Injecting Quantifiability to Promote Software Maintenance

Engr. Syed Rizwan Ali 1st, Azmat Khan 2nd, Muhammad Shahid Khan 3rd, Bilal Muhammad Iqbal 4th

Abstract — Generally, the power of quantifiability is underestimated. Life cycle costs are dependent on software system maintenance costs. Systems engineering can improve with a more structured effort, by setting proper targets instead of applying conventional customs that seem right but in actual may not be sufficient. The maintenance process becomes more efficient once the maintainability process is quantifiable. All requirements should possess capability to be quantitatively measureable, which will eventually result in better software products. Targeted aims and software metrics are essential components to quantify software systems. The degree of software product maintainability is dependent upon several software metrics described in this paper.

Index Terms— Cost, Engineering, Improvement, Maintainability, Measurability, Metrics, Quantifiability

1 Introduction

SOFTWARE maintenance is one of the most important, costly and effort demanding phase of software engineering.

Recently it has been neglected and exposed to programming culture intuition. Software maintenance encompasses error removing (bug fixing), upgrading & updates according to changed requirements, changeability (ease to change) according to specific requirements by customers or external environment, enhanced functionality, modifiability and ease to use of software developed.

2 Software Maintenance

2.1 Types of Maintenance

According to Gilb (2008) the cost of software maintenance incurs major percentage of total cost of software life cycle. It covers significant portion of total cost of software development, maintenance & upgrades. Stavrinoudis (1999) classified Software maintenance into four types, which are as follows:

- **Corrective Maintenance** is bug fixing once software is in use.
- **Adaptive Maintenance** is to make changes in the software according to changes in external environment.
- **Perfective Maintenance** refers to changes requested by the customer to enhance certain features of the software.
- **Preventive Maintenance** refers to those changes that improve future maintainability and future upgrades.

- ***Engr. Syed Rizwan Ali***, *Department of Computer Sciences, Bahria University, Karachi, Pakistan, E-mail: rizwan.ali@bimcs.edu.pk*
- ***Azmat Khan***, *Department of Computer Sciences, Bahria University, Karachi, Pakistan, E-mail: azmat.khan@bimcs.edu.pk*
- ***Muhammad Shahid Khan*** , *Department of Computer Sciences, Bahria University, Karachi, Pakistan, E-mail: m.shahid@bimcs.edu.pk*
- ***Bilal Muhammad Iqbal***, *Department of Computer Sciences, Bahria University, Karachi, Pakistan, E-mail: bilal.miqbal@bimcs.edu.pk*

2.2 Quantifiablility

Quantitative principles has to be set forth to set new targets that pay off the initial investment, improving software function on long-term basis, enhancing functionality with ease to change for upgrades. Software maintenance requirements must be defined quantitatively to architect and engineer for desired results (Gilb, 2008).

3 Software Maintance Problems

The current problem with software maintenance is that it's never systematically engineered to reach specific targets but normally it is crafted in the traditional way with norms, customs and habits that seem suitable according to the situation.

Gilb (2008) suggested that software has to be engineered and maintained to a new level, specific goals and targets that pay off, but usually normal customs are fitted into developmental phase and maintenance without foreseeing the specific results. This research is associated with software maintenance cost on large scale or having critical values. Problem issues are as follow:

- Quantitative principles are not defined up front.
- Non-specified unexpected maintainability is not architected in quality requirements.
- Not testing before release of the software.
- No quantitative measure in software lifetime.
- Not built to meet unspecified requirements.

The current practice and customs in software engineering are crafting the design of software and not actually engineering it. The current practice in department of software maintenance is:

- Software development and maintenance team may list some high-level objectives but never take them as seriously to take any action further.
- The team might even decide the technology for a vague ideal.
- Software architects may carry out certain customs like decomposition of software, platform selection and software tools in order to get help.
- The team might suggest and recommend better tools and resources but fail to provide engineering approach.
- And with no specific objective and targeted goal in mind, this is called craft approach not engineering.

3.1 Break Down Steps for Software Maintenance

Gilb (2008) suggested the following areas of software maintenance where design focus is required and may also have a secondary target level for each:

- **Problem recognition** time consumed from bug occurrence to its detection and report.
- **Administrative delay** time required for bug reported till action is started on it to fix it
- **Tool collection time Delay** time for collection of tools, gather correct, complete and updated information to analyze the bug: source code, changes, database access, reports, similar reports, test cases, test outputs.
- **Problem analysis time** detection and implication in the Scale scope above.
- **Change time** is applied in parallel with Quality Control, modified only if defects are found.
- **Local test time** Automated based on distinct software (two independent changes to distinct modules and running reasonable test sets, until further notice or failure).
- **Change distribution time** all necessary changes are readied and uploaded for customer download even before local tests begin and changed only if tests fail.
- **Customer Installation time** is given option of manual or automated changes, under given circumstances.

4 SOTWARE MAINTENANCE PRINCIPLES

Quantitative principles must be defined for software maintenance in order to reach specific targets and goals. In order to achieve this Gilb (2008) suggest following important principles to gain such targets:

- Designing of software must be done with specific target (conscious design principle).
- Set of changing quality requirements must be clearly described (Many-Splendored Thing Principle).
- Specific target levels must be defined for a compulsory minimum level to avoid and target levels to reach desirable results (Multi-level principle) in terms of profit.
- Cost estimation, budget and what pays off finally is the most crucial step in quantitative (Pay off principle).
- Priority is always set for what is most important and crucial step in coding that pays off with limited resources. Targets are not based on choice or arbitrary levels of maintenance. But maintenance is always specific to targeted levels (priority dynamics principle).

This means that in software maintenance there should be specific targeted levels that pay off with regards to money investment, profits gained and time invested in maintaining and modifying the code. Quantitative targets will not only boost software capability of generating more revenues, it will reduce time in developing and modifying the code and will ease future code upgrades.

Quantifiability is more important for large scale software and critical software, it is not meant for small software that does not pay off. The scale of measure covers the entire software maintenance life cycle for initial bug detection till the customer correction level is reached and satisfactory. This entails to all processes and technologies for the corresponding design, it is not just bug detection and patching.

Since software systems are developed under high pressure of tight deadlines, the goals to achieve certain performance level, consistency, reliability and usability are performed in a conventional way without reaching specific targets.

4.1 Metrics Used to Improve Software Maintenance

The choice of the suitable internal metrics for measuring the maintainability of the specific type of product depends on the qualities of product and the programming language used during its implementation. Conventionally used metrics are Halstead's software science metrics [Halstead, 1975], cyclomatic complexity [McCabe, 1976], Tsai's data structure complexity metrics [Tsai, 1986], lines of code, lines of comments, fan–in, fan–out, etc. all can be used to measure maintainability level of a software product.

4.2 Maintenability Index

It is a software metric which measures how maintainable (easy to support and change) is the source code. The maintainability index is calculated as a factored formula consisting of Lines of Code (LOC), Cyclomatic Complexity and Halstead volume. It is used in several automated software metric tools, including the Microsoft Visual Studio 2010 development environment, which uses a shifted scale (0 to 100) derivative.

Halstead Complexity measures are software metrics introduced by Maurice Howard Halstead in 1977. These metrics are computed statically, without program execution.

First this research investigates to compute the following numbers, given the program source code:

- n1 = the number of distinct operators.
- n2 = the number of distinct operands.
- N1 = the total number of operators.
- N2 = the total number of operands.

From these numbers, five measures can be calculated:

- Program length: N = N1 + N2
- Program vocabulary: n = n1 + n2,
- Volume: V= N X log2 n
- Difficulty: D= n/2 X N2/n2
- Effort: E= D X V

Cyclomatic Complexity (or conditional complexity) is software metric (measurement). It was developed by Thomas J. McCabe, Sr. in 1976 and is used to indicate the complexity of a program. It directly measures the number of linearly independent paths through a program's source code.

Mathematically, the cyclomatic complexity of a structured program is defined with reference to a directed graph containing the basic blocks of the program, with an edge between two basic blocks if control may pass from the first to the second (the control flow graph of the program). The complexity is then defined as:

- M = E − N + 2P where
- M = cyclomatic complexity
- E = the number of edges of the graph
- N = the number of nodes of the graph
- P = the number of connected components

N

E

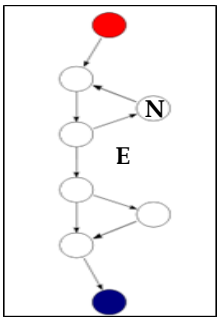

Figure 1: Shows control flow graph of the program

Then this research measures the following metrics from the source code:

- V = Halstead Volume
- G = Cyclomatic Complexity
- LOC = count of source Lines Of Code (SLOC)
- CM = percent of lines of Comment (optional)

From these measurements the MI can be calculated:

The original formula Equation is:
MI = 171 - 5.2 X ln (V) - 0.23 X (G) - 16.2 X ln (LOC)

The derivative used by SEI is calculated as follows:
MI = 171-5.2 X log2 (V)-0.23XG-16.2X log2 (LOC) +50 X sin (sqrt (2.4 X CM)) x

The derivative used by Microsoft Visual Studio (since v2008) is calculated as follows:
MI = MAX (0,(171 - 5.2 X ln(Halstead Volume) - 0.23 X (Cyclomatic Complexity) - 16.2 X ln (Lines of Code)) X 100 / 171)

4.3 Charaterize Software Metrics

According to Stavrinoudis (1999) it is mandatory to define and characterize Software metrics that characterize maintenance process for ease of change when required for specific software. Stavrinoudis (1999) explored criteria of maintainability and processes through which this criteria is interpreted, understood, reached and analyzed by software programmers. This research investigate whether software metrics and maintainability are correlated, which software metrics are proposed for maintainability process, and to determine at what point and in which cases maintainability requires software metrics.

Factor-Criteria-Metrics model states that maintainability encompasses constancy, simplicity, conciseness, self-descriptiveness and modularity. IEEE standard for quality metrics defines maintainability as correct-ability, testability and expandability. ISO– 9126 standards define maintainability as analyzability, changeability, stability and testability [ISO, 91].

4.4 Maintenance Process and its Handling

More and more programs are developed each year. Software maintenance is a daunting task, even though immense amount of resources, efforts and time is consumed due to its ever growing need in software upgrades. Factors that affect software maintenance are ease of system management, availability of qualified staff and use of standardized programming language. Negligence at any point of software maintenance can result in negative impact on software performance. Organizations that are not successfully implementing maintenance rules of design, implementation and testing may not be able to take new projects because all resources are consumed by maintenance of old programs.

Software with high level of maintainability must have modules with strong cohesion and loose coupling, comprehensive, simple, understandable, well-organized and sufficiently commented codes. It must have concurrent style, conventional language, natural expression and well–conceived terminology of their variables. Implementation of each routine must be strictly separated.

The implemented design of software must be well-documented and detailed and a thoughtful module. Furthermore, specific structures of these modules are used to limit the effort spent during the maintenance process. High reusability of one program can enhance the probability of the ease with which software modules can switch to another program. Although maintainability is indirectly related with customers, but only few customers have knowledge about the system to give any proper directives about maintainability procedure. However, such directions are normally absent and not given to the software developer. Customers may revise their requirements and opinions and requests for modifications in the system from time to time. Therefore, external requirements also change and high levels of modifications are required for successful software maintenance.

5 Approaches for Measuring Maintainability

This study has used two extensive methods for measuring maintainability that depicts external and internal views of the feature. Maintainability not only depends on product itself but also on the programmer doing maintenance, which is why it is an external quality factor.

A direct approach to measure maintainability is to describe measures required for maintenance process and then gather the views of the programmers who contribute in the process. However, this external approach is not only costly, but time consuming, as it requires conducting a survey. Alternatively, internal approach is a faster, easier and better approach and requires use of internal metrics that are analytical and more realistic for the programmers' view of the maintainability of software.

Software maintenance process is a very complicated one and the structure of the system usually degrades. Arbitrary patches applied inadequately by any inexperience staff member of the maintenance team often produce a low quality software system. Slowly this leaves the system more vulnerable and difficult to maintain. In order to keep quality of maintenance at a higher level, software metrics must be used efficiently to control the degradation of a system.

5.1 Maintainability Measures

The general patterns classify and differentiate the different classes of change processes on software:

- Adaptability
- Flexibility
- Connectability
- Extendibility
- Interchangeability
- Upgradeability
- Installability
- Portability
- Improvability

5.1.1 Adaptability

The effectiveness of a system can be changed. It is a measure of a system's ability to change. The main concern here would be the availability of resources (time, staff, tools and cost) to bring change in a specific system according to specific needs. Since change in specific system can be implemented anytime provided the resources are available. Scale: Time needed to adapt a defined System from a defined Initial State to another defined Final State using defined means.

5.1.2 Flexibility

This concerns the 'in-built' ability of the system to adapt or to be adapted by its users to suit conditions (without any fundamental system modification by system development).

Includes: {Connectability, Tailorability}. Connectability
Cost to interconnect the system toits environment. It's support in-built within the system to connect the different interfaces.

5.1.3 Extendibility

Scale the cost to add to a defined System, a defined Extension Class and defined Extension Quantity using a defined Extension Means. In other words, add such things as a new user or a new node.

Includes: {Node Addability, Connection addability, Application Addability, Subscriber Addability}.

5.1.4 Interchangeability

The cost is to modify use of system components. This is concerned with the ability to modify the system to switch from using a certain set of system components to using another set. For example, this could be a daily occurrence switching system mode from day to night use.

5.1.5 Upgradeability

The cost is to modify the system fundamentally; either to install it or change out system components the ability of the system to be modified by the system developers or system support in planned stages (as opposed to unplanned maintenance or tailoring the system).
Includes: {Installability, Portability, Improveability}.

5.1.6 Installability

The cost is used to install in defined conditions. This concerns installing the system code and also, installing it in new locations to extend the system coverage. Conditions are such as the installation being carried out by a customer or by an IT professional on-site.

5.1.7 Portability

The cost is used to move from location to location.
Scale: The cost to transport a defined System from a defined Initial Environment to a defined Target Environment using defined Means.

Type: Complex Quality Requirement. Includes: {Data Portability, Logic Portability, Command Portability, Media Portability}.

5.1.8 Improvability

The cost is used to enhance the system. The ability is used to replace system components with others, which possesses improved (function, performance, cost and/or design) attributes.
Scale: The cost to add to a defined [System] a defined [Improvement] using a defined [Means].

Conclusion

Software metrics present a simple and economical way to identify and amend probable sources for low product quality (according to the maintainability factor) as this will be perceived by the programmers. Failures can be prevented by making specific metric standards and building program measurements before starting the maintenance process in order to reduce the essential time and effort required for that phase.

This research found that the Internal and external metrics are highly correlated with the quality and time of maintainability. Low quality standards for internal and external metrics may lead to low maintenance quality, however, this is not always true and maintainability can be normal or as expected.

A maintainability criterion has to be set forth with specific internal metrics in order to produce desired results with specific targets. The main judge for software maintainability is the programmer to indicate inconsistencies in the code or design of the software.

Maintainability requirements must be defined quantitatively and economically. Design must be consciously developed to meet those targets specific for those requirements and that economically pays off. Executing that design and testing to check for required levels is important. Quality checking that specific design, then either degrading back to the original required levels or maintain required quality levels is also crucial.

ACKNOWLEDGMENT

We would like to thanks Bahria University Karachi Campus Computer Science Depratment HoD Dr Humera Farooq for her support and guidance.

REFERENCES

[1] Gilb, T. (2008) Designing Maintainability in Software Engineering: a Quantified Approach.
[2] Stavrinoudis, D. (1999) Relation between software metrics and maintainability.Proceedings of the FESMA99 International Conference, Federation of European Software Measurement Associations, Amsterdam, The Netherlands, pp. 465-476, 1999.
[3] McCabe, J, "A complexity measure", IEEE Transactions of Software Engineering, SE-2(4), 1976.
[4] Halstead, H, 'Elements of Software Science', Elsevier Publications, N-Holland, 1975. Hudli, R,
[5] Hoskins, C, Hudli, A, 'Software Metrics for Object OrientedDesigns', IEEE, 1994.
[6] Tsai, T, Lopez, A, Rodreguez, V, Volovik, D, "An Approach to MeasuringData Structure Complexity", COMPSAC86, pp 240-246, 1986.
[7] IEEE, 'Standards for a Software Quality Metrics Methodology', P-1061/D20, IEEE Press, New York, 1989.