Malaysian Journal of Computer Science, Vol. 9 No. 2, December 1996, pp. 92-97

A SOFTWARE MAINTAINABILITY ATTRIBUTES MODEL

Khairuddin Hashim and Elizabeth Key
Faculty of Computer Science and Information Technology
University of Malaya
50603 Kuala Lumpur
email: kh@fsktm.um.edu.my

ABSTRACT

Examines the factors that affect the maintainability of a software. The proposed model can be used to highlight the need to improve the quality of the product so that proper and efficient maintenance is feasible without much difficulty. It can also be used to develop a measurement for maintainability which can be used to measure the level of maintenance readiness before the completion and delivery of a software product.

Keywords: ***Maintainability model attributes; Software maintainability attributes***

1.0 INTRODUCTION

Maintenance has often taken a back seat where software development is concerned [1]. However, once a software is delivered it gets maintained for the rest of its useful life. With the continuing increase in software production more and more resources are spent on maintenance. The maintenance of existing software can account for 70 percent of all effort expended by a software organisation [2]. It is estimated that many companies will spend close to 80 percent of their software budget on maintenance if nothing is done to improve the current approach. As such, a closer look at improving our maintenance approach is needed.

Software quality can be defined as the totality of features and characteristics of a product that could satisfy a given set of requirements. Some of the factors of software quality include reliability, reusability, maintainability and portability. These quality factors are then broken down into lower level quality criteria which serve as attributes of software.

Contrary to perceived belief that much has been discussed about maintainbility attributes, there is not much available in terms of a refined and organized attributes model. This paper discusses a proposed model that describes the basic attributes that software should have to make it more maintainable. It is time to take a closer look into maintainability attributes again so that effort could be directed more efficiently during the phases before maintenance.

2.0 THE MAINTENANCE PROCESS

Basically, maintainability involves corrective, adaptive and perfective maintenance. It is an important quality as components are dynamic and require modifications in their lifetime. However, maintainability can also be viewed as two separate qualities:

- reparability
- evolvability

2.1 Reparability

Reparability involves corrective maintenance. A software system is repairable or corrective if it allows the removal of residual errors present in the product when it is delivered as well as the errors introduced into the software during its maintenance.

Reparability is affected by the number of parts in a product. A software product comprising well-designed modules is much easier to analyse and repair than a monolithic one. However merely increasing the number of modules does not make a more repairable product. The right module structure with the right module interfaces has to be chosen to reduce the need for module interconnections. The right modularisation promotes reparability by allowing errors to be confined to few modules, thus making it easier to locate and remove them. Reparability can be improved through the use of proper tools, for instance high-level language results in higher reparability in a software product. A product reparability affects its reliability. However, the need for reparability decreases as reliability increases.

2.2 Evolvability

Due to the change in demands on performance over time, software products are modified to provide new functions or to change existing functions. A software product can evolve gracefully if it is designed with care in the first place and each step of modification which is to be done on it is thought out carefully. Evolvability of software is assuming importance due to the increase in the cost of software and the complexity of application. Evolvability can be achieved by modularisation but successive changes tend to reduce the modularity of the original system especially so if the modifications are applied without careful study of the original design and without precise description of changes in both design and the requirements specification. Hence,

the initial design of the product, as well as any succeeding changes must be done with evolvability in mind.

Evolvability involves two type of maintenance. Adaptive maintenance has to do with adjusting the application to changes in the environment, that is, a new release of the hardware or a new database system. In adaptive maintenance the need for software changes cannot be attributed to a feature in the software itself, such as the presence of residual errors or the inability to provide some functionality required by the user. Rather, the software must change because the environment in which it is embedded changes.

Perfective maintenance involves changing the software to improve some of its qualities. Here, changes are due to the need to modify the functions offered by the application, add new functions, improve the performance of the application, make it easier to use, etc. The requests to perform perfective maintenance may come directly from the software engineer to upgrade the status of the product on the market or they may come from the customer to meet some new requirements.

3.0 PROBLEMS

Most problems that are associated with software maintenance can be traced to deficiencies in the way the software was developed. Braind et. al identify the initial quality of software and its documentation as one of the important factors affecting software maintenance quality and productivity. Some of the problems identified are:

- no traceability
- no documentation
- badly designed and implemented
- unsuitable programming language, development tools and techniques

4.0 THE MODEL

An attempt is made here to define maintainability as a manifestation of other lower factors. The model describes software related factors affecting maintainability of software components.

One software quality matrix proposed by McCall [2] and his colleagues looked at the three main aspects of a software product namely product operation, product revision and product transition. Maintainability is classified under product revision with metrics such as concision, consistency, instrumentation, modularity, self documentation and simplicity. However, attributes such as complexity, standardisation, programming language, traceability and others are not considered.

An adequate and complete maintainability attributes identification is necessary. A proposed maintainability attributes model is depicted in Fig. 1. It encompasses modularity, readability, programming language, standardisation, level of validation and testing, complexity and traceability. The model can be used to highlight the need to improve the quality of the product so that proper and efficient maintenance can be performed without much difficulty.

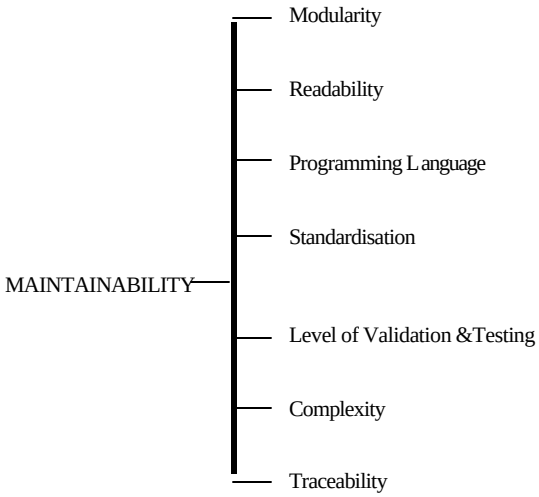

Fig. 1: A Software Maintainability Attributes Model

The maintainability attributes model described here is part of a reusability attributes model developed to measure the reusability of software components[4]. The model can also be used to develop a measurement for maintainability which can be used to measure the level of maintenance readiness before the completion and delivery of a software product. A high level of maintenance readiness would mean that a software is equipped with the necessary program features and documentation to allow it to be maintained efficiently.

Factors affecting maintainability are:

- modularisation
- readability
- programming language
- standardisation
- level of validation and testing
- complexity
- traceability

Factors relating to maintainability are discussed below.

4.1 Modularisation

Modularisation allows one to decompose a system into functional units, to impose hierarchical ordering on functional usage, to implement data abstractions, and to develop independently useful subsystems. In addition, modularisation can be used to isolate machine dependencies, to improve the performance of a software product, or to ease debugging, testing, integration, tuning and modifications of the system. A complex system may be divided into simpler pieces called modules. A system that is composed of modules is called modular. High modularity allows the principle of separation of concerns to be applied in two phases:

(i) when dealing with details of each module in isolation and ignoring details of other modules and
(ii) when dealing with the overall characteristics of all modules and their relationships in order to integrate them in a coherent system.

The goals of modularity are as follows [5]:

- decompose a complex system
- compose it from existing modules
- understanding the system in pieces
- continuity of modules
- protection of modules

Decomposability of a system is based on dividing the original problem top down into sub-problems whose solution may then be pursued separately and then applying the decomposition to each sub-problem recursively. This reflects the well-known Latin motto divide et impera (divide and conquer). The rational behind decomposition involves the method in reducing the apparent complex system into a set of simpler or less complex subsystems which they themselves would be decomposed into atomic components.

The composability of a system on the other hand is based on starting bottom up from elementary components and proceeding to the finished system. In software production, one is able to assemble new applications by taking modules from a library and combining them to form the required product, possibly in an environment quite different from the one in which the components were developed. This is in direct concern with reusability whereby a component library must be referred to before any activity is done from scratch. The three most widely recognised mechanisms for compositions are the pipe mechanism in Unix, inheritance in Smalltalk and the sharing of interfaces between modules.

In composability, one would have to integrate all the building blocks to form a system. The size of such blocks can vary from a small subroutine to a program, as in the Unix pipe. Granularity refers to the size of these building blocks. If the reusable component is bigger, there is a better chance that the interface it provides will be more abstract and hence easier to reuse. Also the payoffs are larger in the case of bigger building blocks.

Ease of composability refers to the effort required to compose reusable blocks. Understandability of block would be an factor in composability. It could vary from the understanding of the source code to just knowing the input and output of the program. If the effort is less, there is a better chance that reusers will use existing entities. Hence, the popularity of the pipes in Unix. Such reusable modules should be designed with reusability in mind. By using reusable components, one may be able to speed up both the initial system construction and its fine-tuning.

Comprehending each component is a prior step for modifying a system. Each component if understood separately aids in modifying a system. If the entire system can only be understood in its entirety then modifications are unlikely. Otherwise, modification would result in an unreliable system. This criterion is important as proper modularity would also help to confine the search for the source of malfunction to single components in the case of maintenance.

If a component exhibits the continuity characteristic then an occurrence of a small change in a specification would not result in ripple effects in the system. A small change should only affect individual modules in the architecture of the system rather than the architecture itself or the relationship among the modules [5].

Component protection refers to the ability of architectures of components to withstand any abnormal condition occurring at a run-time. The effects of abnormalities should be confined to that particular module or be propagated to a few minimal neighbouring modules. The abnormalities here refers to run-time errors, resulting from erroneous input or lack of needed resources. Hence, the architecture of component ought to be robust to protect itself from any form of abnormal conditions.

To achieve the goals of modularity (modular composability, modular decomposability, modular understanding, modular continuity, modular protection) the following principles must be observed. Five principles are examined[6] as below, the first principle being related to notation and the rest four principles addresses the issue of communication between modules.

- linguistic modular units
- few interfaces
- small interfaces (weak coupling)
- explicit interfaces
- information hiding

Modules must have high cohesion if all its elements are related strongly. Elements of the same module are grouped together in the same module for a logical reason in order to achieve the goal that is the function of the module. Coupling characterises a module's relationship to other modules. It measures the independence of two modules. High coupling exists between two modules when they depend on each other heavily. Ideally, modules in a system should exhibit low coupling as this would ease analysing, understanding, modifying, testing or reusing them separately.

Hence modular systems are desirable as they are [7]:

- easier to understand and explain because they can be approached a piece at a same time and because each piece have well-defined inter-relationships.
- easier to understand and explain and hence easier to document.
- easier to program as independent groups can work on separate modules with little communication.
- easier to test because they can be tested separately and then integrated and tested together, one module at a time.
- easier to maintain because changes can be made without disturbing the rest of the system.

4.2 Readability

Readability refers to the degree to which a reader can quickly and easily understand source code. This is important as every program is read again and again during its creation, testing, debugging and maintenance. Readability is affected by the quality and quantity of program documentation. If a program is supported by clear, complete yet concise documentation, the task of understanding the program can be relatively straightforward. Consequently, program maintenance costs tend to be less for well-documented systems than systems supplied with poor or incomplete documentation.

Readability is enhanced by its

- internal documentation
- external documentation

Self-documentation refers to the source code which provides meaningful description to increase comprehensibility and legibility of program enabling user to understand the software functionality, operational environment and all other required attributes. A well-documented program should have its interface and design specifications of components too.

Comments refer to supplementary text, table, graphs interspersed with source code. Comments could include goals of program, plans which outline the processing steps for achieving program goals.

External documentation includes:

- program unit notebook
- implementation notes

This external documentation comprises material about source code external to the source code file. It helps readers to understand code, and it provides an implementation tracking mechanism, a mechanism for tracing the fulfilment of requirements and helpful summaries of the testing, debugging and change history of code segments. Program unit notebook is a diary of the life of a program unit, written by the unit's programmer. It contains

- a synopsis of the requirements fulfilled by the program unit.
- a review of the program unit's design
- discussion of difficult, unusual or tricky aspects of implementation.
- implementation milestones and completion dates for the program unit.
- the program unit test plan
- the modification history for the program unit.

Implementation notes helps to improve readability as it discusses difficult or subtle algorithms and data structures. It includes graphs, drawing, charts and other representations difficult to reproduce in source code library. It also comprises photocopies of portions of books or articles relevant to the design or implementation. All these documentation enhance readability of program. The more readable a module the faster and more accurately a reuser can obtain information about it. Here readability can be gauged by the average number of right answers to a series of questions about the program in a given length of time. Comments could indirectly rescue a not so modular program and make it as readable as modular program by increasing its readability.

4.3 Programming Language

Is the code written in the desired language? A code written in a language where the programmers of an organization are not familiar with results in the difficulty in maintaining the reused code. The language used will also affect the readability of the programmer. A standard language for each domain application is highly recommended as an essential strategy in software development.

Programs written in a high-level programming language are usually easier to understand than that in a low-level language. The understandability could enhance other qualities like evolvability and verifiability.

4.4 Standardisation

A set of programming standards should be available to act as a guide in code writing to avoid idiosyncrasy among programmers. Project coding standards specifying internal documentation guide where such guidelines list rules for improving code readability should include naming and formatting conventions, practices and the use of types and control structures, practices and templates for writing comments and targets for code size and comment density.

Program conventions, programming language, macros, program formats and documents are standardised for better understandability. The system should be consistent in the use of notations, terminologies and symbols. Code should be indented in a homogeneous manner.

The programming standards, guidelines and practices used in writing a program clearly contributes to its readability and hence understandability, therefore, directly affecting modifiability and maintenance.

4.5 Level of Validation and Testing

Generally, more time and effort spent on design validation and program testing, results in fewer errors in program and consequently decreases maintenance cost resulting from error correction. Maintenance costs due to error correction are governed by the type of error to be repaired. Coding errors are relatively cheap to correct, whereas design errors are much more expensive as they may involve the rewriting of one or more program units. Errors in the requirements specification are normally the most expensive to correct because of the drastic redesign which is usually involved.

4.6 Complexity

The complexity of a software affects its maintainability. It is supposed to reflect to a certain extent, the difficulty in comprehending or maintaining codes. It can also be used as guideline for estimating the number of test cases and so forth. Measuring the complexity of a module involves measuring the control flow of the module, data flow and even data structures used. A module's complexity control flow [8] is a general concept relating to the order in which the various instructions of a program are executed and as such any measure indicative of the number and nature of statements that alter the sequential flow can be used as a measure of complexity.

4.7 Traceability

Traceability refers to the ability to trace a design representation or actual program components back to requirements. It is manifested in the availability of information linking requirements with corresponding design components and to their corresponding code fragments, providing reverse mapping information.

Traceability is best implemented by implementing cross referencing features such as requirements labelling and function indexing across all software development documents.

5.0 CONCLUSION

A software maintainability attributes model is proposed taking into consideration important factors such as traceability which affects the maintainability of software. As the amount of effort and resources expended on software maintenance grows, new approaches to software development have to be explored, highlighting the importance of developing software in a controlled and documented manner. It is hoped that through highlighting the important attributes described in this paper, evaluation of proper completion of a software development project with respect to its maintainability level can be checked.

REFERENCES

[1] Schach, *Classical and Object-Oriented Software Engineering*, 3rd. Ed., IRWIN, 1996.

[2] R. S. Pressman, *Software Engineering: A Practitioner's Approach*, McGraw-Hill Book Co., 1992.

[3] L. C. Briand, V. R. Basili, Y. M. Kim, and D. R. Squier, “A Change Analysis Process to Characterise Software Maintenance Projects”, *Proceedings of the International Conference on Software Maintenance*, Victoria, Canada, 1994.

[4] E. Key, *"A Tool For Software Reuse"*, M. Comp. Sc.. Thesis, University of Malaya, 1994.

[5] B. Meyers, *Object-oriented Software Construction*, Prentice Hall, New York, 1988.

[6] I. Sommerville, *Software Engineering*, Addison-Wesley, 1992.

[7] C. Ghezzi, M. Jazayeri and D. Mandrioli, *Fundamentals of Software Engineering*, Prentice Hall, 1991.

[8] P. K. Sinha, S. J. Prakash, K. B. Lakshmanan, " A new look at the control flow complexity of computer programs", in Barnes D. & Brown P., *Software Engineering 86*, Short Run Press Ltd., 1986 pp. 88-102.

BIOGRAPHY

Khairuddin Hashim is an associate professor at the Faculty of Computer Science and Information Technology, University of Malaya where he teaches Software Engineering, Systems Analysis and Design and Programming Principles and Techniques. He is a doctoral graduate in Computing Science from the University of Bath, UK. His research interests include Programming Languages, Software Reusability, Requirements Engineering, Software Metrics, Software Maintenance and Process Modelling.

Elizabeth Key graduated with a Master in Computer Science from University of Malaya, Kuala Lumpur, majoring in software reusability.
.