AN UNDERGRADUATE SOFTWARE ENGINEERING MAJOR EMBEDDED IN A COMPUTER SYSTEMS ENGINEERING DEGREE

by K. Reed, MSc, FACS, MIE(Aust) and T. S. Dillon, PhD, FIE(Aust), SMIEEE

Department of Computer Science
La Trobe University
Bundoora, Melbourne 3001
Victoria
Australia

ABSTRACT

This paper describes an undergraduate major stream in Software Engineering embedded in a four year Bachelor of Computer Systems Engineering Degree. This major allows students to specialize in Software Engineering in their third and fourth years, with the result that some 60% of their time is dedicated to Computer Science and Software Engineering.

This contrasts with the post graduate Master's programs offered in the US, and the relatively minor Software Engineering subjects offered in some undergraduate Computer Science courses.

1. INTRODUCTION

The fact that traditional engineering concepts can be applied to software production is finally being widely recognized. It is now accepted that Software Engineering, a practicable discipline similar in form to other established engineering disciplines, can be considered to have come of age.

At the same time, the need for formal education for software engineers is becoming generally accepted, although there are relatively few degree courses with dedicated to this subject. The need for specialized courses in Software Engineering arose from the widely held belief that conventional Computer Science and Business EDP degrees do not really concentrate on producing practitioners capable of building reliable software to a given specification and to some predetermined schedule [GIB89a]. Specifically tailored programs were first introduced about ten years ago, and were based on Master's degrees by course work [FRE87], [KEE81] following a series of curriculum proposals by ACM [FAI87]. Today, some twenty years after the original NATO Conferences on Software Engineering, serious consideration is only just being given to the introduction of undergraduate Software Engineering degrees.

Interestingly enough, similar arguments can be applied to Digital Computer Engineering, and to Computer Network Engineering, bodies of knowledge which are currently (in Australia at least) taught as minor elements of either Computer Science or Electronic Engineering degrees.

Recent studies in Australia have provided evidence of a chronic shortage in each of the above areas. This prompted the Department of Computer Science at La Trobe University in Melbourne Australia, to develop a four year undergraduate Computer Systems Engineering Degree, with majors in Software Engineering, Digital Computer Engineering and Computer Network Engineering.

This paper concentrates upon the Software Engineering major, however, the the other two majors provide students with an opportunity to acquire a specialized qualification in courses especially constructed for that purpose, something which is rare in this country at least.

We begin by summarizing the current state of Software Engineering education, and relate that to both Digital Computer and Computer Network Engineering. We then provide an outline of the complete degree structure, and continue by dealing with the difference between Software Engineering and Computer Science as seen by educators. We conclude with a detailed description of the Software Engineering major, summarizing the syllabi, and relating them to the existing undergraduate Computer Science offerings.

2. THE COMPUTER SYSTEMS ENGINEERING DEGREE

2.1. Motivation

Software Engineering education currently falls into approximately two categories:

a) Master's by course work degrees dedicated to the subject, such as those pioneered by Seattle University [LEE81] and the Wang Institute ([ARD87] and [ARD85], and [MCK87].

b) A cursory outline of Software Engineering, usually concentrating on project management and systems analysis, and incorporating a team subject. This would usually be taught as a final year component in an undergraduate Computer Science degree.

Both of these approaches are unsatisfactory, although the underlying reasons for their existence are understandable. The English tradition of "liberal arts" education which has been adopted in the United States and many other countries (including Australia) means that the total amount of teaching hours available for vocationally oriented material is limited. The time available for specialist subjects which could legitimately be included in a Computer Science major is therefore restricted, preventing additional space from being made available for Software Engineering.

There has also been a strong view, particularly in the United States that software engineering should be only taught to people who already have an undergraduate qualification and significant industrial experience (see [GIB89A] p272). Increasingly however, the view is being taken that this approach is unsatisfactory because it smacks of locking the stable door after the horse has bolted.

An alternate view is that proper software engineering habits should be learnt at the outset of one's career as a software producer. This is to be preferred to the current approach of trying to correct a lengthy period of inappropriate practices which were acquired during basic undergraduate training with the full authority of academic staff. Fortunately, the last few years have seen a gradual increase in the undergraduate emphasis on Software Engineering, however, the current situation still leaves a lot to be desired.

A slightly different problem exists with respect to computer systems engineering, an area, which, in recent times, has seen a rash of offerings in Computer Engineering. These have frequently been mounted by Electronic Engineering Departments and sometimes consist of little more than the study of the application of microprocessors in embedded computer systems for process control or real time applications.

The reality is that computer engineering or computer systems engineering is in fact a much broader discipline requiring a proper understanding of the engineering of an entire computer system. A detailed knowledge of computer architecture, computer communication systems, software systems and lastly of multi-processor computer systems must be acquired before one can be considered a Digital Computer Engineer.

Recognition of this fact is leading to a reappraisal of the notion of computer systems engineering as is illustrated by the recent move of Computer Engineering from the Electronic Engineering Department to the Department of Computer Science and Engineering at the University of Washington. Recognition that Digital Computer Engineering is a discipline in its own right, much in the manner of Civil Engineering, Mechanical Engineering, Electrical or Electronic Engineering, rather than a sub-branch of Electronic Engineering is likely to grow. At the same time, practitioners and educators are becoming increasingly aware that the the principles underlying this discipline are somewhat different from those that underly the enabling technologies of microelectronics and digital design.

Similar remarks apply to Computer Network Engineering.

The authors' view is that all three bodies of knowledge have matured to a point where they can be regarded as a set of related but separate disciplines.

This had led the Department of Computer Science at La Trobe University to develop a four year undergraduate degree in Engineering providing for major studies in each of the above disciplines. This paper will provide an outline of the degree but will focus on the Software Engineering component in some detail.

2.2. Aims of the Course

The aims and objectives of the course are to produce qualified professional personnel in the fields of:

1. Software Engineering

2. Computer Architecture and Systems

3. Computer Networks

These personnel should be capable of participating effectively in the design, construction and management of each of the relevant systems. The best graduates should also be well equipped to carry out research in each of the above areas. The course has, however, been designed to be vocational in intent and has been designated as an engineering rather than a science course.

We have chosen this approach since we believe that science is essentially concerned with the understanding of the phenomena under study and the development of new knowledge, whether these phenomena are natural or manmade.

Engineering on the other hand is distinguished by the need not only for an understanding or an analysis of systems but also a strong emphasis on synthesis, and design and construction of systems. It is felt that these processes play a primary role in engineering activities, and that they are best taught within the context of an engineering framework. Amongst other things, it is our belief that recognition of the above has a significant impact on the practical work prescribed as well as the choices that are made with respect to course content.

In the latter context we regard traditional Computer Science and Electronics as enabling technologies which have the same relationship to Software Engineering, Digital Computer Engineering and Network Engineering as do Physics and Mathematics to Mechanical and Civil Engineering and Chemistry to Chemical Engineering. In particular there is, in Software Engineering, a body of experimental evidence which suggests that the natural problem solving processes used by programmers and system developers require a substantial knowledge of a large number of basic concepts and techniques. This is a matter which we have attempted to address in this course.

2.3. Course Structure

The first two years of the Degree are common to all streams with students being able to select one or the three major streams after the third and fourth year. Common core material is necessary in the basic sciences and Mathematics, and in the enabling technologies of Computer Science and Electronics. This is included in the first two years. (See figure 1 below).

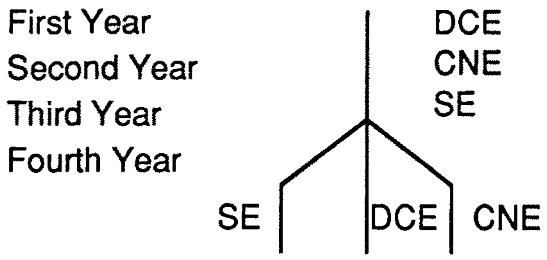

Figure 1. BCSE Degree Structure

Specialization in the principles of Computer Systems Engineering begins in the second year. This reflects our view that Computer Systems Engineers should, irrespective of chosen specialization, be well educated in the basic technologies and the basic sciences concerned. In the third year there is an opportunity to specialize in each of the three streams, while still taking some units from the other two. For example Digital Computer Engineering majors will undertake more computer architecture and related material than related Software Engineering majors. Software Engineering majors will focus very heavily on their chosen filed, while still taking some material in relation to computer architecture and networks. This specialization increases considerably in the fourth year of the course, and allows the individual disciplines to be given adequate weight, a situation that is not possible in traditional educational systems.

There is an emphasis throughout the course in understanding both the software and hardware implication of each of the major streams.

The belief that an engineering education should involve a strong theme of "learning by doing" is reflected in a strong component of practical work in each of the second, third and fourth years. This is achieved through the medium of laboratory classes which are closely supervised, whilst a considerable component is in the form of project work. The emphasis in Software Engineering will be on the acquisition of practical experience in applying techniques in second and third year within the framework of group projects. A major team project in fourth year would be conducted in a simulated software house environment, Horning's famous "software hut" [**HOR77**].

3. SOFTWARE ENGINEERING AND COMPUTER SCIENCE EDUCATION

3.1. What is Software Engineering?

Software Engineering is the result of bringing traditional engineering discipline to bear on the process of building software systems. Software Engineering is a discipline dependent on a number of technologies. It uses **managerial techniques** to control and manage projects, **systems analysis** techniques to capture a series of descriptions of a system that is capable of being converted, by **programming**, into a machine executable form, as has been pointed out by Reed [**REE87**] in 1987.

It was further suggested that in broad terms [**ibid**], that one would expect, for any discipline to be described as "engineering" that it should allow the:

a) Design of a system capable of performing a specified "function".

b) Design of a system to a specified performance given certain available run-time and implementation resources.

c) Implemention of a system to a required time-cost schedule.

d) Maintainence and enhancement of the system during its life.

These goals can be achieved through...

A) Component re-use, both code, module and design,

C) Tool use, to control and lend power to the process of system description, code production, testing and quality control,

C) Design iteration, to meet both performance and functional goals,

and

D) Management techniques to describe the system development process, estimate resource requirements, and monitor resource usage.

The above constitutes an extension of the definitions given in Richard Fairley's book [FAI85] in that we explicitly recognize the element of performance as a possible design criteria, and add the design iteration requirement. Interestingly enough, otherwise excellent Software Engineering textbooks do not attempt a compact definition of the field. See for example Pressman [PRE88].

These goals cannot be met without a large number of techniques and methodologies many of which we currently associate with Systems Analysis and Design [HAW88] and with programming.

Requirements b) & c) cannot be achieved without iteration in the design process, since both the system performance and implementation cost will depend upon aspects of the design.

It should be kept in mind that requirements b) & c) are the keys to effective software projects and products - without them we produce system that will not run on available hardware. The fact is that run-time resource utilization, performance and implementation are affected by the design itself, making design iteration and perhaps prototyping essential. (We should point out that design iteration may also be necessary to determine the functionality of a system cf. prototyping etc. [CAR83]).

It is possible to consider an engineering design process to be one in which a set of parameters are successively altered until a set of target relationships between all of them is obtained, while another set of relationships (constraints), is maintained. The design process may include back-tracking as well as iteration [REE87].

3.2. The Educational Implications of Two Differing Disciplines

Computer Science has as its goal the systematic development of knowledge about all aspects of computing systems, and their use. In other words, the Computer Scientist is primarily engaged in the discovery of new knowledge about computing, while the Software Engineer is interested in applying that and other knowledge.

There is, in this context, a fundamental difference between the prime objectives of Computer Science and Software Engineering education. In particular Software Engineering education must be concerned with the acquisition of practical skills relating to the use of existing techniques and technologies coupled with the development of managerial skills.

Computer Science education, on the other hand, tends to be biased towards the theoretical issues and the study of existing systems with a view to the development of new concepts and knowledge (See Sommerville pp302-303 [SOM85]).

As an example, one would expect to find the formal study of algorithm and module re-use at a practical level in a Software Engineering course, but one would not be surprised to find such a topic omitted from an undergraduate Computer Science offering.

Further differences are apparent when considering practical assignments offered in each case. Assignments in Computer Science tend to exhibit the following features, as was recently noted by Ciesielski et. al [CIE88]. They tend to be:

a) small individual assignments designed to test ingenuity, or familarise the student with some new technique or language feature,

b) large individual projects,

c) team projects, such as the implementation of a compiler, or a modest commercial system.

Assignments are frequently artificially constructed to meet narrow pedagogic goals, and the results are not used by those other than the developers. However, in DP departments and software houses programmers are required to work as teams, to generate programs and documentation that will be used by others and to define and satisfy the the needs of users, often working from poor specifications. There is frequently a large gap between the educational experience of students and the situation in the work place, as has been noted by other software engineering educators [MCK87], [BUR87], and [FRE87].

In addition, it is not always the case that there are sufficient resources available for detailed verification of a student's submission's correctness. Issues such as modular design, and elementary module reuse are also unlikely to be used as the basis for designing assignments.

The result is that students in a traditional Computer Science degree tend to see a satisfactory program as one which will pass a cursory assessment procedure when tested using selected data.

A Software Engineering programming assignment, on the other hand, may be set to provide experience in some particular technique. In addition, the standard of testing and documentation required may be significantly higher. Ideally, sufficient staff resources should be available to allow incorrect code and poor design to be detected - with students being asked to correct deficiencies for a passing grade.

Software Engineering practical work may also contain a "maintenance experience", that is, the discovery and correction of errors in some large piece of software (see [TOM87]), some modifications involving the addition of functionality to some previously developed piece of software may be required.

3.3. A Philosophical Basis for Software Engineering Education

The authors of this course proposal have extensive experience as both Computer Science and Software Engineering educators. Both are qualified engineers, and believe that their approaches have been influenced by their respective basic educations. The proposed degree, which will be offered beginning in 1990, is based upon a particular philosophical view of both engineering and Software Engineering[1].

In addition, the syllabus designers, had, as a primary goal, the training of graduates capable of producing high quality software. They were substantially influenced by Parnas' papers [PAR72], by work on both State Transition Diagrams ([DAY70] and [WAS85]) and Petri Net techniques as system design and program implementation techniques [STO89], by Basili's approach to metrics and quality evaluation [BAS85], by Beohm's work on estimating [BEO82] and by Belady's general approach to Software Engineering. Other techniques emphasized include modular programming[2] and interpretive and table-driven systems.

The philosophical underpinning of our syllabi can be summarized as follows, although the priority is not necessarily absolute:

- **We assume that product and design re-use are the fundamental means of achieving effective software development (see** [SIL85]). *This leads us to promote modular design, data-driven programming techniques and the acquisition of a wide range of algorithmic techniques.*

- **We assume that Software Engineers require an extensive knowledge of traditional system description techniques.** *This leads us to include the study of a variety of description and development methodologies usually associated with commercial systems analysis.*

- **We assume that quantative measures of product and project quality and behavior are important.** *This leads us to treat software metrics, project structure and organization, software testing* [MYE79], and reliability.

- **We assume that tools are essential means of increasing software productivity.** *This leads us to study their design and use.*

- **We assume that effective specification of software is a prerequisite to good design.** *This leads to the study of formal and informal analysis and specification techniques, and to the study of prototyping techniques(see* [CAR83]).

- **We assume that effective project management is necessary.** *This leads us to the study of estimating techniques, estimating, project management and the software process.*

---

[1] The first author has nine years experience in conducting a three subject undergraduate major in SE in a prior incarnation, while the second author has extensive experience with the use of protocol engineering and reliability engineering techniques in large scale software development.

[2] The first author was substantially influenced by early experience with the KDF9 K Autocode and IBM Fortran 4-E subroutine libraries in the mid 1960's.

- **We assume that a wide knowledge of programming languages and utilities such as database and operating system interfaces is required.** *This leads us to the study of database and operating systems,*

  and finally,

- **We assume that a Software Engineer should have a good general Computer Science education.** *Which means that the course contains a complete undergraduate major on Computer Science*[3].

This material is spread throughout the course in a manner intended to develop both practical and theoretical skills.

4. THE DEGREE STRUCTURE

All students undertake a common first and second year, which includes Software Engineering related material in Computer Science I (about 10%), and in Computer Systems Engineering II (about 50%).

Students specialize in third and fourth year, with all candidates taking Computer Science III and components of Computer Systems Engineering III in third year. Fourth year Software Engineering majors take only Computer Science and Software Engineering. This is shown in a little meore detail in Table I.

As already discussed, all students other than the Software Engineering majors take a Software Engineering team project in third year. The latter under take their project in fourth year as part of Software Engineering IV, and operate in a Software Hut mode.

5. THE SOFTWARE ENGINEERING MAJOR STREAM

5.1. First Year

First year students are introduced to some basic Software Engineering topics. These include an introduction to

a) the Waterfall model of the software process **[AGR86]**,

b) modular design, information hiding concepts, and the use of existing modules,

c) simple prescriptive testing procedures **[MYE79]**,

[3] The normal undergraduate major offered by LTU would occupy about 30% of the students total load over three years. As such, it is used as the basis for the BSE degree.

TABLE I. SUBJECTS IN THE BCSE DEGREE

First Year (4.5 UNITS) (COMMON)

Electronics I	(1.0)
Physics ICS*	(0.5)
Computer Science I	(1.0)
Computer Systems I	(0.5)
Mathematics IA	(1.0)
Mathematics IDM	(0.5)

Second Year (3.5 UNITS) (COMMON)

Electronics II	(1.0)
Applied Maths II	(0.5)
Computer Systems Engineering II*	(1.0)
Computer Science II	(1.0)

MAJOR STREAMS

Third Year (2.4 Units)

Software Engineering		Digital Computer Engineering		Network Engineering	
Computer Sci. IIIA*	(0.8)	Computer Sci. III	(1.0)	Computer Sci. III	(1.0)
Comp. Sys. Eng. IIIB	(0.5)	Comp. Sys. Eng. IIIB	(0.5)	Comp. Sys. Eng. IIIA*	(0.5)
Software Eng. III*	(0.7)	Comp. Sys. Eng IIIB*	(0.5)	Comp. Sys. Eng. IIIC*	(0.5)
Statistics IIICS*	(0.2)	Statistics IIICS*	(0.2)	Statistics IIICS*	(0.2)
Soc. Imp. of Eng. *	(0.2)	Soc. Imp. of Eng. *	(0.2)	Soc. Imp. of Eng. *	(0.2)

Fourth Year (1.25 Units)

Software Engineering		Digital Computer Engineering		Network Engineering	
Comp. Sci. IVA*	(0.50)	Comp. Sci. IVA*	(0.50)	Comp. Sci. IVA*	(0.50)
Software Eng. IV	(0.50)	Comp. Sys. Eng. IVA*	(0.25)	Comp. Sys. Eng. IVA*	(0.25)
Comp. Sys. Proj. IV*	(0.25)	Comp. Sys. Eng. IVB*	(0.25)	Comp. Net. Eng. IV*	(0.25)
		Comp. Sys. Proj. IV*	(0.25)	Comp. Sys. Proj. IV*	(0.25)

* New Subjects

d) informal specification techniques,

and

e) an analysis-synthesis approach to program design which maximizes simple module re-use.

It should be kept in mind that ALL Computer Science students take Computer Science I, and therefore take some basic Software Engineering.

The subject Computer Systems I contains material on graphics, human interfaces, C and other topics.

5.2. Second Year

Computer Systems Engineering II is, as already mentioned, roughly 50% Software Engineering, allowing 4 hours of lectures per week for two semesters. The course focuses on the following topics, building on earlier material:

a) System description techniques, such as Data Flow Diagrams, Jackson's methodology [**JAC83**], structure charts etc.,

b) Modular programming,

c) Object Oriented approaches,

d) Macrogenerators,

e) Database systems and their interfaces,

f) Operating System interfaces,

g) Transportability and practical algorithm reuse.

Students also take Computer Science II which is a fairly conventional second year course.

5.3. Third Year

Students majoring in Software Engineering take Software Engineering III, Computer Science IIIA, Statistics IIICS and Social Implications of Engineering, as well as Computer Systems Engineering IIIB, which is an advanced computer architecture unit.

Software Engineering III deals in depth with a substantial body of the discipline, supporting lectures with practical assignments designed to demonstrate the value of the techniques presented. A total of four hours of lectures are available each week, for two semesters.

Topics presented include:

a) Software metrics, both structure and quality,

b) System description techniques, such as State Transition Diagrams, the NEC SPD and others [AZU85],

c) Interpretive and table-driven programming techniques, and co-routines

d) Productivity issues, software and process re-use, fourth generation and special purpose languages, application generators,

e) Software Security and Reliability,

f) Transportability, "virtual" systems, operating system interfaces,

g) System partitioning issues, eg IBM SAA

h) Test coverage metrics, software maintenance and modifiability

Computer Science III contains a wide range of components which complement the Software Engineering material. Students taking the Software Engineering major must take Computer Science IIIA, which consists of units in Fourth Generation Languages and in Business Management, as well as six other components from the rest of that offering. Available topics include Parallel Computing, Formal languages, Graphics, Artificial Intelligence[4], and Workload Analysis.

Statistics IIICS deals with queuing theory and related material, and is one hour per week for the whole year, while Social Implications of Engineering addresses ethics, environmental and other issues relevant to the relationship between Engineering and society.

5.4. Fourth Year

The final Software Engineering subject serves two purposes, and runs for two semesters at four hours per week. The first is to introduce the student to advanced topics likely to be influencing the future directions of the field, while the second is to provide

[4] AI is also introduced in first year.

practical knowledge of estimating and project management techniques.

Again, students must take Computer Science IVA, which is a selection of the standard Honours year offering, including the Advanced Software Engineering component therein.

Software Engineering IV proposes to deal with:

a) Formal Methods,

b) CASE tools and their history,

c) Impact of system structure on functionality and performance,

d) Software reliability,

e) Software quality control and measurement, including the TAME concept (see Basili and Rombach [BAS88]),

f) Computer resource usage, monitoring and estimation,

g) Contractual and project management issues,

h) Configuration management and version control,

i) Information and Data Engineering,

j) System testing

k) Application of A.I. to software engineering,

l) Experimental methods and data collection.

Final year students also complete a major project which will be a team project, run on Software Hut lines.

6. CONCLUSION

The Software Engineering course outlined above will produce graduates able to make a substantial contribution to solving the Software crisis. We see no real difficulty in mounting the Degree, and are confident that students will be able to cope with the subject matter. We also believe we have demonstrated that an undegraduate program in Software Engineering is readily achievable.

In addition, we submit that the ease with which we were able to identify material for our syllabi is ample proof that we have a discipline mature enough to warrant such a course.

7. ACKNOWLEDGEMENTS

The author's would like to acknowledge those members of the Department of Computer Science who assisted in the design and construction of the Degree. Special thanks are due to Dr. Rhys Francis and Dr. Ian Robinson for assisting in ensuring that the proposal met the requirements of the University's various committees. Our thanks also go Ms. Kit Martin, Ms. Colleen Pearce and Ms. M. Clamp who prepared the various documents that were needed. Thanks are also due to our colleagues in the Electronic Engineering Department for their stimulating comments and advice.

In the end, however, the views expressed in this paper are the authors, and any errors or omissions are their responsibility.

REFERENCES

[AGR86] Agresti, W.W. "The Conventional Software Life-cycle Model: Its Evolution and Assumptions", in IEEE Tutorial on New Paradigms for Software Development, Agretsi, W. W. (ed) 1986 pp. 2-6

[ARD85] Ardis, M., Brouhana, J., Fairley, R., Gerhardt, S., Martin, N., and McKeeman, W. "Core Course Documentation: Master's Degree Program in Software Engineering". School of Information Technology, Wang Inst. Graduate Studies Tech. Report TR-65-17 Sept. 1985

[ARD87] Ardis, M. "The Evolution of the Wang Institute's Master of Software Engineering Program", IEEE Trans. on Software Engineering vol. SE-13 no. 11 1987

[AZU85] Azuma, M., Tabata, T., Oki, Y. and Kamiya, S. "SPD: A Humanized Documentation Technology", IEEE Trans. on Software Engineering vol. SE-11 no. 9 Sep. 1985 pp. 945-953

[BAS85] Basili, V.R. "Quantitative Evaluation of S.E. Methodology", (Keynote Address) Proc. First Pan Pacific Computer Conference, Melbourne Australia, Sep. 1985

[BAS88] Basili, V.R. and Rombach, H.D. "The TAME Project: Towards Improvement-Oriented Software Environments", IEEE Trans. on Software Engineering vol. SE-14 no. 6 Jun 1988 pp. 758-773

[BEO82] Boehm, B.W. "Software Engineering Economics", Prentice-Hall 1982

[BUR87] Burns, J.E. and Robertson, E.L. "Tow Complementary Course Sequences on the Design and Implementation of Software Products", IEEE Trans. on Software Engineering vol. SE-13 no.11 1987

[CAR83] Carey, T.T., and Mason, R.E.A. "Information System Prototyping: Techniques, Tools, Methodologies", INFOR - Canadian Journal of Computational Research and Information Processing Vol. 21 No. 3 May 1983 pp. 177-191

[CIE88] Ciesielski, V.R., Reed, K. and Cybulski, J.L., "Experience with a Project Oriented Course in Software Engineering", Proc. of the Australian Software Engineering Conference (ASWEC), May 1988 pp. 125-131

[DAY70]

Day, A.C. "The use of symbol-state tables", Computer Journal Vol. 13 No. 4, Nov 1970

[FAI85] Fairley, R.E. "Software Engineering Concepts", McGraw-Hill 1985

[FAI87] Fairley, R.E. "Guest Editor's Introduction", IEEE Trans. on Software Engineering vol. SE-13 No. 11 Nov. 1987 pp. 1141-1142, Special Issues on Software Engineering Education

[FOR89] Ford, G.A. and Gibbs, N.E. "A Master of Software Engineering Curriculum", IEEE Computer, vol. 22 no. 9, Sep. 1989 pp. 59-71

[FRE87] Freeman, P. "Essential Elements of Software Engineering Education Revisited", IEEE Trans. on Software Engineering vol. SE-13 no. 11 Nov 1987 pp.1143-1148

[GIB89a] Gibbs, N.E. "The SEI Education Program: The Challenge of Teaching Future Software Engineers", Comm ACM, Vol. 32 No. 5, May 1989 pp. 594-605

[GIB89b] Gibbs, N.E. "Is the Time Right for an Undergraduate Software Engineering Degree?" in Proc. Software Engineering Education Conference July 1989, Springer-Verlag LCNS 376, Gibbs, N. E. (ed)

[HAW88] Hawryszkiewycz, I.T. "Introduction to Systems Analysis and Design", Prentice-Hall 1988

[HOR77] Horning, J.J. and Wortman, D.B. "Software Hut: A computer program engineering project in the form of a game", IEEE Trans. on Software Engineering vol. SE-3 No. 4 Jul 1977 pp325-330

[JAC83] Jackson, M.A. "System Development", Prentice-Hall 1983

[LEE81] Lee, K.Y. "Status of Graduate Software Engineering Education", Proc. ACM81

[MCK87] McKeeman, W.M. "Experience with a software engineering project course", IEEE Trans. on Software Engineering vol. SE-13 no. 11 Nov 1987 pp. 1182-1192

[MYE79] Myers, G.J. "The Art of Software Testing", Wiley, 1979

[PAR72] Parnas, D.L. "On the criteria to used in decomposing systems into modules", Comm ACM, vol. 15 no. 2 1972

[PRE88] Pressman, R.S. "Software Engineering, A Practitioners Approach", McGraw-Hill,1988

[REE87] Reed, K. "Commercial Software Engineering, the Way Forward", Keynote Address to the Australian Software Engineering Conference (ASWEC) Cnaberra, May 1987

[SIL85] Siverman, B.C. "Software Cost and Productivity Improvements: An Analogical View" IEEE Computer Vol. 18 No. 5 May 1985 pp. 86-96

[SOM85] Sommerville, I. "Software Engineering", 2nd ed. Addison-Wesley, 1985

[TOM87] Tomayko, J.E. "Teaching Maintenance Using Large Software Artifacts", Proc. Software Education Conference Pittsburgh July 1989 Springer Verlag LNCS 376 pp. 3-15

[WAS85] Wasserman, A.I. "Extended State Transitions diagrams for the Specification of Human Computer interfaces", IEEE Trans. on Software Engineering vol. SE-11 no. 8 1985