Malaysian Journal of Computer Science, Vol. 9 No. 2, December 1996, pp. 36-42

A TASK-ORIENTED SOFTWARE MAINTENANCE MODEL

Md. Khaled Khan
Centre for Computing and Mathematics
Southern Cross University
P.O. Box 157 Lismore
NSW 2480 Australia

Mohammad Abdur Rashid
Computer Science Program
University of Brunei Darussalam
BSB 2028; Brunei Darussalam
email: rashid@ubd.edu.bn

Bruce W. N. Lo
Centre for Computing and Mathematics
Southern Cross University
P.O. Box 157 Lismore
NSW 2480 Australia
email: blo@scu.edu.au

ABSTRACT

Software maintenance is the only phase in the system life cycle that does not have any firm theoretical foundation for its practice due to the lack of precise definitions and a maintenance model defining the tasks involved. This paper presents a generic software maintenance model based on various tasks. It also focuses on the relationship between software development and the software maintenance process in terms of software life cycle model. This model of software maintenance illustrates a comprehensive approach attempting to integrate the software maintenance process with that of software development in a single software life cycle framework.

Keywords: ***Software maintenance, development process, life cycle model.***

1.0 INTRODUCTION

Software maintenance is the "modification of a software product after delivery to correct faults, to improve performance, or to adapt the product to a changed environment" (ANSI/IEEE standard 729-1983). Following Grady Booch [1], "it is *maintenance* when we correct errors, it is *evolution* when we respond to changing requirements, it is *preservation* when we continue to use extraordinary means to keep an ancient and decaying piece of software in operation". Munro [2] and Swason [3] classify software maintenance activities into four major types: *(i) Corrective maintenance, (ii) Adaptive maintenance, (iii) Preventive maintenance, and (iv) Perfective maintenance.*

It was generally estimated that as much as 50 to 70 percent of the annual budget of DPDs (Data Processing Departments) is being spent on software maintenance [3]. Unfortunately, there is no sound baseline in the software engineering field on how to systematically maintain existing software systems. More than one decade ago, it was predicted that software maintenance would become a problem in DPDs, and the ratio of efforts devoted to software maintenance would increase indefinitely in the future [4]. This is exactly the scenario we are experiencing today.

The natural question is to ask why software maintenance is so difficult to manage. The answer lies in various related factors. It includes a lack of management understanding and support; incompatible design documents which makes the understanding of the program code difficult; lack of proper maintenance methodology and defined procedures. The last one is of particular concern. The procedures followed, if any existing software maintenance by practitioners are generally very poor. Only very few scattered guidelines are available for information systems managers to enable them to tailor process models to maintenance projects. It is well understood that a better formalism of the maintenance process would significantly improve the efficiency of the software maintenance activity [5]. A more defined formalism, describing a precise relationship between the maintenance and the development process, is required to enable a clear understanding of the various tasks involved in software maintenance. In this direction, modeling the maintenance process would be the initial requirement. Some works have already been done on this, particularly by the Software Engineering Institute [6]. But within the software engineering context, it has not yet been well established how software maintenance would relate to the software development process. To achieve effective software maintenance, a precise framework in software engineering should be proposed. The framework should clearly define each phase of software maintenance. It is important to formalise maintenance requirements at the beginning and use them as a basis of a realistic maintenance plan for implementation [7].

Software development and software maintenance, the two processes in software engineering, constitute a cycle for the entire life span of a software system. This concept of a software life cycle is a model used to describe and explain software development and the maintenance process in an engineering fashion. The top-down waterfall model [8] has been widely accepted by the software community while the spiral model [9] has also received considerable attention in recent years. The latter emphasises the risk analysis aspect of a software project, while the former views maintenance as a single phase in the post development chain. In contrast, this study attempts to make a case that while software maintenance can be regarded as a separate phase, it can also be practised as a legitimate engineering process in its own right at the same level as the software development cycle. Thus, in the following section, we define a model that treats software maintenance as a collection of well-defined procedures aims to maintain an existing software product. We also propose an extended software life cycle model in which these procedures can be linked with development in a different way from that focused in previous models.

2.0 MAINTENANCE MODEL

The maintenance model shown in Fig. 1 defines how various tasks in a chain are to be performed in the context of the maintenance objectives and constraints of the maintenance project. However, in his review of the literature, Deraman [17] identified three major common features of the overall software maintenance model. These are *understanding the software, modifying the software, and revalidating the software*. Our model encompasses these features. It further shows that the software maintenance process consists of a group of systematic and well-defined tasks which should be performed one after the other in sequence in order to achieve maintenance goals. The software maintenance process takes the existing source code, its documents, and the modification requests as its input and produces a "modified" system as output to meet the user's "new" requests. Hereafter, we shall use the term "modification" to denote any addition, deletion, alternation or any other maintenance actions. The phases in this model are explained in the following sections.

2.1 Modification Requirements Analysis

In this phase, requests from users already using the system are received. These can be anything like adding new functions, improving the performance of existing functions, migrating the system to other operational platforms due to a change of hardware/operating system, modifying the existing function due to a change of business rules in the organisation, or correcting errors. These requests can be analysed in detail for the existing system in such a way that the entire modification requirements can be well understood both by the user and the staff who are involved in the maintenance project. Changes may also be required to the system itself due to the need for a new operational environment.

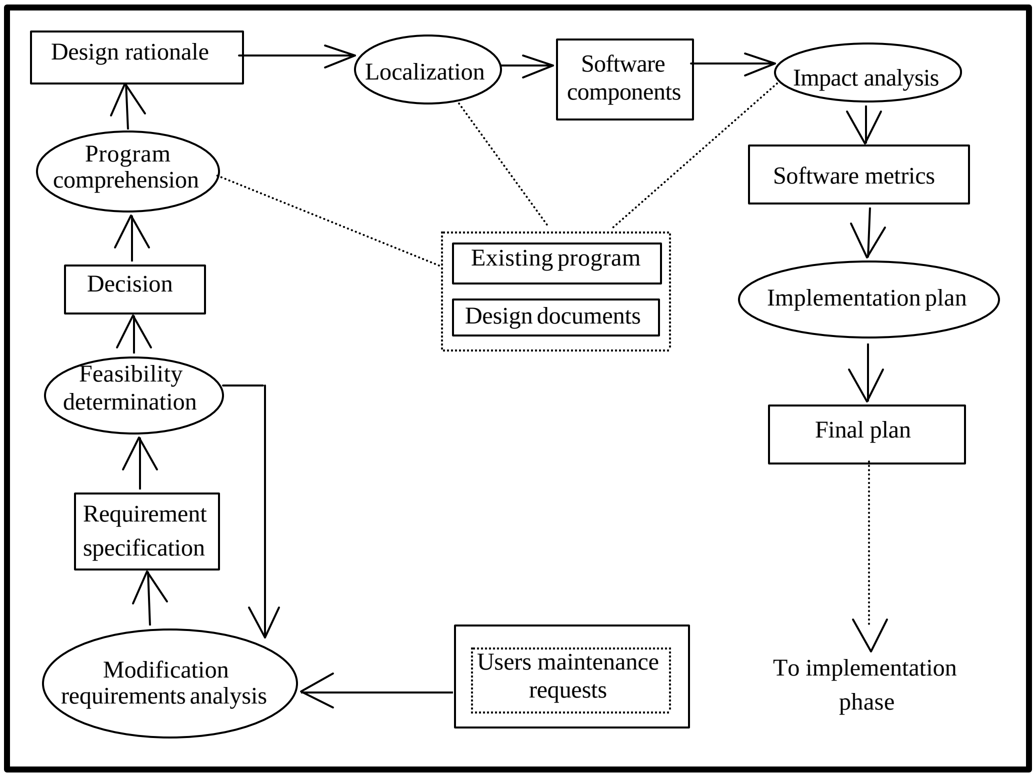

Fig. 1: A task-oriented software maintenance model

2.2 Feasibility Determination

This is a constraint mechanism in the maintenance process. In this phase the user requirements are examined in terms of technical and economical feasibility. Two important issues are analysed: Is it really important and relevant to introduce this modification to the existing system? And if it is, how much effort will be required to implement the change? The answers to these two fundamental management issues will determine whether the proposed modification will be introduced or not. Thus cost to benefit ratio is considered as a determinant of the modification process. The user requirements are refined and filtered at this stage. It actually includes estimating cost, staff, and tool availability, and quality requirements. A feedback loop exists between this phase and the previous one. The outcome of this phase is a decision whether to implement all or part of the user's maintenance requests.

2.3 Program Comprehension or (Code X-ray)

In this phase, the complete design rationale of the system will need to be well understood by the maintainer. Source code level understanding is necessary to carry out the maintenance work. In a survey on software maintenance managers [11] it was revealed that incompatible design documents for candidate software are the major cause of maintenance problems for 48% of the participating managers. If the design documents cannot be relied on for some reasons, then the source code alone needs to be studied for this purpose. In most cases, there is no historical trail of how the product was actually developed and why design and implementation were done as they were [12]. Understanding the source code involves code reading, program execution, and the use of existing design documents. In such a situation, the assistance of automatic or semi-automatic program understanding tools like reverse engineering or re-engineering can be sought. These tools can be a valuable aid for understanding programs. Automatic design recovery, or code comprehension, is sometimes referred to as 'code x-ray'.

2.4 Localization

In this phase, the precise location in the system where the proposed modification to be made is identified. It is important to trace out the exact modification area in the source code. This phase is also referred as 'spotting'. The technique of program slicing or design recovery can be applied to the candidate code for this purpose. It is important to emphasis that, the program comprehension phase must be done correctly and completely before one can proceed to the localization phase. Otherwise, a wrong candidate module may be found.

2.5 Impact Analysis

This is an important and difficult phase in software maintenance. Without proper attention to and mastery of the candidate system design, it would be hard to find out how the intended modification would cause side effects in the system, and where these would occur. The consequences of the proposed modifications, and the effect of the overall complexity of the system must be examined at this stage, with reference to the candidate system design.

2.6 Implementation Plan

In this phase, implementation of the intended modification is planned. It includes how to update the existing specification and design documents, and how to re-code and configure the new and modified components of the system. This is the final phase in the maintenance process. When all these phases are completed the usual development process is activated: the maintenance process triggers the requirement analysis phase in the development process, which entire process is split into smaller tasks that can be developed by teams. However, the model in Fig. 1 only shows how to prepare the maintenance environment, and to define strategy for software maintenance. It actually does not consider the development of the existing software product. Ways in which the implementation details of the maintenance process can be carried out in conjunction with the development process are discussed in the next section.

3.0 AN EXTENDED SOFTWARE LIFE CYCLE

The maintenance model needs to be integrated into the entire software life cycle which will show how it can be linked with the development process. In such a model, it is important to show that the development environment can support the underlying maintenance method and activities within a software development life cycle framework, i.e. the life cycle model must satisfy both the development and maintenance processes. The early top-down life cycle model fails to represent many elemental tasks necessary for maintaining existing systems, and gives rise to misleading concepts of software maintenance. Software maintenance is viewed differently now than was the case in the past and it has been stated that a new life cycle model is needed to cope with the maintenance problems [7, 14]. In fact, in the present context, maintenance is seen as a continuation of the development process that begins the moment a software product starts its operation [15]. It has been claimed that a significant part of software maintenance is in itself the development of new functions [15]. However, although many software maintenance models have been proposed so far, most of them do not provide clear guidelines on how to integrate maintenance into the development process. However, if we integrate the maintenance model in Fig. 1 with the development process, the resulted scenario of the life cycle model is illustrated in Fig. 2. This extended life cycle model focuses on phenomena that occur during software building and re-building. Thus the extended life cycle model can be used in both the software maintenance context as well as the software development context.

3.1 The Maintenance Context

The extended software life cycle model in Fig. 2 shows how phases in the development process are used in implementing maintenance decisions to the existing system. The arrows in the model in Fig. 2 are used to represent the direction of the process. The numbers of the corresponding phases show the sequence of each task.

Requirements analysis: In this phase (designated number 7), the specification of the existing system is re-specified and re-written according to the modification requests. The results of the phase 1 modification request analysis is added to the existing requirements specification.

Design: The re-design of the existing system reflecting the new request occurs in this phase. The design architecture is restructured according to the proposed change, and all design documents are updated.

Coding: The modified design artifact is transformed into a coded program.

Testing: This corresponds to the most crucial phase in the entire maintenance process in which the modified system is tested for ripple effects, existing test cases are updated, and regression testing is performed. The post-maintenance testing approach of the development phase uses the following two steps [16]:

- identification of the newly added, deleted, or modified program paths;
- re-design and re-run of the test cases which execute the identified paths.

Here the information gained in the impact analysis phase of the maintenance life-cycle will be useful.

Finally, configuration management and control are essential in this phase before the modified system is certified for installation. Operational manuals will also need to be updated.

3.2 The Development Context

When a completely new development is to take place, the sequence of the various phases looks like the extended life cycle model depicted in Fig. 3. Both Fig. 2 and Fig. 3 are essentially the same in nature, the only difference being the order in which the phases occur. The models shown in Fig. 2 (and 1) can still be followed in tracing reusable components in the development of a new software product.

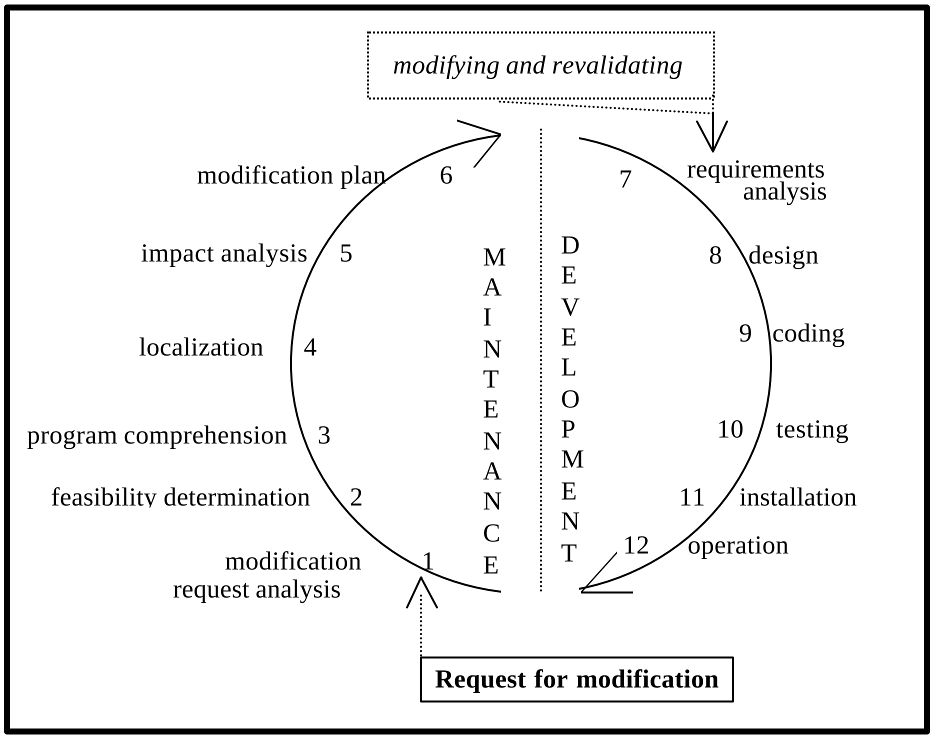

Fig. 2: An extended software life cycle model: the maintenance context

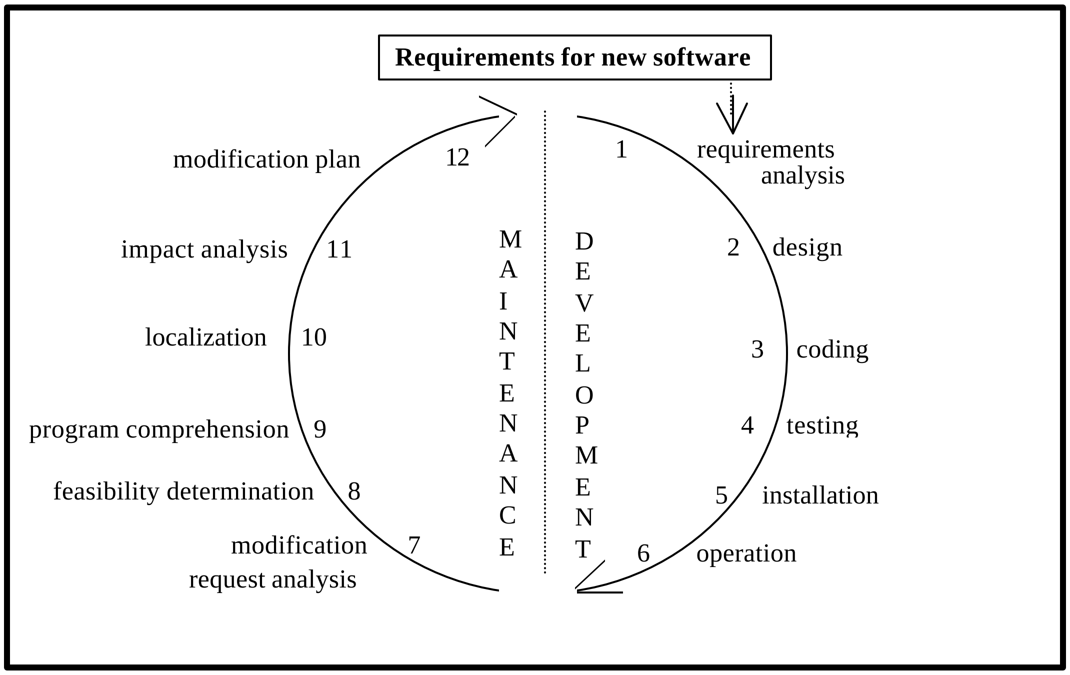

Fig. 3: An extended software life cycle model: the development context

The extended life cycle model in Fig. 2 can be interpreted as a generic software re-engineering model. It suggests that the re-engineering process begins with the existing system (phase 1, Fig. 2), and produces a new form of the old system (from phase 7 to phase 12 in Fig. 2). Interestingly it can be noted that the extended life cycle as presented in Fig. 3 illustrates the forward engineering process as well. This approach can be seen as similar to the spiral model, where the development cycle is repeated in ever increasing spirals.

4.0 COMPARISONS WITH OTHER MAINTENANCE MODELS

Before we proceed to compare our model with other maintenance models, it is important to point out that the "Extended Model" proposed here includes not only the development phase, but also the maintenance phase of the entire life cycle. It has been pointed out that [12], a well-defined process should have the ability to blend maintenance and development homogeneously into a single cycle. This is the essence of our "extended" life cycle model. It is not specific to maintenance tasks only but may be used for development tasks as well.

Another modified software life cycle model expressing the maintenance process integrated with software development has been proposed by Skramstad & Khan [17]. It shows similar steps but in a reverse time sequence. Theater Software Maintenance Environment (TSME) model [18] shows how the software maintenance process can be supported and automated through the use of an integrated software engineering environment. This high level process model does not address how this automated maintenance environment would be related to the development life cycle model. Ripple effects analysis and localization activities are not explicitly identified in the model. This model basically concentrates on process automation and process visibility.

A maintenance model proposed by Makoto Ino [19] is composed of five phases. It is very similar to a model of the development process. The five phases are: analyse user's maintenance requirement, design and approval, implementation, testing, and installation. The model in turn defines each task in these five phases, the actual work involved of each task and their input/output data. This model does not actually focus on how it can eventually be related to the development life cycle. However, the author did state that the model was not complete.

A generic maintenance process model adopted within the ESF/EPSOM project [20] may be seen at the first glance similar to that proposed here. A closer examination reveals that this is not the case. The model does not include an important maintenance phase like 'program comprehension'. This model comprises 11 main activities. These activities are grouped to form a maintenance process specific 'V' life cycle. The process is initiated with a trigger. It does not show the cyclic order of the entire life time of a product including the development component.

The central part of Maintenance Assistance Capability for Software (MACS) [21] is the understanding of the existing application software. It covers the phases of reverse engineering, modification management, and ripple effect analysis. It does not address how a maintenance process will be initiated. It does not identify or verify the real need for maintenance of a software product. On the other hand, the Durham Maintenance Model (DMM) [22] is essentially a process improvement model. This model considers managerial forecasting and management analysis in respect of software maintenance.

5.0 CONCLUSION

Software maintenance problems are somewhat more serious and important than they were thought to be and acknowledged by the software engineering community. The maintenance model and the extended life cycle model presented in this paper provide software organisations with a task-oriented framework on how to control their process for developing and maintaining software. The paper proposes that maintenance phases are better integrated into the development life-cycle, allowing for a fuller feedback loop. It can be effectively applied to large or medium-sized software products, and it requires a disciplined approach to ensure that the maintenance process is optimised. Researchers are also encouraged to verify the feasibility of this model in empirical studies.

REFERENCES

[1] G. Booch, *Object-Oriented Design with Applications*. Addison-Wesley, 1990, p. 5.

[2] M. Munro, “Software Maintenance, Reuse and Reverse Engineering”, *Proc. Reuse, Maintenance and Reverse Engineering of Software: current practice and new direction* Nov. 29-Dec. 1, 1989.

[3] E. B. Swason, “The dimensions of Maintenance”, *Proc. of 2nd Int. Conf. on Software Engineering*, San Francisco, Oct. 1976.

[4] B. W. Boehm, *Software Engineering Economics*, Englewood Cliff, Prentice Hall Inc. 1981, pp. 533-553.

[5] M. Haziza, J. F. Voidrot, E. Minor, L. Pofelski and S. Blazy, “Software Maintenance: An Analysis of Industrial Needs and Constraints ”, *IEEE Proc. Conf. on Software Maintenance*, Nov. 1992, pp. 18-26.

[6] W. S. Humphrey, *Managing the Software Process*, Addision-Wesley, 1989.

[7] B. Pearson, “Procuring New Systems for Maintenance” 4th European Software Maintenance Workshop, Durham, U.K., September 1990.

[8] B. W. Boehm, “Software Engineering”, *IEEE Trans. on Computers*, Vol. C-25, Dec. 1976, pp. 1226-1241.

[9] B. W. Boehm, “A Spiral Model of Software Development and Enhancement” *IEEE Computer*, May 1988, pp. 61-72.

[10] Aziz Deraman, “Requirement for a Software maintenance Process Model: A Review” *Malaysian Journal of Computer Science*, Vol. 8 No. 2, Dec. 1995. pp. 174-202.

[11] N. Chapin, “Attacking Why Maintenance is Costly-a software engineering insight” *IEEE Proc. Software Maintenance Workshop*, 1983, pp. 251-252.

[12] B. Curtis, “Maintaining the Software Process” *IEEE proc. Conf. on Software Maintenance*, 1992, pp. 2-8.

[13] N. Chapin, “Software Maintenance Life Cycle” *IEEE proc. conf. on Software Maintenance*, 1988, pp. 6-13.

[14] H. D. Rombach, V. R. Basili, “A panel Discussion, Position Statement”, *IEEE Conf. on Software Maintenance*, 1988, p. 3.

[15] C. Wild, K. Maly and L. Liu, “Decision-Based Software Development” *J. of Software Maintenance, Research and Practice.* John Wiley & Sons, Vol. 3, March 1991, pp. 17-43.

[16] P. Benedusi, A. Cimitile and U. de Carlini, “Post-Maintenance Testing based on Path Change Analysis", *Proceedings of the IEEE Conference on software Maintenance*, 1988, pp. 352-361.

[17] T. Skramstad and M. K. Khan, “A Redefined Software Life Cycle Model for Improved Maintenance”, *IEEE proc. Conf. on Software Maintenance*, 1992, pp. 193-197.

[18] R. Cherinka, C. M. Overstreet, A. Cadwell, J. Ricci, “Issues in Software Process Automation - From a Practical Perspective” *IEEE Proc. Conf. on Software Maintenance*, 1994, pp. 109-118.

[19] M. Ino, “Current State of Software Maintenance in Japan: In Depth View” *IEEE proc. Conf. on Software Maintenance*, 1992, pp. 27- 29.

[20] Del-Raj Harjani and Jean-Pierre Queille, “A Process Model for the Maintenance of Large Space Systems Software” *IEEE proc. Conf. on Software Maintenance,* 1992, pp. 127-136.

[21] C. Desclaux and M. Ribault, “MACS: Maintenance Assistance Capability for Software a K.A.D.M.E.”, *IEEE Proc. Conf. on Software Maintenance,* 1991, pp. 2-12.

[22] D. Hinley, and K. Benneth, “Developing a model to manage the software maintenance process” *IEEE Proc. Conf. on Software Maintenance,* 1992, pp. 174-182.

BIOGRAPHY

Md. Khaled Khan received his B.Sc. and M.Sc. degrees both in Computer Science from the University of Trondheim, Norway. His current research interests include software maintenance, reverse engineering, conceptual modeling of software systems, CASE technology and object-oriented design. He is a member of IEEE Computer Society, Southern African Mathematical Sciences Association.

Mohammad Abdur Rashid is a senior lecturer of Computer Science at the Department of Mathematics of the University of Brunei Darussalam. Dr. Rashid received an M.Sc. degree in Electronics Engineering from the Institute of Engineering Cybernetics, Faculty of Electronics of the Technical University of Wroclaw, Poland, in 1978 specialising in Engineering Cybernetics Systems and a Ph.D. in 1986 from the Department of Electronic and Electrical Engineering of the University of Strathclyde for his research on Packet Voice Communication on Carrier Sense Multiple Access Local Area Networks. Between years 1979 and 1990 Dr. Rashid worked as lecturer, assistant professor and associate professor at the University of Dhaka, Bangladesh. Dr. Rashid's research interests include Local Area Networks for real time multiclass communication, high speed networks, microprocessor based systems and computers and society. He has published research papers related to these areas in international journals and IEEE sponsored conference proceedings.

Bruce W. N. Lo, B Sc (London), M Ed Stud (Newcastle) and Ph.D. (Monash), is an Associate Professor and the Head of the Centre for Computing and Mathematics at Southern Cross University, NSW, Australia. Prior to that he has taught at Universities of Wollongong and Newcastle in Australia, and University of Guelph in Canada. His research interests include software metrics and cost estimation models, software tools and development environments, intelligence information systems, IT skills requirements and IT education, and computer-mediated learning.