1996

Achieving balance in software engineering curricula

Tim Comber
Southern Cross University

Bruce Lo

Richard Watson

Publication details

Comber, T, Lo, B & Watson, R 1996, 'Achieving balance in software engineering curricula', *Proceedings of the International Conference Software Engineering: Education and Practice - SEEP'96*, Berlin, Germany, 25-30 March, IEEE Computer Society Press, Los Alamitos, CA.

Achieving Balance in Software Engineering Curricula

Bruce Lo: blo@scu.edu.au
Richard Watson: rwatson@scu.edu.au
Tim Comber: tcomber@scu.edu.au

Southern Cross University
Lismore, NSW, Australia 2480

Abstract

Achieving balance is an issue that faces all curriculum designers. The complexity of the software process demands a pluralistic approach to systems development. This pluralism must also be reflected in the education and training of future software engineers. How can we integrate the diverse views into a unified curriculum framework? How can we cover all topics that are deemed essential for the discipline. We must balance specialised software engineering topics with fundamental topics in computer science and we must also balance the variety of software engineering topics amongst themselves within the relatively short three-year undergraduate curriculum. Very often, what is left out is just as important as what is left in. This paper describes the process and outcomes of a recent attempt at Southern Cross University to develop a balanced software engineering course. Comparisons were made between this implementation and the ACM/IEEE model curriculum and other related efforts.

Introduction

This paper describes the process and outcomes of a recent effort at Southern Cross University (SCU) to develop a software engineering course. The initiative is a response to changes in the Information Technology (IT) industry where graduates are expected to possess a much wider range of skills than those of traditional programmers and analysts.

In her classic book on curriculum, Taba [1] argued that a scientific approach to curriculum development needs to draw upon:

- analyses of society and culture,
- studies of the learner and the learning process and,
- analyses of the nature of the discipline and knowledge domain.

We believe software engineering curricula has an added dimension; it is based on a dynamic and fast evolving discipline.

Unlike the more established disciplines in the traditional sciences and humanities, software engineering is relatively "young", and its boundary and nature is constantly being shaped and moulded by current practices in the IT industry as well as the research findings of the community of scholars. While traditional academic disciplines like chemistry, physics or psychology are continuously being modified as the result of advances made at the frontier of research in those knowledge domains, software engineering is different in three fundamental aspects:

- The *definition* of the discipline is changing. Unlike chemistry or physics, a significant proportion of the content of the software engineering knowledge domain is continuously being redefined.
- The *speed* with which the changes are taking place is much faster than longer established disciplines such as chemistry or physics. The rate of change is driven by the rapid development of the technology itself.
- The *penetration* of industry software practices into undergraduate software engineering curriculum is much more significant than is the case in established disciplines, where industry research and development are more likely to be reflected mainly in the graduate curriculum.

What complicates the issue even further is that the legal status of the software engineering profession is not well established [2]. This state of flux is often manifested in the lack of agreement about what constitutes a reasonable program in software engineering, particularly at the undergraduate level [3]. Even when consensus was obtained on what may constitute model software engineering curricula [4], [5], [6], they too soon become dated.

Dynamic nature of the software engineering discipline

The dynamic nature of the IT industry poses a special challenge to those responsible for the development of educational curriculum in the training of software professionals. While standard system design and programming skills are still considered essential, new developments in information and communication technology, and new perspectives in the understanding of software project management have highlighted the importance of human as well as technical skills.

Several studies [7], [8], [9] have shown a tendency among employers of IT professionals to value general business and interpersonal skills over technical skills. This is not necessarily an actual shift but may reflect a general reaction towards the recognition of the need for good project management and people skills to ensure successful completion of software projects. The tutorial series on software engineering project management [10] is an example of such recognition. People and management skills are difficult to teach. Some would even argue that professional maturity is gained only after some experience in "real-life", in industry. This has led some to regard software engineering education as belonging to graduate programs [11] rather than to a first degree course. More recently, it has been accepted that the reality of the industry is such that IT graduates are increasingly required in their first employment to work with large teams, to use new and complex software tools and to be able to communicate effectively both formally and informally in their work environments [12].

On the technical side, the emphasis of system development has changed considerably during the past decade. The question of "make or buy" has raised the issue of system acquisition and adaptation as opposed to custom development. Software re-use and re-engineering tools and techniques have now become important repertoire of all but the very few IT professionals working in niche areas. Looking from a different angle, it is noted that many organisations still rely on computer information systems that were developed many years ago. The maintenance, re-engineering and re-structuring of these legacy systems is likely to remain a substantial part of the organisational total IT effort. Finally, with greater emphasis on distributed computing and the drive toward client/server computing, the very nature of programming has also undergone substantive changes. An example is the drive toward visual programming [13] together with the trend towards object technology both of which are impacting on programming practices.

Different approaches to software engineering

There are many different approaches that have been adapted to the study of software engineering. Each approach differs in the emphasis it places on the particular phase (or phases) of the systems development life cycle. The most common approaches are:

- the systems design approach
- the programming approach
- the tool-kit approach
- the process modelling approach
- the management approach
- the formal approach

The *systems design* approach emphasises the "design" aspect of the system development process. The basic premise is that good software systems are essentially the results of good design. Most of the earlier software engineering books tend to emphasise this approach eg [14], [15]. The more recent text by Conger [16] also employs this approach. It may be regarded as an extension of the traditional curriculum in systems analysis and design. More recently, the emphasis in software system design has moved toward the object paradigm [17].

The *programming* approach to software engineering emphasises the "coding" or "system implementation" phase of the system development process. The text by Bell, Morrey and Pugh [18] is a good example. This approach examines the process of algorithm analysis, program design, coding and program testing. It is the natural extension of earlier courses in computing programming.

Another approach that is closely related to the above two is the *tool-kit* approach (or the "trades-person" approach). Here the emphasis is on the professional tool set, the "bag of tricks" as may be colloquially referred to, that software engineers may use in the development of software systems. The tool-kit may include analysis or design tools (upper CASE), programming/coding tools (lower CASE), configuration management tools, or project management tools [19], [20].

The *process modelling* approach examines the software process in its entirety. Recognising the shortcomings of waterfall model, many alternative models were suggested. These include, the prototyping model, the iterative-exploratory model, the spiral model. The concept of "software factory" is at the basis of this approach. The emphasis is often on the practical or experimental aspects of the software process [21], [22].

The *management* approach is closely associated with the process modelling approach. However, the emphasis here is on the need for good project management with clearly identified time lines and project milestones. The success of a software project depends just as much on the management of human resources as on the availability of technical resources [10]. Boehm's treatment on software

risk management is another example of this approach [23].

The *formal* approach recognises that the critical steps are at the beginning of the systems life cycle, where user requirements are first analysed. It emphasises the need to ensure correct requirement specifications to avoid software errors. Proponents of this approach, often feel that the best way to ensure correct and unambiguous "requirement specs" is to employ formal methods [24]. They argue that once a system is correctly and unambiguously specified; the actual development and implementation of software system is relatively straightforward.

The above approaches may be classified into two categories: the *micro* approach versus the *macro* approach. The formal, systems design, programming or tool-kit approaches may be regarded as the micro approach where attention is focused on a single phase of the systems life cycle. The process modelling and management approaches took an overview of the system process, and thus may be regarded as the macro approach.

Achieving balance

The software process is a very complex one. While each of these approaches has its merits, the truth probably lies in a combination of these. The complexity of the software process demands a pluralistic approach to systems development. This pluralism must also be reflected in the education and training of future software engineers. The challenge now becomes: how to integrate the diverse views into a unified curriculum framework?

A common problem that faces all curriculum designers is how to cover all topics that are deemed essential for the discipline. The development of an undergraduate software engineering curriculum is no different [25]. Very often, what is left out is just as important as what is left in. The question of achieving balance in a software engineering curriculum is in fact two-fold. Firstly, there is the *inter-discipline* balance, where more specialised software engineering topics have to compete with the fundamental topics in computer science. Secondly, there is the *intra-discipline* balance, where the variety of software engineering topic have to compete among themselves in a relatively short three-year undergraduate curriculum.

The team at Southern Cross University decided to address this problem from a different angle. Rather than being constrained to explicitly align each course unit or module within the range of essential software engineering topics, the topics are embedded in the various curriculum units. We believe that the philosophical orientation of a course unit is just as important as its explicit title. The essential software engineering principles are thus built into the curriculum unit structure and integrated into the total curriculum. In fact a spiral approach is adapted to the study of software engineering principles. The topics are revisited several times during the course of the study. Each successive visit will cover the topics in greater depth and with a more comprehensive view of the software process. In the remainder of this paper, the authors will attempt to give a more detailed description of the software engineering course implemented at Southern Cross University.

The course development process

A Course Review Workgroup consisting of staff and postgraduate students was established to provide a focal point for discussion of the need for change and the direction for proposed changes in the computing courses at Southern Cross. This course review was undertaken as a regular process of review [26] and in response to:

- rapid changes in Information Technology [27];
- changes in curriculum recommendations by ACS, ACM, IEEE, and IFIP [6];
- recommendations of the Computing Discipline Review (Hudson report) [28];
- feedback from surveys of past students.

The course review was designed to ensure the currency and relevance of the curriculum content in the Bachelor of Information Technology program and to produce graduates with a "balanced organisational, individual and technical outlook on the information and computer field" [29]. The proposed course was documented and passed to computing industry professionals as well as experienced academics for review. The workgroup revised the course in light of the advice offered before submitting the course for approval.

The aims of the course.

The aims for the Bachelor of Information Technology were identified as:

- to produce high quality graduate capable of meeting the demands of the information technology profession;
- to develop communication skills in the student and to emphasise that the use of information technology is based on an understanding of the needs of users, clients, employers and colleagues;
- to prepare the student to cope with the rapidly changing information technology environment;
- to teach students the concepts, theories, techniques and skills applicable to Information Technology;
- to inculcate in students the need for continuing professional development to keep abreast of information technology developments;

- to develop in students an understanding of the need for careful analysis of situations, a synthesis of new solutions and an evaluation of the impact of these solutions;
- to nurture in students a respect for research in information technology and to provide them with an understanding how research may be conducted.

More specifically, the aims of the *Software Engineering Major* of the Bachelor of Information Technology course are:

- to provide students with a good understanding of the theory and practice of software engineering;
- to impart to students the skills needed to efficiently develop and maintain high quality software systems;
- to provide students with a good knowledge of how to participate in and manage software projects.

The Bachelor of Information Technology: Software Engineering

The Bachelor of Information Technology (B Inf Tech) at SCU will require the completion of 24 semester units:

- 15 Core Computing Units;
- 2 elective computing units;
- 3 core mathematics units;
- 2 elective non-computing units;
- 2 core non-computing units.

Those majoring in Software Engineering must take Operating Systems and Computing Architecture, Introduction to Operations Research, Interface Development and Evaluation, Programming Languages, Artificial Intelligence, Software Engineering, Computing Project, and Information Resource Management. Figure 1 shows the progression of units and the logical dependencies between units in the Software Engineering major. Computing units are shown in rectangular boxes.

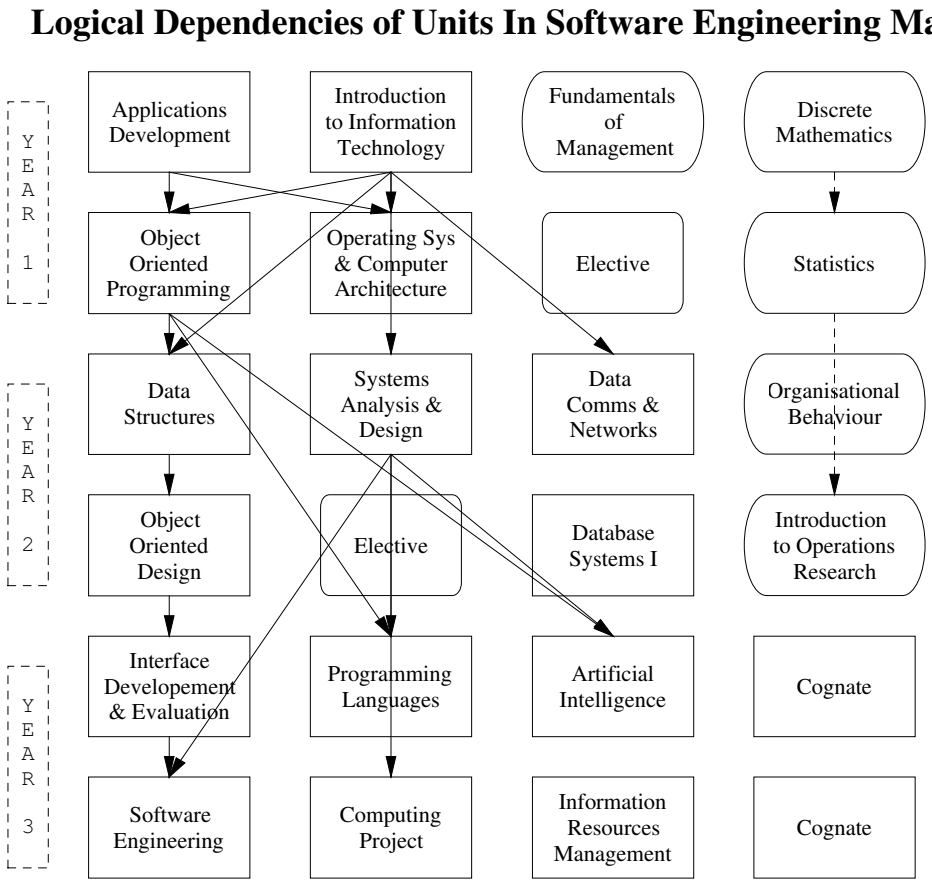

Figure 1. BIT course structure

It is important to produce graduates with a broad educational background [25]. In line with this, Fundamentals of Management and Organisational Behaviour introduce our students to management and social interaction principles. Discrete Mathematics, Statistics, and Introduction to Operations Research

provide the mathematical background necessary for a sound knowledge of basic computational skills, the principles of formal methods in the Software Engineering unit, and for future research.

During the course review much thought was given to the inter-relationships between units and where they would fit within the three-year structure. One important concern was to have a progression through the four main programming units; Application Development, Data Structures, Object Oriented Programming and Interface Development and Evaluation with each subject being dependent on the previous. This means that for each unit the lecturer concerned can assume that certain key concepts have been covered, and new ones can be built on them.

A separate group produced a report on programming languages where after much consideration it was decided to move from Pascal as a first language to C. It was also decided that it was a waste of scarce resources to teach more than one introductory programming language. Another concern was to concentrate on one language (or family of languages) as the principle programming language used across most units thus allowing more advanced concepts to be covered. C and C++ were chosen.

Comparison with other curricula

Having developed a course that meets our defined goals and needs, it is reasonable to compare it with other published software engineering curricula. It is compared with the well-known ACM/IEEE software engineering curriculum [6], and with a recently developed Australian software engineering degree course [30] at Queensland University of Technology (QUT). Table 1 summarises how course content of the three courses compares in terms of broad subject areas.

- The ACM/IEEE course is delivered over four years and is based on a core of computing and mathematics subjects, and augmented with numerous electives in the humanities and sciences as well as general electives. The SCU and QUT courses run over three years, and so there is much less scope for non-computing subjects and electives. The last row, labelled "ACM/IEEE-3", enables a more direct comparison to Australian courses by removing enough non-computing elective subjects to produce a three-year course.

Several points of comparison may be noted from Table 1. The SCU and QUT courses are much more closely matched than the ACM/IEEE course. The ACM/IEEE course has a much lower proportion of computing and mathematics subjects than the SCU and QUT courses. Even after removal of a large proportion of non-computing material, the computing and core mathematics content of the ACM/IEEE-3 course is only 55% compared with 83% and 79% for SCU and QUT respectively. We assume that this reflects general differences in the tertiary education between the United States and Australia, rather than a major divergence in the computing curricula. A minor difference is that the SCU course has a lower proportion of computing electives relative to the computing core, and that the ACM/IEEE course has a higher proportion of core mathematics subjects. The core—elective relationship for the SCU course indicates the intention of the course designers to ensure that relevant software engineering topics outside the core of the ACM/IEEE curriculum were included as core items. The reverse philosophy applies to mathematics: students can choose to undertake non-core mathematics subjects as electives.

A general conclusion could be drawn that the SCU course is more discipline-specific than the ACM/IEEE model which aims to provide a broader coverage at a lower depth.

Table 1. General course comparison

	Percentage of total course content				
	core SE	**elective computing**	**core maths**	**elective non-computing**	**core other**
ACM/IEEE	24.3	6.4	10.7	54.3	4.3
SCU	62.5	8.3	12.5	8.3	8.3
QUT	70.8	8.3	0	16.7	4.2
ACM/IEEE-3	32.4	8.6	14.3	39.0	5.7

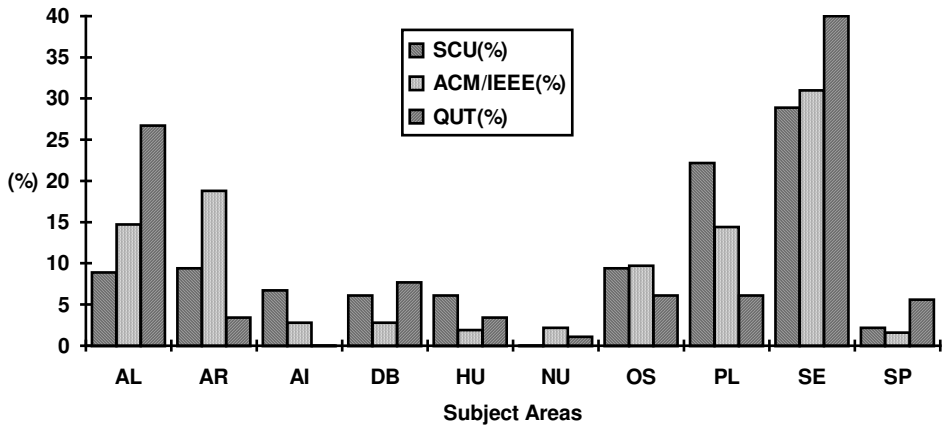

Figure2. Comparison of different course structures

Table 2. Course comparison by subject area

	AL	AR	AI	DB	HU	NU	OS	PL	SE	SP
SCU:ACM	0.60	0.50	2.36	2.17	3.25	0.00	0.97	1.54	0.93	1.42
SCU:QUT	0.33	2.81	--	0.80	1.82	0.00	1.55	3.65	0.72	0.40

The ACM/IEEE report specifies the following fundamental set of ten subject areas:

- Algorithms and Data Structures (AL),
- Architecture (AR),
- Artificial Intelligence and Robotics (AI),
- Database and Information Retrieval (DB),
- Human-Computer Communication (HU),
- Numerical and Symbolic Computation (NU),
- Operating Systems (OS),
- Programming Languages (PL),
- Software Methodology and Engineering (SE) and,
- Social, Ethical, and Professional Issues (SP).

It further describes the minimum content from each subject area considered necessary to form the basis of a software engineering course.

Figure 2 and Table 2 compare the SCU, QUT, and ACM/IEEE SE courses, on the basis of the ten subject areas. Figure 2 shows the percentage total core subjects (including project work) devoted to the various subject areas.

Table 2 shows the ratio of SCU's proportion to the corresponding proportion in the other courses. A minor point to be noted is that the SCU course contains non-core contribution to the NU subject area. This is covered as the mathematics elective Numerical Analysis.

We are able to make some interesting comparisons that highlight the changing nature of software engineering education.

Consider the areas in which we differ most from the ACM/IEEE model. Clearly we have chosen to emphasise database, artificial intelligence, and especially human-computer topics at the expense of the more traditional architecture and (to a lesser extent) algorithms subject areas. We justify this shift in emphasis by noting that major changes in software technologies have occurred since the ACM/IEEE curriculum was published in 1991, and we are attempting to respond to these. The most influential developments from our point of view have been the emergence the WIMP-style of user interface, greater reliance by industry on 3 and 4GLs, and the increased use of networked computers in information systems. To some extent this philosophy is shared by the QUT course — database and human-computer areas are emphasised, and architecture is de-emphasised, though we note a complete absence of AI content! We note that the QUT and SCU data differ widely in the AL and PL areas, though the *sum* of these two areas is very similar; we assume that this simply reflects a difference between the allocation of sub-topics to the AL and PL areas, in the two course summaries.

Another divergence in this course is that we have sought to structure the sequence of the units to build knowledge in a gradual and incremental manner, rather

than cover an area in one or two concentrated units. This is particularly evident in the software engineering and programming subjects, which are spread uniformly over the full three years of the course. By comparison, in the ACM/IEEE course, software engineering subjects appear in only the first three semesters and programming subjects in the following two semesters.

Conclusions and plans for the future

We have presented a new course in SE which has some significant differences in emphasis with comparative courses. These differences arise in part from the set of choices we made in seeking to balance the often conflicting requirements of a new course. The major divergence from older curricula appears to be a shift in emphasis from the traditional computer science core curriculum which stressed depth in a small number of key areas, to a more balanced approach where traditionally less significant areas like database and HCI contribute more to the core curriculum. More recent curricula tend to confirm our approach, though differ sufficiently for us to conclude that curriculum design is clearly still a highly subjective field!

Recognising the need to approach the software process from different perspectives, our course attempts to include the various approaches, i.e. systems design, programming, tool-kit, formal specification, process modeling and project management, within its curriculum framework. However, there may be some who would argue for a monolithic rather than pluralistic approach.

While it is possible to devise a curriculum where software engineering principles and techniques are singled out and studied as stand-alone topics, the approach that we have chosen is not to deal with them in abstract, but to treat them in the context of various applications areas. The close relationships between theory and practice are thus maintained.

References

[1] Taba, H. (1962) *Curriculum Development, Theory and Practice*, Harcourt, Brace and World: New York.

[2] Jones, C. (1995) Legal status of software engineering, *IEEE Computer*, May 1995, pp.98-99.

[3] Tymann, P.T., Lea, D. and Raj, R.K. (1994) Developing an Undergraduate Software Engineering Program in a Liberal Arts College, *Proceedings of Twenty-Fifth SIGCSE Technical Symposium on computer Science Education*.

[4] Denning, P.J., Commer, D.E., Gries, D., Mulder, M.C., Tucker, A., Turner, A.J., and Young, P.R. (1989), Computing as a Discipline, *Communications of the ACM*, Vol 32, No 1, pp.9-23.

[5] Ford, G.S. (1991) The SEI Undergraduate Curriculum in Software Engineering, *SIGCSE Bulletin*, Vol 23, No 1, March 1991, pp.375-385.

[6] Tucker, A.B. (1991), Computing Curricula 1991, A summary of the ACM/IEEE-CS Joint Curriculum Task Force Report, *Communications of the ACM*, Vol 34, No. 6, pp.69-81..

[7] Crockett, H.D., Hall, G.R. & Jeffries, C.J. (1993) Preferred Information Systems Skills: Are undergraduate IS programs serving their markets?, *Interface*, Vol 15, Issue 2, pp.9-13.

[8] Cassidy, M.J. (editor) (1990) *Education and training needs of computing professionals and para-professionals in Australia*, *Volume 1: Data and findings; Volume 2: Methodology and appendices*, Department of Employment, Education and Training, Australian Government Publishing Service, Canberra.

[9] Informatics (1994) Job skills for 1994: What's in demand?, *Informatics*, March 1994, Australian Computer Society.

[10] Thayer, R.H. (1988) *Software Engineering Project Management*, IEEE Computer Society Press,: Los Alamitos, CA.

[11] Gibbs, N.E. (1989) The SEI Education Program: The challenge of teaching future software engineers, *Communication of ACM*, **32** (5) 594-603.

[12] Mohay, G., Morarji, H., and Thomas, R. (1994) Undergraduate, Graduate and Professional Education in Software Engineering in the 90's: A Case Study, *Proceedings of the Southeast Asian Regional Computer Confederation SRIG-ET'94 Software Engineering Conference*, University of Otago, November 1994, pp.167-171.

[13] Snell M (1995) Analysts predict $3.79 billion market for visual development tools by 1999, *Computer*, March 1995, pp.8-9.

[14] Ince, D.C. (1989) *Software Engineering*, Chapman and Hall, London.

[15] Finkelstein, C. (1989) *An Introduction of Information Engineering: from strategic planning to information systems*, Addison-Wesley: Singapore.

[16] Conger, S. (1994) *The New Software Engineering*, Wadsworth Pub Co.: Belmont, CA.

[17] Meyer, B. (1991) *Object Oriented Software Construction*, Prentice Hall.

[18] Bell, D., Morrey, I. and Pugh, J. (1992) *Software Engineering: a Programming Approach*, Prentice-Hall International: Hemel Hempstead, Hertfordshire, UK.

[19] Case, A.F. Jr. (1986) *Information Systems Development: Principles of computer-aided software engineering*, Prentice-Hall, Englewood Cliff, NJ.

[20] Martin, J. (1985) *Fourth-Generation Languages*, Prentice-Hall: Englewood Cliffs, NJ.

[21] Pressman, R.S. (1992) *Software Engineering: A Practioner's Approach*, McGraw-Hill: New York.

[22] Humphrey, W.S. (1991) Software and the factory paradigm, *Software Engineering Journal*, pp.370-76.

[23] Boehm, B.W. (1989) *Software Risk Management*, IEEE Computer Society Press, Washington, D.C.

[24] Wordsworth, J.B. (1992) *Software Development with Z: a Practical Approach to Formal Methods in Software Engineering*, Addison-Wesley.

[25] Hurst, A.J., Hurst, B.J., and Finlay, A. (1994), Breadth versus depth in software engineering education, *Proceedings Software Education Conference*, November 1994, University of Otago, New Zealand, IEEE Computer Society Press, pp.191-195.

[26] Southern Cross University. (1994) Policies and Procedures: Review of Courses.

[27] Hammond, J.H. (1989),Education and training needs for computing professionals: Seeking new strategies, *Proceedings of the IFIP TC 3/WG 3.4 Working Conference on Methodologies of Training Data Processing Professionals and Advanced End-Users*, July 1989, Helsinki, Finland, Elsevier, pp.73-80.

[28] Hudson, H.R. (Chair) (1992) *Report of the Discipline Review of Computing Studies and Information Sciences Education, Volume 1: A Summary, Volume 2: The Main Report and Volume 3 Supporting Work*, Department of Employment, Education and Training, Department of Industry, Technology and Commerce, and Information Industries Education and Training Foundation Limited, Australian Government Publishing Service, Canberra.

[29] Kerola, P.(1989) Knowledge about human information processing and learning styles in the education of system architects, *Proceedings of the IFIP TC 3/WG 3.4 Working Conference on Methodologies of Training Data Processing Professionals and Advanced End-Users*, July 1989, Helsinki, Finland, Elsevier, pp.3-14.

[30] Thomas, R., Semeczko, G., Morarji, H. and Mohay, G. (1994) Core software engineering subjects: A case study('86-'94), *Proceedings of the Southeast Asian Regional Computer Confederation SRIG-ET'94 Software Engineering Conference*, University of Otago, November 1994, pp.24-31