

# Improving Software Maintainability through Risk Analysis

N.Narayanan Prasanth<sup>1</sup>, S.P.Raja<sup>2</sup>, X.Birla<sup>3</sup>, K.Navaz<sup>4</sup>, S.Arif Abdul Rahuman<sup>5</sup>  
narayana.prasanth@gmail.com, +919943182582

<sup>1,2,3&4</sup>Lecturer, Department of IT, National College of Engineering, Tirunelveli, TamilNadu, India

<sup>5</sup>Senior Grade Lecturer, Department of IT, PET Engineering College, Vallioor, TamilNadu, India

**Abstract:** The lifetime of a product is dependent on its maintainability. It is proved that a lot of cost spent towards maintainability. Researchers find it as a difficult task to measure and improve software maintainability. Here we proposed a risk based approach to improve software maintainability. Risk based approach accounts the probability of failure of a portion of code as determined by its complexity. Four products with different functionalities are taken into consideration and its risk prone areas in the code are identified. A set of six metrics with its threshold values are used to locate the highly risked areas of a code. Suitable changes are recommended to these highly risked codes to improve software maintainability.

**Index Terms:** Software Maintainability, Coupling, Software Complexity

technique to enhance the maintainability of a program. They proposed a quantitative evaluation method to measure the maintainability enhancement effect of program refactoring. They used coupling metrics to evaluate the refactoring effect. By comparing the coupling before and after the refactoring, they evaluated the degree of maintainability enhancement. Their results showed that their method was effective to quantify the refactoring effect. Suri[5] developed a simulator to compute n-step transition probabilities successively until the software reaches steady state. This process is very much depicted by Markov analysis [5]. There are many measures available in the market to measure product maintainability. However, here we proposed an risk based approach to improve the software maintainability.

## I. INTRODUCTION

The purpose of software maintainability is to improve the quality of the system. Eventhough a lot of effort can be taken during a product's development lifecycle, it is important to note that the quality of the product lies on maintainability. Maintainability of the product has a direct relationship with its lifetime. During software development life cycle, each phase is worked with a lot weigh. However, this does not guarantee that the product will reach its entire lifetime. Whereas by investing lot of effort on measuring software maintainability which inturn increases the lifetime of the product. So it is important to measure maintainability.

Maintainability can also be defined as the probability that a specified maintenance action on a specified item can be successfully performed (putting the item into a specified state) within a specified time interval by personnel of specified characteristics using specified tools and procedures [2]. Rikard Land [8] investigates how the maintainability of a piece of software changes as time passes and it is being maintained by performing measurements on industrial systems. Y. Kataoka, T. Imai .H. Andou T. Fukaya [3] discussed program refactoring as a

## II. CAUSES OF MAINTAINABILITY

There are three aspects, which causes the maintainability: Complexity, Reparability and Serviceability. Serviceability is the probability of returning the item to normal service. Reparability is the probability of repairing the actual or impending fault due to various internal or external reasons. These two factors are controllable factors and are dealed effectively in the maintainability phase itself whereas complexity is the core factor that affects the maintainability a lot. Complexity mainly lies on the requirement of execution time, storage required to perform the computation, the difficulty of performing tasks such as coding, debugging, testing, or modifying the software and so on. Therefore, it is important to control the complexity from the first phase of the software development life cycle.

## III. PROPOSED MODEL

In our model we introduced a risk based approach to find and fix the most important problems as quickly as possible. Risk can be characterized by a combination of two factors:

the severity of a potential failure event and the probability of its occurrence. Risk can be quantified by using the equation:

$$\text{Risk} = \sum p(E_i) * c(E_i),$$

Where  $i = 1, 2, \dots, n$ .  $n$  is the number of unique failure events,  $E_i$  are the possible failure events,  $p$  is probability and  $c$  is cost.

Risk-based approach focuses on analyzing the software to identify the area's most likely to experience a problem that would have the highest impact on maintainability. This looks like a daunting task, but once it is broken down into its parts, a systematic approach can be employed to make it very manageable. The severity factor  $c(E_i)$  of the risk equation depends on the nature of the application and is determined by domain analysis. Severity assessment requires expert knowledge of the environment in which the software will be used as well as a thorough understanding of the costs of various failures. Both severity and probability of failure are needed before risk-based approach can proceed. The first task of risk-based approach is to determine how likely maintainability is reduced due to complexity of the software. It has been proven that code that is more complex has a higher incidence of errors or problems that affects the product maintainability. For example, cyclomatic complexity has been demonstrated as one criterion for affecting the product maintainability. Therefore, using metrics to predict module failures might simply mean identifying and sorting them by complexity. Then using the complexity rankings in conjunction with severity assessments from domain risk analysis would identify which modules should get the most attention. However, module complexity is a univariate measure, and it could fail to detect some very risk-prone code. Here we identified a set of metrics that affects the complexity internally and externally.

Total Size(TS) – It is important to note that that the total size of a system should feature heavily in any measure of maintainability. A larger system requires, in general, a larger effort to maintain. We use a simple line of code metric, which counts all lines of source code that are not comment or blank lines.

Number of Modules(NM) - is a simple count of the different methods in a software code. A software system is decomposed into modules. The number of modules, and the ratio between the number of modules and the total lines of code is a measure of how "well" it is decomposed.

Coupling between Modules(CM) - is a count of the number of modules coupled with other modules. It is measured by counting the number of distinct non-inheritance related

hierarchies on which a module depends. Coupled modules must be bundled or modified if they are to be reused

Depth of a Module(DM) - The depth of a module lies in the way where it has been reused. Module Decomposition is the way to reuse the modules.

Duplication of Module(DOM) - We have analyzed a number of systems that were larger (in lines of code) than we had intuitively expected. Although a clear argument can be made for the size of a system being a large factor in the maintenance effort, it immediately poses the question: "But how large should this system be, given this functionality?" We have found that measuring code duplication gives a simple estimate of how much larger a system is than it needs to be.

Units Cyclomatic Complexity(UCC) - Since the unit is the smallest piece of a system that can be executed individually, it makes sense to calculate the cyclomatic complexity on that unit. The complexity follows a power law distribution, so calculating an average will give a result that may smooth out the outliers. The summation of the individual complexities of each unit does not lead to a meaningful number, so another way to aggregate the complexity needs to be found. Having defined the metrics, we need interpretation guidelines to assist in identifying those areas of code deemed to be at high risk. We had analyzed far more than 15 programs. The results of the analyses have been discussed with project managers and programmers to identify threshold values that do a good job of discriminating between "solid" code and "fragile" code. Once the individual metric thresholds were determined, they are tabulated.

The following threshold values for the individual metrics were derived from studying the distributions of the metrics collected.

- Total Size gets varied based on the programming languages used for implementation. Here we considered TS preferably be 1000 to 1500 LOC.
- Number of modules preferably less than 20 and upto 40 is acceptable.
- $CM < 5$ . A high CM indicates codes may be difficult to understand, reuse or maintain. The larger the CM, the higher the sensitivity to changes in other parts of the design and therefore maintenance is more difficult. Low coupling makes the class easier to understand, less prone to errors spawning, promotes encapsulation and improves modularity.

- $DM < 5$ . It shows that  $DM=0$  then there is no depth. If  $DM=1$  then there is root alone. Upto the value 3 is preferable and 5 is acceptable.
- $DOM=0$  is acceptable. We calculate code duplication as the percentage of all code that occurs more than once in equal code blocks of at least X lines. So if a single statement is repeated many times, but the statements before and after differ every time, we do not count it as duplicated.
- $UCC < 8$  is preferred and 10 is acceptable for each unit. Halstead measure is used to measure UCC.

IV. TEST RESULTS

A sample of 6 products was taken into consideration and the metrics evaluation as sited above was applied. This approach assists us to identify the highly prone complexed code. The table 1 shows the test results.

Table 1: High Risk Products

Product	TS	NM	CM	DM	DOM	UCC
P1	1252	51	9	12	21	11.725
P2	1448	32	5	8	15	7.21
P3	1107	12	3	3	4	5.7
P4	1960	29	7	9	16	9.39

V. DISCUSSIONS AND CONCLUSION

In our approach, we focused purely on failure probabilities. From the Table1, we can identify the products (shaded fields) with higher risks that will affect the maintainability. CM, DM and DOM are the major metrics that decides the code complexity. Since here we have enough information, about the product, we identified the risk prone areas but for the case with less information it is upto the product to determine the criticality and to make final determination towards products. Now it is easy for the developer to implement a product with less risk since the threshold values are set. It also reduces the effort that needs to be taken at maintainability phase. We are also going to use the above-tabulated values as benchmarks for further assessments.

REFERENCES

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

[2] Jarrett Rosenberg "Can We Measure Maintainability?", Sun Microsystems, 901 San Antonio Road, Palo Alto, CA 94303.

[3] Kataoka Y, Imai T, Andou T. Fukaya "A Quantitative Evaluation of Maintainability Enhancement by Refactoring" 18th IEEE International Conference on Software Maintenance, pp. 0576, 2002.

[4] Khairuddin Hashim and Elizabeth Key,"A Software Maintainability Attributes Model", Malaysian Journal of Computer Science, Vol. 9 No. 2, December 1996, pp. 92-97.

[5] Gillite Billy E., "Operations Research", Tata McGraw Hill Publishing Company Limited, 2004.

[6] Ramil J. F. and Lehman M. M., "Metrics of Software Evolution as Effort Predictors - A Case Study", In Proceedings of International Conference on Software Maintenance, IEEE, 2000.

[7] Ramil J. F. and Lehman M. M., "Defining and applying metrics in the context of continuing software evolution", In Proceedings of 7th International Software Metrics Symposium (METRICS), IEEE, 2001.

[8] Rikard Land, "Measurements of Software Maintainability" Mälardalen University, Department of Computer Engineering, Box 883, SE-721 23 Vasteras, Sweden.

[9] SEI Software Technology Review, *Halstead Complexity Measures*, URL: <http://www.sei.cmu.edu/>, 2002

[10] Suri P.K and Bharat Bushan., "Simulator for Software Maintainability", IJCSNS International Journal of Computer Science and Network Security, VOL.7 No.11, November 2007.