SOFTWARE METRICS IN MAINTENANCE PHASE OF OBJECT ORIENTED SOFTWARE DEVELOPMENT LIFE CYCLE

SHWETA SHARMA*

Dr S. SRINIVASAN**

Abstract

Software maintenance is a task that every development team has to face when software is delivered to the customer's site, installed and is in operational state. The time spent and the effort required keeping the software operational after release is very significant and consumes about 50-70% of the cost of the entire lifecycle. Hence it becomes a challenging task for an organization to predict the maintainability of their software system in order to manage their software resources efficiently. This helps them to in reducing the maintenance effort as well as the total cost and time spent on a software project. This main aim of this paper is to focus on some existing maintenance metrics and their limitations. Some new metrics have also been suggested to effectively measure the maintenance efforts depending upon the software project needs.

Keywords: Fan-In, fan-out, Complexity Density, NAC, NDC, Inheritance, Defect Fix

* Phd. Research Scholar, Mewar University, Rajsthan, India

** Head of Dept. PDM College, Haryana, India

Introduction

Software maintenance is a very vast activity that includes error rectifications, enrichment of capabilities, deletion of those capabilities which are not in trend and optimization. The whole purpose is to preserve the value of software over time. Maintenance may be extended up to 20 years, whereas development process may be 1-2 years long.

The Object-Oriented (OO) paradigm has become increasingly popular in recent years. On the other side, the insertion of OO technology in the software industry has created new challenges for companies that use product metrics as a tool for monitoring, controlling, and improving the way they develop and maintain software [1]. Researchers agree that, although maintenance may turn out to be easier for OO systems, it is almost next to impossible that the maintenance burden will completely vanish [2].

The maintenance is a time consuming and expensive activity because most of the time of a maintainer is spent to understand the structure of the program. The greater the complexity of the program, more efforts will be required to maintain it. The metrics help the engineer to recognize parts of the software that might need modifications and re-implementation. The decision of changes to be made should not rely only on the metric values [3]. The metrics are guidelines and not rules and they should be used to support the desired motivations.

In object-oriented programming, the main focus is to support the qualities such as software reusability which can be encouraged by use of abstractions, dividing responsibilities and detecting and correcting anomalies in the designs. The conventional metrics such as Line of Code (LOC) and Cyclomatic Complexity have become standard metrics for measuring complexities of procedural languages.

But in The metrics for object-oriented systems are different due to the different approach in program paradigm and in object-oriented language itself. An object-oriented program paradigm uses localization, encapsulation, information hiding, inheritance, object abstraction and polymorphism, and has different program structure than in procedural languages [4].

The design evaluation step is an integral part of achieving a good quality and an eminent design. The metrics which is used to measure the quality software should have the feature that it can help in improving the total quality of the end product, which means that problems related to the quality of the software could be resolved in the early stages of the development process. It is an

obvious fact that the earlier the problems can be resolved the less it costs to the project in terms of time-to-market, quality and maintenance.

Existing Maintenance metrics for Object Oriented Systems

This section of the paper provides a study of some metrics which are typically used in maintenance phase of development lifecycle. The following are the important maintenance metrics available in existing literature [7]:

- Complexity Density Metrics for maintenance productivity.
- Fan-in and Fan-out metrics for estimating maintenance complexity
- Commentedness and LOC metrics for maintenance and productivity estimation.
- Halstead metrics for man-months calculation.
- Complexity Density Metrics for maintenance productivity.
- Metrics for maintainability of class inheritance hierarchies.

Complexity Density Metrics for maintenance productivity: Complexity density is inversely proportional to the maintenance productivity. It can be defined by the following equation:

$$\text{Maintenance Productivity} = \frac{\text{Total number of Lines Added}}{\text{Time spent in coding and testing upgrades in hours}}$$

Hence complexity density metric can be used as a measurement of difficulty level in the maintenance of a program.

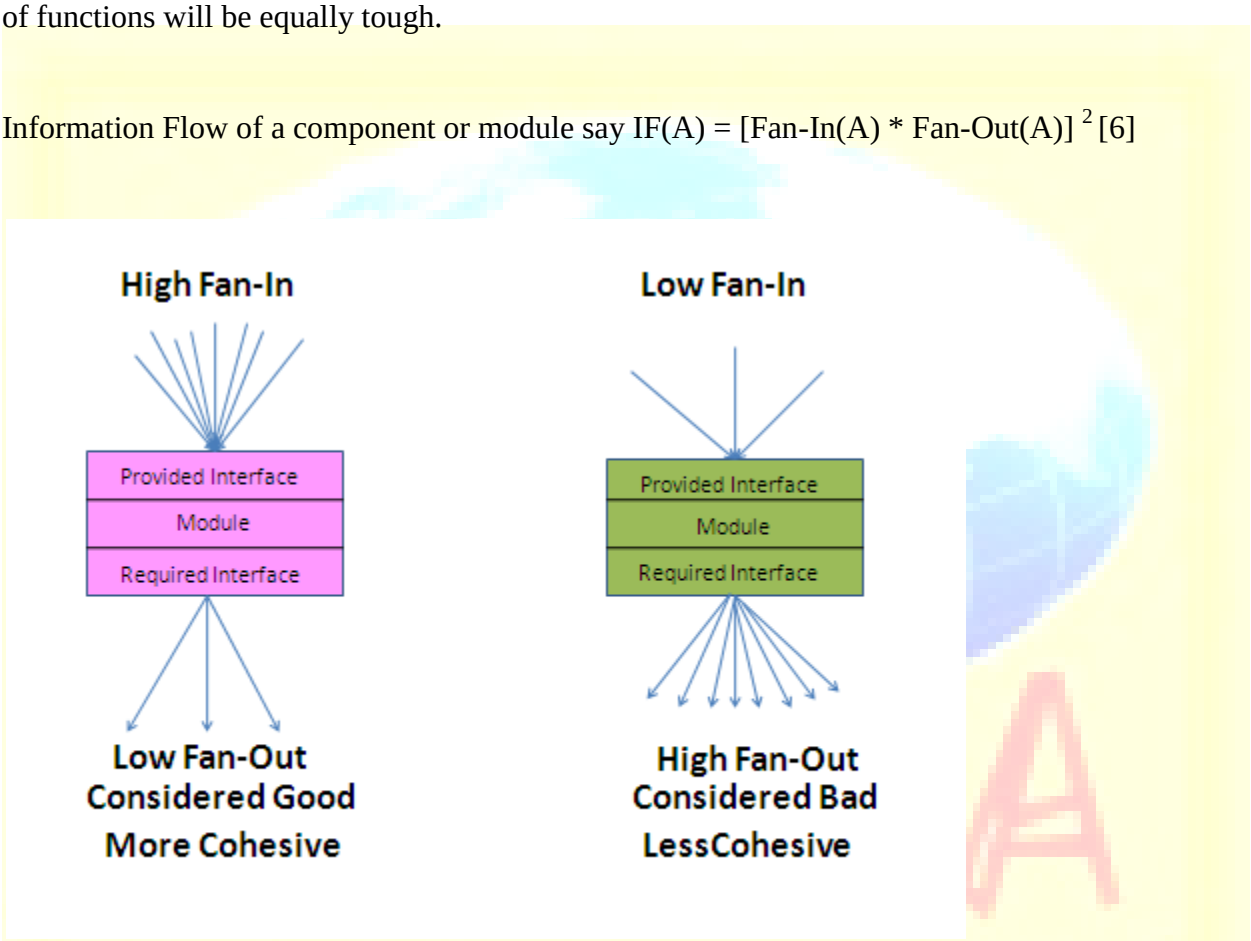
Fan-in and Fan-out metrics for estimating maintenance complexity [7]: The Fan-in and Fan-out metrics are used in estimating the complexity of maintaining the software. Fan-out is a count of the total number of functions a function calls. If we modify a function then it may cause abrupt changes in the functions called by the modified function. A maintainer of this module has to understand many other functions causing maintenance a very harder and time consuming job. Therefore, it will be more expensive to maintain functions with a large Fan-out.

Fan-in is the count of the number of functions that call a function. A function which has high Fan-in means that this particular function is used by many other functions. In other words we can also understand that the function is implementing a number of functionalities. If we alter the

specifications of a function with large Fan-in then all the other functions that use this have to be accordingly modified.

Hence a function with high Fan-in and low Fan-out will be considered good as it promotes reusability and extensive functionalities on the other hand the function with low Fan-in and high Fan-out will be considered as an example of bad design. However the maintenance of both type of functions will be equally tough.

Information Flow of a component or module say IF(A) = [Fan-In(A) * Fan-Out(A)] ^{2} [6]



Commentedness and LOC metrics for maintenance and productivity estimation: Commentedness metrics is required to enhance the readability of a program. This metrics proves to be very significant in the maintenance process and also if someone wants to perform re-engineering on the existing software product.

lines of code (LOC) is a software metric used to measure the size of a computer program by counting the number of lines in the text of the program's source code which is in Kilo lines of code (KLOC). This metric is basically used to predict the amount of effort that will be required

to develop a program, also to estimate programming productivity or maintainability once the software is produced.

Halstead metrics for man-months calculation [7]: Halstead metrics is based on the fact that the complexity of a program is related to the number of operators and operands in the program. Operators are syntactic elements such as +, -, <, > and operands are quantities that receive the operators namely variables and constants. Halstead metrics calculates the Program Volume, Difficulty and Programming Effort. The Effort measured in the Halstead metrics is a measure of the effort invested in man-months programming the module.

Complexity Density Metrics for maintenance productivity: Complexity Density is considered as inversely proportional to the maintenance productivity. As the complexity density increases the maintenance productivity decreases. Maintenance productivity can be calculated as the total number of lines added divided by the time in hours spent in coding and testing upgrades to the software. Hence, Complexity Density can also be used as a measure of the difficulty of maintaining a program.

Metrics for Maintainability of class inheritance hierarchies: Inheritance is very beneficial when we want to reuse an already existing class. However it also has some side-effects as it introduces some coupling among the classes. If some alterations are made in the ancestor class then those changes can potentially affect all the descendent classes. Therefore the inheritance maintenance metrics NAC and NDC given by Li are very significant in measuring potential change propagation through the inheritance hierarchy [5].

NAC (Number of Ancestor Classes): NAC measures the total number of ancestor classes from which a class inherits in the class inheritance hierarchy.

NDC (Number of Descendent Classes): The NDC is the count of total number of descendent (subclasses) of a class.

Proposed Metrics for Maintenance Measurement: The software maintenance is a process of modifying the software system or components to detect and correct defects to improve

performance according to the changing environment. Hence there is a metrics required to measure the impact of how defects are fixed to initiate corrective actions.

$$\text{Defect Fixes (\%)} = \frac{\text{Cumulative number of defect fixes}}{\text{Cumulative Total Fixes}} * 100$$

Here total fixes imply all kinds of fixes that may be defect fixes as well as changes made to enhance productivity, changes made to reduce testing efforts and so on. The percentage will show the frequency of the occurrence of defects in overall changes made to the software system.

Effort Spent on Reviews and Testing: This metric will determine the cost of identification of a defect at different stages, and help in defining review/testing strategy. During the analysis and design phases, the data is collected at the end of the phase. From the coding phase onwards, the data is generally collected at the end of each review and test cycle.

% Review Effort= (effort spent on review of a module) / (Total effort required to complete that module) * 100

% Testing Effort= (effort spent on testing of a module) / (Total effort required to complete that module)* 100

This data can be used along with the number of defects found to monitor the effort spent per defect. The effort spent per defect can be defined as follows:

$$\frac{\text{total effort spent in review/test}}{\text{total number of defects found}}$$

The data can also be used to monitor the differences between the planned effort and the actual effort spent in reviews and testing.

Effort Saved by Reuse: This metrics is related to the reusability property of object oriented programs as in inheritance, where we can reuse the properties of parent class in child classes. But if we calculate the efforts spent in making changes to the parent classes(causing changes in their respective child classes) then we should also calculate the efforts saved in reusing the classes and objects where required.

How to calculate: The effort saved by the reuse of the code, design or system architecture may be estimated as the difference between the effort that would have been spent if there was no such reuse and the effort that was actually spent.

This may be used to evaluate the degree of reuse and monitor the estimated effort saved against the effort expected to be saved.

Conclusion

In this paper we tried to reinforce the fact that the software metrics are important not only for the design, coding, reliability and testing phases of a software development life cycle but play a very significant role in the maintenance phase too. We presented a thorough study of the existing metrics available for the maintenance activity. These metrics show that how much time, effort and money is spent in incorporating changes to the software system. We also suggested some metrics for measuring the efforts involved in maintenance activity. Also we've seen from the existing literature that there are many metrics for efforts calculation but there is hardly any work done in calculating the efforts saved in reusing the components in object-oriented environment. Hence we suggest that there should be some metrics for effort saving calculations in using the features of object-oriented languages for example inheritance, reusability.

References

[1]Victor R. Basili, Fellow, IEEE, Lionel C. Briand, Walcélio L. Melo, Member IEEE Computer Society:A Validation of Object-Oriented Design Metrics as Quality Indicators, IEEE Transactions on Software Engineering, Volume 22, Number 10, pages 751 - 761, October 1996.

[2] Rajendra K. Bandi, Vijay K. Vaishnavi, Fellow, IEEE, Daniel E. Turk, Member, IEEE: Predicting Maintenance Performance Using Object-Oriented Design Complexity Metrics, IEEE Transactions on Software Engineering, Volume 29, Number 1, pages 77 - 87, January 2003.

[3] Tarja Systä, Ping Yu: Using OO Metrics and Rigi to Evaluate Java software, University of Tampere, Department of Computer Science, Series of Publications A A-1999-9, 24 pages, July 1999.

[4] Shuqin Li-Kokko: Code and Design Metrics for Object-Oriented Systems, Helsinki University of Technology, 9 pages, 2000.

[5] Harrison R, Counsell SJ, Nithi RV. An evaluation of the MOOD set of object-oriented software metrics. IEEE Trans. on Software Engineering 1998; 24(6):491–496.

[6] K. K. Aggarwal & Yogesh Singh, "Software Engineering", 2nd Ed., New Age International, 2005.

[7] "MetricAdvisor", www.cdac.org.i, [online] Feb 2002, cdac.in/html/**pdf**/meticadv.**pdf** (Accessed: 25 September 2013)