics for Object Oriented Designs*’, IEEE, 1994.

[18] Tsalidis, C, Christodoulakis, D, and Maritsas, D, ‘*Athena: A Software Measurement and Metrics Environment*’, Software Maintenance: Research and Practice, 1991.