Technical Report

CMU/SEI-90-TR-003
ESD-TR-90-204
March 1990

1990 SEI Report on Undergraduate Software Engineering Education

Gary Ford

Software Engineering Curriculum Project

Approved for public release.
Distribution unlimited.

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This technical report was prepared for the

SEI Joint Program Office
ESD/XRS
Hanscom AFB, MA 01731

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

Review and Approval

This report has been reviewed and is approved for publication.

FOR THE COMMANDER

Karl H. Shingler
SEI Joint Program Office

This work was sponsored by the U.S. Department of Defense.

Copyright © 1990 Carnegie Mellon University

This document is available through the Defense Technical Information Center. DTIC provides access to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential contractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center, Attn. FDRA, Cameron Station, Alexandria, VA 22304-6145.

Copies of this document are also available through the National Technical Information Services. For information on ordering, please contact NTIS directly: National Technical Information Services, U.S. Department of Commerce, Springfield, VA 22161.

Table of Contents

Table of Figures

1990 SEI Report on Undergraduate Software Engineering Education

Abstract: Fundamental issues of software engineering education are presented and discussed in the context of undergraduate programs. Included are discussions of the definition of software engineering and its differences from computer science, the need for undergraduate software engineering education, possible accreditation of undergraduate programs, and prospects for professional certification and licensing of software engineers. The objectives and content of an undergraduate program are described, as are strategies for the evolution and implementation of such programs. An appendix presents a report on the 1989 SEI Workshop on an Undergraduate Software Engineering Curriculum.

1. Introduction

1.1. Background

The Software Engineering Institute (SEI) was established at Carnegie Mellon University in December 1984, under a contract with the United States Department of Defense. Its primary mission is to advance the state of the practice of software engineering by accelerating the transition of promising new methods and technologies from concept demonstration to routine use. A significant part of the strategy is to promote software engineering education as a means to help alleviate the chronic shortage of highly qualified software engineers. In describing the institute's role in education, the SEI charter states, "It shall also influence software engineering curricula development throughout the education community."

The SEI Education Program was established to undertake this task. During our first four years, we concentrated on master's level curriculum development [Ford87, Ardis89, Gibbs89a, Ford89b]. We believed that this work would provide the quickest "payoff," in that universities could establish and students could complete a master's program more quickly than an undergraduate program.

However, because the majority of computer science undergraduates never pursue an advanced degree, we also believed it to be essential to find ways to increase the software engineering capabilities of professionals who earn only a bachelor's degree. In 1989, therefore, we began to investigate undergraduate issues as part of a long-term strategy for software engineering education.

1.2. Issues in Software Engineering Education

Our investigations of undergraduate software engineering education identified many of the same issues that we had previously considered in the context of graduate education. These included the following:

- What is the definition of *software engineering*?
- How is software engineering different from computer science?
- Is software engineering a sufficiently mature discipline that a university degree program is appropriate?
- What are the objectives of a software engineering degree program?
- What is the content of a software engineering degree program?
- Is a bachelor's degree in computer science an essential prerequisite for meaningful study of software engineering?

In the context of undergraduate education, some additional issues were raised, including:

- Is undergraduate software engineering education necessary?
- Can software engineering be taught at this level?
- Can an undergraduate software engineering degree program exist only in a university's engineering school, or can it exist wherever a computer science department exists?
- Can an undergraduate software engineering degree program be accredited as an engineering program? As a computer science program?
- Can a graduate of such a program be called a software engineer?
- Can a graduate of such a program be licensed as a professional engineer?

In the remainder of this report we will address these issues. Chapter 2 surveys some definitions of the term "software engineering" in order to help distinguish it from computer science. Chapter 3 addresses the necessity of undergraduate software engineering. Issues of program accreditation and professional licensing are investigated in Chapters 4 and 5, respectively. Chapters 6 through 8 present ideas

about how undergraduate programs might evolve. Chapter 9 summarizes the report and presents our recommendations to the academic community.

Many of these issues were discussed at the SEI Workshop on an Undergraduate Software Engineering Curriculum, which was held in Pittsburgh on July 21, 1989. The results of the workshop are summarized in Appendix 1 of this report.

1.3. Purpose of This Report

The SEI is a mission-oriented organization. We accept the premise that software-based systems will continue to be important to the economy and defense of the United States. We also accept the premise that education is an essential part of the solution to the problem of producing, on a continuing basis, a skilled professional work force of sufficient size to produce those systems. Education will necessarily occur at many levels, including secondary school, college, and graduate school, plus continuing education of professionals.

The development of appropriate educational programs is a significant task requiring the efforts of a large number of individuals and organizations over many years. We believe it is appropriate for the SEI to undertake activities to accelerate that development. One of the first steps is to promote widespread rational discussion of the issues, problems, and potential solutions of software engineering education. That is the fundamental purpose of this report.

We recognize that many of the issues addressed by this report are controversial and have provoked passionate debate. We shall try to present objectively a diverse collection of information that we believe is relevant to software engineering education, in the hope that it will stimulate increased and more reasoned debate, especially among educators and software professionals. Readers should keep in mind, however, that we believe that undergraduate programs in software engineering, whatever their formal names may be, are a desirable and inevitable part of the spectrum of software engineering education.

2. Software Engineering and Computer Science

A fundamental question that arises in discussions of software engineering education is whether software engineering and computer science are separate disciplines each deserving academic degree programs. There is no easy answer to this question. To promote discussion, this chapter presents a selection of the definitions of the term "software engineering" that have appeared in the literature.

2.1. Definitions of "Software Engineering"

F. L. Bauer, one of the principal organizers of the 1968 NATO conference that led to the widespread use of the term "software engineering," gives this definition [Bauer72]:

> The establishment and use of sound engineering principles (methods) in order to obtain economically software that is reliable and works on real machines.

The IEEE glossary [IEEE83] defines software engineering as:

> The systematic approach to the development, operation, maintenance, and retirement of software.

The National Science Foundation sponsored a study by the American Federation of Information Processing Societies to develop a taxonomy of computer specialist occupations [NSF88]. The taxonomy includes ten categories: computer scientist, computer hardware engineer, computer software engineer, telecommunications specialist, systems programmer, systems analyst, programmer, computer operations specialist, technical support specialist, and computer trainer. A computer scientist is defined to be "[a]n individual, usually with an advanced degree, who is engaged as a theorist, researcher, designer or inventor (or any combination of these roles) in the fields of computer hardware or software." A software engineer is defined to be "[a] highly trained specialist, usually with a degree in either engineering or computer science, who applies state-of-the-art knowledge to the design of overall software systems, to the setting of operational specifications, quality standards and testing procedures, and to the definition of user needs." This is distinguished from a programmer, who is "[a] specialist, usually with a college degree, who writes, tests and applies the instructions that define the operations performed by a computer."

Software engineering textbooks have also taken up the challenge to define software engineering. This definition appears in [Fairley85]:

> Software engineering is the technological and managerial discipline concerned with systematic production and maintenance of software products that are developed and modified on time and within cost estimates.

In [Humphrey89], there are a number of basic definitions of terms related to the software process, including this one of software engineering:

> The disciplined application of engineering, scientific, and mathematical principles, methods, and tools to the economical production of quality software.

The British Computer Society and the Institution of Electrical Engineers answer the question "What is software engineering?" with a rather lengthy description [BCS89]:

> Software Engineering requires understanding and application of engineering principles, design skills, good management practice, computer science and mathematical formalism. It is the task of the Software Engineer to draw together these separate areas of expertise and bring them to bear upon the requirements elicitation, specification, design, verification, implementation, testing, documentation and maintenance of complex and large scale software systems. The Software Engineer thus fulfils the role of architect of a complex system, taking account of user requirements and needs, feasibility, cost, quality, reliability, safety and time constraints. The necessity to balance the relative importance of these factors according to the nature of the system and the application gives a strong ethical dimension to the task of the Software Engineer, on whom the safety or well-being of others may depend, and for whom, as in medicine or in law, a sense of professional morality is a requirement of the job. Sound engineering judgement is required.
>
> The Software Engineer must be able to estimate the cost and duration of the software development process, and determine the achievement of correctness and reliability. Such measurement and estimation may involve financial and managerial understanding as well as sound grasp of mathematical concepts. The precise use both of formal notations and of words is necessary to express them with a degree of precision meaningful to other engineers and informed clients. In most circumstances the technical, theoretical and managerial strands of the Software Engineer's task cannot be pursued in isolation from each other.
>
> Both to build large products and to achieve high productivity from skilled labour, Software Engineering requires the use of software development tools and of components and reusable components capable of subsequent safe modification and maintenance.
>
> The task of the professional Software Engineer spans the range of activities involved in the lifecycle of a software system. Requirements elicitation, specification, design, verification and construction are all critical in achieving the quality of the product and are all the responsibility of the Software Engineer.
>
> Since software determines the behaviour of an automaton, the Software Engineer needs to understand digital hardware and communications. Although the discipline of Software Engineering can be identified independently of application area, its realisation must be in the context of specific applications. The Software Engineer must therefore be able to collaborate with other professionals who bring complementary skills to the task of specifying, designing and constructing a hardware-software system which serves

the needs of the client, makes use of hardware and software solutions in optimum combination, and provides good quality human-computer interfaces.

Most software is built by teams, often interdisciplinary teams. The ability to work closely with others is essential.

Some of the intellectual tools and methods of Software Engineering are at present still in process of development, and rapid change is to be expected for some time to come. Software Engineers therefore need the theoretical understanding which will be a foundation for learning and using new methods in the future, and the cast of mind which sees the constant updating of knowledge as required professional behaviour.

The report also specifically rejects two usages of the term:

Software Engineering is not simply a more organized approach to programming than that which was prevalent in the early days of computer science and remains widespread among amateurs or through lack of education and training.

Software Engineering is not the design of programs to be implemented primarily in traditional engineering applications. It is the software itself which needs to be engineered, irrespective of its application.

The Software Engineering Institute undertook a small effort to develop a definition that would be suitable for its own use in planning and explaining its activities:

1. Core definition:

   - Engineering is the systematic application of scientific knowledge in creating and building cost-effective solutions to practical problems in the service of mankind.
   - Software engineering is that form of engineering that applies the principles of computer science and mathematics to achieving cost-effective solutions to software problems.

2. Elaborations or interpretations:

   - For software, “creating and building” must include maintenance. We have used the word “achieving” to cover the entire software life cycle.
   - “Cost-effective” implies accounting not only for the expenditure of money, but also clock time, schedule, and human resources. “Cost-effective” also implies getting good value for resources invested; this value includes quality by whatever measures are considered appropriate.
   - Software engineering is not limited to applying principles only from computer science and mathematics–like any engineering discipline it is based primarily in principles from one discipline but may draw on whatever principles it can take advantage of.
   - Similarly, software engineering, like all engineering, draws on the principles and techniques of management in order to carry out its activities of “creating and building.”

3. Distinction between current use of the phrase “software engineering” and definition appropriate to SEI mission:

   - At present, the phrase “software engineering” has multiple sets of poorly understood and conflicting meanings, ranging from simple coding to management to system design. We recognize that the phrase has connotations we do not accept.
   - At present, the phrase “software engineering” is more an aspiration than a description. This should not deter us from aspiring to the definition above.

2.2. Discussion

[Denning89] includes “software methodology and engineering” as one of the nine subareas of the computing discipline. On the other hand, [Jensen79b] includes computer science as one of the major components of software engineering education. It is clear that there is no consensus on whether either field is a subset of the other.

Software engineering and computer science are probably somewhere near the midpoint of their evolution into distinct disciplines. They will eventually bear a relationship to each other very much like that of the more traditional engineering disciplines to their fundamental sciences.

The implications for education will derive primarily from the differences between science and engineering. We have seen several catchy phrases that try to capture those differences, such as “The scientist analyzes; the engineer synthesizes,” [Jensen79b], or “A scientist builds in order to learn; an engineer learns in order to build,” (heard from Fred Brooks; see [Gibbs87], p. 5). The fundamentally different philosophies of science and engineering will ultimately determine much of the content and pedagogy of undergraduate programs in the two disciplines.

The most important statement in the preceding definitions is probably this one from the BCS/IEE report [BCS89]: “Software Engineering is not simply a more organized approach to programming than that which was prevalent in the early days of computer science and remains widespread among amateurs or through lack of education and training.” Software engineering education will not be achieved by adding modern programming languages and techniques to existing computer science courses, nor by adding a group programming course to the curriculum. An engineering approach to the whole curriculum is necessary. In Chapters 7 and 8 of this report we describe how this might be achieved.

3. The Need for Undergraduate Software Engineering Education

From time to time we have seen studies that purport to have quantified the need for software engineers in the near future. For example, in [OTA89] the Office of Technology Assessment says, "The shortfall of software professionals in the United States is estimated at 50,000 to 100,000 and is forecast to grow steadily over the next decade." The basic assumptions underlying those studies and the resulting numbers vary widely, so we do not consider any of them to be definitive. However, we do accept the general premises that computers and software will be increasingly important in the fabric of society for the indefinite future and that there will be an increasing need for software professionals.

The educational system generally acknowledges a responsibility to serve the needs of both the individual and society. Higher education tries to balance these needs by providing fundamental knowledge, intellectual and reasoning skills, and specific career knowledge and skills. It is our opinion that a legitimate goal of higher education is to produce an appropriate number of graduates who are well prepared both to function in a technological society and to pursue careers in the computing disciplines.

The SEI Education Program has a major goal of assisting the academic community to provide high-quality software engineering education. As we mentioned in the introduction, our activities have emphasized master's level education during our initial four years in order to achieve the quickest "payoff."

Recent evidence, however, supports the premise that the vast majority of the software engineering work force will have only an undergraduate degree. Some of that evidence is presented below. It has led us to the conclusion that we should begin more substantial efforts to help the academic community to provide better undergraduate software engineering education.

3.1. Trends in Undergraduate Enrollments in the Computer Sciences

Trends in computer science enrollments offer some insight into the difficulties we face in providing a highly qualified software engineering work force. Figure 3.1 (derived from data in [NSF88]) shows reasonably consistent growth in the number of computer science degrees granted in the United States through the early 1980s, and then the beginning of a declining growth rate in the mid-1980s. That trend is perhaps clearer in Figure 3.2, which shows data from [NSB86] on the percentage of college freshmen choosing computer science majors. (We have also heard that one

study places the number of freshmen choosing computer science majors in 1989 at less than one percent.) Partial data for 1984-1988 is shown it Figure 3.3 (derived from data in [Gries89]); this data reflects bachelor's degrees in computer science and computer engineering granted by schools that also grant doctorates in those fields. This data shows a clear leveling off of the growth rate in the second half of the 1980s. We have also found considerable anecdotal evidence that computer science enrollments have decreased nationwide since 1985.

Year	BS degrees	Increase	% Increase
1976	5,664	–	–
1977	6,426	762	13
1978	7,224	798	12
1979	8,769	1,545	21
1980	11,213	2,444	28
1981	15,233	4,020	36
1982	20,431	5,198	34
1983	24,678	4,247	21
1984	32,435	7,757	31
1985	39,121	6,686	21
1986	42,195	3,074	8

Figure 3.1. Growth of degrees in computer sciences

Year	1983	1984	1985
% CS Majors	8.8	6.1	4.4

Figure 3.2. Percentage of freshman choosing computer science majors

Year	CS BS degrees	CS and CE BS degrees
1984-85	10,422	–
1985-86	10,947	–
1986-87	10,540	12,643
1987-88	10,759	12,687
1988-89	10,688	12,646

Figure 3.3. Bachelor's degrees from PhD-granting institutions

We have been unable to find reliable explanations for the apparent declining interest in computing. One possibility is that the wide availability of computers in high schools and in the home has satisfied the students' curiosity about computing before they reach college. Another speculation is that computing has lost some of its glamour along with other technical disciplines as part of a growing societal concern that technology causes as many problems as it solves. A third possibility is that computer science has simply earned a reputation among students of being difficult, especially now that it has evolved beyond just learning many programming languages.

One other fact is worth noting. Demographic statistics [NSB86] show that the number of 18-year-olds in the United States reached a maximum of about 4.2 million in each of the years between 1976 and 1982. In the early 1990s, that number will drop to about 3.3 million, and it will remain below 4 million well beyond the turn of the century. The combined factors of fewer college-age students and a smaller percentage of them choosing computer science majors indicate a potential for a substantial reduction in the number of new computing professionals in the coming decade.

3.2. Trends in Graduate Enrollments in the Computer Sciences

The number of master's degrees granted in the computer sciences from 1976 to 1986 are shown in Figure 3.4 (derived from data in [NSF88]). These figures suggest that, in the mid-1980s, only about 20% of computer science students pursued a master's degree. (We recognize that it is not necessarily the case that students pursue master's degrees in the same disciplines as their undergraduate degrees.)

Year	MS degrees	Increase	% Increase	BS degrees	MS % of BS
1976	2,603	–	–	5,664	46
1977	2,798	195	7	6,426	44
1978	3,038	240	8	7,224	42
1979	3,055	17	1	8,769	35
1980	3,647	592	19	11,213	33
1981	4,218	571	16	15,233	28
1982	4,935	717	17	20,431	24
1983	5,321	386	8	24,678	22
1984	6,190	869	16	32,435	19
1985	7,101	911	15	39,121	18
1986	8,070	969	14	42,195	19

Figure 3.4. Master's degrees in computer sciences

A more significant statistic is that only 5% of the recipients of computer science bachelor's degrees in 1984 and 1985 were enrolled as full-time graduate students in 1986, as compared with 25% of students in the sciences as a whole [NSF88]. This suggests a decrease in the number of master's degrees awarded in the late 1980s.

Again, the reasons for the declining interest in computer science education are not known. One of the probable reasons that students do not pursue a graduate degree immediately after receiving a bachelor's degree is that employment opportunities continue to be very attractive. We have found cases of starting salaries of almost $60,000.

The number of doctoral degrees in the computer sciences continues to increase, as shown in Figure 3.5 (derived from data in [Gries89]). Based on conversations with colleagues, we believe that only a small percentage of PhD students are working in software engineering, but that percentage may be increasing.

Academic Year	CS PhD degrees	CS and CE PhD degrees
1980-81	230	–
1984-85	326	–
1985-86	412	–
1986-87	466	559
1987-88	577	744

Figure 3.5. Doctoral degrees in computer sciences

3.3. The Current State of Undergraduate Degree Programs in Software Engineering

In [Ardis89], the SEI described eleven master's level programs in software engineering at United States universities. In this report we would like to describe the corresponding undergraduate programs, but there are currently none to report. However, we are aware of some schools that are actively investigating or developing undergraduate software engineering curricula.

The Florida Institute of Technology has committed to the development of a six-quarter (freshman-sophomore) sequence in formal program development for all computer science majors. The course development is being led by Harlan Mills (who is now a member of the FIT faculty), and the course content incorporates much of Mills' work on formal methods, including the "cleanroom" techniques. The school intends to develop additional software engineering courses for the junior and senior years, providing a complete program in software engineering. This program will replace their current computer science program.

The Rochester Institute of Technology is developing a four-semester sequence beginning with an introduction to software engineering at the sophomore level. The remaining three courses (junior-senior) are software specification and design, software testing and reliability, and a project course. Currently this sequence is a concentration within a computer science curriculum. Some of the faculty are hoping this concentration will expand to more courses over the next few years.

The University of Houston at Clear Lake has reported to us that they are trying to develop an undergraduate software engineering degree program. We do not have details of their progress.

In contrast, there are nearly 1000 colleges and universities in the United States that offer degrees in computer science. We believe that 10% to 20% of those schools could eventually offer undergraduate software engineering degrees. We hope to report their progress in the SEI's annual reports on undergraduate software engineering education.

3.4. Implications for Software Engineering Education

The facts described above have led us to conclude that, in the foreseeable future, the vast majority of new software engineers will have only a bachelor's degree and that their degrees will not be in software engineering. Most software engineers will complete their careers without an advanced degree. We therefore believe that those of us concerned about the quality and quantity of the software engineering work force should devote more of our efforts directly to improving the state of undergraduate software engineering education.

Part of the solution is to find ways to attract more of the best-qualified freshmen to study software engineering. Professional disciplines, including engineering, are widely respected in our society and continue to attract students. We hope that as the software engineering discipline matures, it will earn its share of respect, and more students will choose it as a career. Ultimately, software engineering may be more attractive to students than computer science. For the immediate future, however, we must undertake more direct efforts to attract students.

Students will explore career options only in areas with which they have some familiarity. Thus, it may be valuable to promote at the high school and college freshman level a wider understanding of the nature of software engineering and its intellectual challenges. It will also be valuable to target women and minorities, who have traditionally been underrepresented in the engineering profession.

Once we have attracted these students, the academic community must be prepared to provide appropriate educational opportunities to them. This will require the development of undergraduate courses, curricula, and faculty expertise. The SEI's Software Engineering Curriculum Project has a major goal of assisting schools and individual faculty members in this effort.

4. Accreditation Issues

As undergraduate software engineering programs begin to emerge, pressure for accreditation is likely to follow. Therefore, designers of such programs should be familiar with the potential accrediting agencies and their policies. The purpose of this chapter is to present relevant basic information on accreditation issues.

Accreditation has long been recognized as a mechanism for helping assure quality of educational institutions and academic programs. It has been especially significant in disciplines leading to professional practice, such as engineering. Accreditation of computer science programs began in the 1980s, resulting in increased awareness of issues of accreditation among educators and students. It is not surprising that discussions of undergraduate software engineering programs often include accreditation issues.

There are two accrediting bodies in the United States that might accredit software engineering programs: the Accreditation Board for Engineering and Technology and the Computing Sciences Accreditation Board. We will discuss each of these bodies in this chapter.

Before looking at these issues in some detail, we should note that the goal of accreditation is to define *minimum* standards for programs and that accreditation does not guarantee high quality (by whatever definition of quality we choose). Furthermore, accreditation is not universally accepted as important for the success of a program. Schools that are generally regarded as among the best in the world (such as Harvard University and the California Institute of Technology) have engineering programs that are not accredited.

Our discussions of these issues with engineering deans, computer science department chairs, and faculty members have uncovered a variety of concerns about the value of accreditation. Some have told us that the accreditation guidelines are too restrictive or too vocational. It seems likely that many very good computer science programs will never seek accreditation. Some accredited engineering programs may not seek renewal because the effort involved does not produce commensurate benefits. Thus, it is not necessarily the case that schools considering introducing a software engineering program must put accreditation at the top of their lists of issues. On the other hand, many large industrial companies have policies that they will only hire engineers who are graduates of accredited programs. This fact can be very important to students in their career planning.

4.1. ABET Accreditation

The Accreditation Board for Engineering and Technology, Inc. (ABET) is recognized by the U. S. Department of Education and the Council on Postsecondary Accreditation (COPA) as the sole agency responsible for accreditation of educational programs leading to degrees in engineering. Prior to January 1980, ABET was known as the Engineers' Council for Professional Development (ECPD). Under that name it first addressed accreditation of engineering programs in 1933.

ABET publishes a number of documents describing its policies, procedures, and accreditation criteria. We strongly recommend that educators or schools considering development of undergraduate software engineering tracks or programs obtain copies of these documents. ABET is located at 345 East 47th Street, New York, NY 10017; telephone (212) 705-7685.

In the next five sections, we examine the policies and criteria of ABET that we believe to be particularly interesting or potentially relevant to software engineering programs. The information presented is taken from [ABET88]. In Chapters 7 and 8 we discuss implications, interpretations, and relevance of these criteria for the design of a software engineering curriculum.

4.1.1. Purposes and Policies

Purposes. In its "Statement of Principles," ABET describes its purposes:

> The purposes of ABET shall be the promotion and advancement of engineering education with a view to furthering the public welfare through the development of the better educated and qualified engineer, engineering technologist, engineering technician and others engaged in engineering or engineering-related work.
>
> To achieve these purposes ABET shall:
>
> (1) Organize and carry out a comprehensive program of accreditation of pertinent curricula leading to degrees, and assist academic institutions in planning their educational programs.
>
> (2) Promote the intellectual development of those interested in engineering and engineering-related professions, and provide technical assistance to agencies having engineering-related regulatory authority applicable to accreditation.

It also states that one of the specific objectives of accreditation is "[t]o provide guidance for the improvement of existing programs in engineering education and for the development of future programs."

Program titles. Section II.A.10.c of ABET's criteria for accrediting programs in engineering states, "All engineering programs must include the word 'engineering' in

the program title." Section 7.2 of this report addresses the issue of possible names of undergraduate software engineering programs.

When to seek accreditation. The accreditation process requires an on-site visit by an ABET accreditation team. Section II.A.9 of the accreditation criteria states, in part, that it is a basic policy "[t]o grant initial accreditation only if students have graduated from a program prior to the on-site visit." This policy answers a question we have often heard, "Is it possible to create a new engineering program without ABET accreditation?" Clearly, it is not only possible, it is necessary.

Program level. ABET will accredit both bachelor's and master's degree programs. However, it will not accredit programs at both levels in the same discipline at the same institution (Section II.A.4). A number of schools have already established master's programs in software engineering (see [Ardis89] for descriptions of several of these), so it is possible for those who advocate ABET accreditation of software engineering programs to develop and propose accreditation criteria first for these programs.

Related programs. ABET also accredits engineering technology and engineering-related programs. Excerpts of ABET's definitions of such programs are quoted below:

> Engineering Technology is that part of the technological field which requires the application of scientific and engineering knowledge and methods combined with technical skills in support of engineering activities; it lies in the occupational spectrum between the craftsman and the engineer at the end of the spectrum closest to the engineer. ... Graduates of baccalaureate programs are called "engineering technologists."
>
> [Engineering-related programs] to be considered are conducted in the field of higher technical education, with close practical and academic ties with engineering. The programs do not fall under the strict engineering or engineering technology definitions. The mathematics, basic sciences, and humanities content of the engineering-related programs are similar to those contained in engineering and engineering technology programs; however, some of the engineering science and engineering design components contained in a typical engineering program are replaced by the engineering-related specialties. The programs should consist of a cohesive set of courses sequenced so that reasonable depth is obtained in the upper level courses. Certain programs may prepare graduates for practice at a professional level in an engineering-related specialty which cannot be classified as engineering or engineering technology. However, such programs derive their professional nature from specific professional-entry curriculum requirements imposed by the program criteria that a cognizant technical society has submitted to the ABET Board of Directors for approval.

We have heard suggestions among educators that software engineering, at least at its current state of development, might be better classified as an engineering technology or an engineering-related field. We have also heard predictions that as the

discipline matures, it will have both engineering and engineering technology components (see Section 6.3 of this report).

4.1.2. The Range of Engineering Disciplines

The ABET program accreditation criteria consist of common criteria that apply to all programs and separate criteria for programs in individual engineering disciplines. These latter criteria are developed by an appropriate professional society (or several societies, with one being designated as the *lead society*) and then reviewed and approved by ABET. Figure 4.1 shows the wide range of these disciplines and their professional societies.

The names of accredited programs vary even more widely. Recognizing this variety, the accreditation categories are described by ABET under headings such as "Program Criteria for Aerospace and Similarly Named Engineering Programs." In addition, ABET provides a category called "Nontraditional Programs" for programs that are not covered by specific program criteria developed by a professional society. Figure 4.2 gives examples of the names of several accredited programs and their categories. (Note that several of the program names seem to violate the ABET requirement that the word "engineering" appear in program names.)

The accredited programs in the computer engineering category are of particular interest to those concerned with the possibility of future accreditation of software engineering programs. First, notice that in this category are programs named computer science and engineering (at least eleven schools have such programs). We understand that ABET and the Computing Science Accreditation Board (CSAB) have recently agreed that such programs must, in the future, be accredited by both ABET and CSAB. This is consistent with ABET's rule that programs bearing the name of two engineering categories must be accredited in both.

Second, notice that the computer engineering category includes a program named computer science. An obvious question, in light of the cooperation between ABET and CSAB, is whether this program will continue to be accredited by ABET under this name.

The fact that there is such a variety of programs in the computer engineering category raises the possibility that future software engineering programs might be accredited according to the existing (or future) ABET criteria in this category. We will examine those criteria in the next two sections.

Program Name	Professional Society (* Lead Society)
Aerospace	American Institute of Aeronautics and Astronautics
Agricultural	American Society of Agricultural Engineers
Bioengineering	Institute of Electrical and Electronics Engineers*
Ceramics	National Institute of Ceramic Engineers
Chemical	American Institute of Chemical Engineers
Civil	American Society of Civil Engineers
Computer	Institute of Electrical and Electronics Engineers*
Construction	American Society of Civil Engineers
Electrical	Institute of Electrical and Electronics Engineers
Engineering Management	Institute of Industrial Engineers*
Engineering Mechanics	American Society of Mechanical Engineers*
Environmental, Sanitary	American Academy of Environmental Engineers*
Geological	Society of Mining Engineers of AIME
Industrial	Institute of Industrial Engineers
Manufacturing	Society of Manufacturing Engineers
Materials	Metallurgical Society*
Mechanical	American Society of Mechanical Engineers
Metallurgical	Metallurgical Society*
Mining	Society of Mining Engineers of AIME
Naval Architecture and Marine Engineering	Society of Naval Architects and Marine Engineers
Nuclear	American Nuclear Society
Ocean	Society of Naval Architects and Marine Engineers*
Petroleum	Society of Petroleum Engineers of AIME
Surveying	American Congress on Surveying and Mapping*

Figure 4.1. Engineering disciplines with ABET program criteria

Accreditation Category	Program Name
Aerospace	Aeronautical Engineering Aeronautics and Astronautics Astronautical Engineering
Agricultural	Biological Engineering Forest Engineering
Ceramic	Ceramic Science Glass Science
Computer	Computer and Electrical Engineering Computer and Information Engineering Sciences Computer and Systems Engineering Computer Science Computer Science and Engineering Computer Systems Engineering
Construction	Structural Engineering
Electrical	Electric Power Engineering Electrical Engineering and Computer Science Microelectronic Engineering
Environmental, Sanitary	Environmental Resources Engineering
Materials	Materials Science
Petroleum	Natural Gas Engineering
Nontraditional	Architectural Engineering Engineering and Public Policy Engineering Physics Fire Protection Engineering Fluid & Thermal Sciences Food Process Engineering Plastics Engineering Polymer Science Systems Analysis and Engineering Systems and Control Engineering Systems Engineering Textile Engineering Welding Engineering

Figure 4.2. Examples of names of ABET-accredited programs

4.1.3. General Criteria for Engineering Programs

The general criteria for engineering programs include sections on faculty, curricular objective and content, student body, administration, institutional facilities, and institutional commitment. We will look briefly at the curriculum content criteria in this section. We recommend that schools and individuals interested in engineering curricula look at the entire accreditation criteria document from ABET because it contains a wealth of other important information.

The curriculum content is described in several categories: mathematics, basic sciences, engineering sciences, engineering design, humanities and social sciences, laboratory experience, computer-based experience, written and oral communication, and the ethical, social, economic, and safety considerations of engineering practice. The minimum curriculum requirements are summarized in Figure 4.3, where the requirement is measured as a percentage of the entire program.

Requirement	ABET Content Category
25%	Mathematics and Basic Sciences
25%	Engineering Sciences
12.5%	Engineering Design
12.5%	Humanities, Social Sciences
25%	Electives

Figure 4.3. ABET curriculum content for engineering programs

The mathematics requirement includes differential and integral calculus and differential equations. Additional work is encouraged in probability and statistics, linear algebra, numerical analysis, and advanced calculus.

The basic sciences requirement is intended to give students fundamental knowledge about nature and its phenomena, including quantitative expression. It includes chemistry and physics, with a two-semester sequence in either area. Additional science may also be in the life sciences or earth sciences, as appropriate to a particular engineering discipline.

The engineering sciences are described as having their roots in mathematics and basic sciences but carrying knowledge further toward creative application. They provide a bridge between mathematics/basic sciences and engineering practice. Examples are mechanics, thermodynamics, electrical and electronic circuits, materials science, transport phenomena, and computer science (other than computer

programming skills). The requirement includes at least one engineering science course outside the major discipline area.

Engineering design lies at the heart of an engineering curriculum. ABET defines it in this way:

> Engineering design is the process of devising a system, component, or process to meet desired needs. It is a decision-making process (often iterative), in which the basic sciences, mathematics, and engineering sciences are applied to convert resources optimally to meet a stated objective. Among the fundamental elements of the design process are the establishment of objectives and criteria, synthesis, analysis, construction, testing, and evaluation. The engineering design component of a curriculum must include at least some of the following features: development of student creativity, use of open-ended problems, development and use of design methodology, formulation of design problem statements and specifications, consideration of alternative solutions, feasibility considerations, and detailed system descriptions. Further, it is essential to include a variety of realistic constraints such as economic factors, safety, reliability, aesthetics, ethics, and social impact.

The humanities and social sciences component of the curriculum must be designed not only to meet the general objectives of a broad education, but must also fulfill an objective appropriate to the engineering profession. The coursework must make students aware of their social responsibilities as engineers. It cannot be a selection of unrelated introductory courses.

The laboratory requirement normally includes both basic science and engineering design laboratories. These need not be separate courses, but may be integral parts of other courses.

The computer-based experience requirement includes the use of computers in support of engineering activities, such as technical calculation, problem solving, data acquisition and processing, process control, or computer-assisted design.

The requirements for communications skills and societal issues are normally distributed across the curriculum rather than being structured as independent courses.

4.1.4. Program Criteria for Computer and Similarly Named Engineering Programs

Program criteria for computer engineering were submitted by the Institute of Electrical and Electronics Engineers in cooperation with the Institute of Industrial Engineers. These criteria amplify, rather than supplant, the general program criteria described in the previous section.

Our research into ABET accreditation was based on a document published in 1988 [ABET88]. As might be expected, program criteria for any engineering program must evolve to keep pace with the advances in the discipline. The 1988 document

included proposed changes for the next edition, and it is these proposed criteria that are reflected in the discussion below.

The overall curriculum structure for a computer engineering program must provide breadth across the field of computer science and engineering, both hardware and software. Depth must be attained in at least one area of computer science and engineering.

The mathematics requirement includes discrete mathematics, probability and statistics, and either linear algebra or numerical methods.

The engineering science and design courses must provide a balanced view of hardware, software, application tradeoffs, and the basic modeling techniques used to represent the computing process.

A strong laboratory sequence of hardware and software development experiences must provide the student with an appropriate range of problem solving, design, implementation, documentation and oral presentation activities and the use of a variety of hardware and software tools.

4.1.5. General Criteria for Engineering-Related Programs

The definition of engineering-related programs appeared in Section 5.1.1 of this report. We include a brief discussion of the accreditation criteria for such programs because we have heard suggestions that software engineering might fit better in this classification than in engineering.

The overall curriculum requirements are shown in Figure 4.4.

Requirement	ABET Content Category
19%	Mathematics and Basic Sciences
38%	Engineering-Related Sciences and Engineering-Related Specialties
18%	Humanities, Social Sciences
25%	Electives

Figure 4.4. ABET curriculum content for engineering-related programs

The ABET descriptions of engineering-related sciences and engineering-related specialties are as follows:

> Engineering-related sciences have their roots in mathematics and basic sciences, but carry knowledge further toward creative application. When a field of mathematics or basic science proves pertinent to an engineering-related application, there develop corresponding courses in engineering-related science to afford a bridge between the basic science and engineering-related practice.
>
> The requirements for coursework in engineering-related specialties have been established in recognition of the need to reorient the student toward specialized practice in the engineering-related discipline. In specialized practice, the needs and problems of society are treated by innovative application of the technological foundations of mathematics, basic sciences, and engineering-related sciences to achieve viable solutions. Among the fundamental elements of the problem solving process are the establishment of objectives and criteria, synthesis, analysis, and evaluation. The specialized practice component of a curriculum should include some of the following features: development of student creativity, use of open-ended problems, formulation of problem statements and specifications, consideration of alternative solutions, feasibility considerations, and detailed solution descriptions. It is also important to include a variety of realistic constraints such as economic factors, safety, reliability, aesthetics, ethics, and social impact. Courses that include specialized practice may be included at all levels of the program. However, the major portion of the engineering-related specialties requirement is to be satisfied by courses that follow mathematics, basic sciences and engineering-related sciences.

4.2. CSAB Accreditation

The Computing Sciences Accreditation Board (CSAB) was formed as a corporation in the state of New York on January 8, 1985. Its creation was the result of more than two years of work by the Association for Computing Machinery (ACM) and the Computer Society of the Institute of Electrical and Electronics Engineers (IEEE-CS). A more detailed history of CSAB is presented in [Cain86]. CSAB's Computer Science Accreditation Commission (described below) is recognized by the United States Secretary of Education and by the Council on Postsecondary Accreditation as the nationally recognized agency for the evaluation and accreditation of baccalaureate programs in computer science. The material presented below is from [CSAB87], which is available from CSAB, 345 East 47th Street, New York, NY 10017; telephone (212) 705-7314.

4.2.1. Purpose and Policies

CSAB's constitution defines its purpose as "... to advance the development and practice of the computing sciences in the public interest through the enhancement of quality educational programs in the computing sciences. The term 'computing sciences' is defined to include the broad spectrum of computer disciplines."

The fact that the phrase "computing sciences" in CSAB's title is plural implies that there may be more than just "computer science" in the domain of accreditable programs. In fact, CSAB is structured to allow creation of accreditation commissions, each of which is responsible for developing accreditation criteria for a specific area of the computing sciences. The first (and currently only) accreditation commission is the Computer Science Accreditation Commission (CSAC). Its criteria for accreditation of computer science programs are discussed in the next section.

At least one other accreditation commission, for information systems, is being discussed [Gorgone89]. It is conceivable that an accreditation commission for software engineering could be established within CSAB, although we would expect the use of the word "engineering" in its title to be controversial, given the current cooperative understandings between CSAB and ABET. On the other hand, an accreditation commission within CSAB, perhaps with a more neutral name such as "software systems," may be an appropriate mechanism for accrediting programs in colleges and universities that do not have engineering schools. These issues deserve additional consideration and discussion.

The basic policies of CSAB are very similar to those of ABET. In fact, the wording of many policies is identical for both organizations. Of interest to developers of new programs is the policy to grant accreditation only if students have already been graduated. Also, like ABET, CSAB has a policy "[t]o avoid applying minimum standards in a way that would discourage well-planned experimentation."

4.2.2. Program Accreditation Criteria

Despite the short history of the CSAB accreditation criteria for computer science programs, there has already been one major revision. This was partly a result of concern expressed during and after the first cycle of accreditation visits that the criteria required too many technical courses for a liberal arts curriculum (see [Gibbs86] for a discussion of computer science curricula for liberal arts colleges). The most significant changes in the revised criteria are a reduction in computer science content from one and one-half years to one and one-third years and a slight reduction in the overall mathematics and science requirements.

The curriculum criteria are summarized in Figure 4.5. The requirements are stated in [CSAB87] in a combination of measures (semester courses, years); so to determine the percentage requirements, we assume a total requirement of 120 semester hours[1], with each course being 3 semester hours.

[1] Note for readers not familiar with United States universities: A *semester hour* represents one contact hour (usually lecture) and two to three hours of outside work by the student per week for a semester of about fifteen weeks. A *course* covers a single subject area of a discipline and typically meets three hours per week, for which the student earns three semester hours of credit. A course with a laboratory component might give four semesters hours of credit.

Requirement	Percentage	CSAB Content Category
0.5 year	12.5%	Mathematics
2 courses	5%	Laboratory Science Sequence
2 courses	5%	Science or Quantitative Methods
1.33 years	33.3%	Computer Science
40-60% of CS	13-20%	Core
60-40% of CS	20-13%	Advanced
1 year	25%	Humanities, Social Sciences, Arts
1 course	2.5%	Other Required Course
0.67 year	16.7%	Free Electives

Figure 4.5. CSAB curriculum content for computer science programs

The computer science core requirement specifies a "reasonably even emphasis over the areas of theoretical foundations of computer science, programming languages, and computer elements and architecture. Within this portion of the program, analysis and design experiences with substantial laboratory work, including software development, should be stressed." The advanced computer science courses should "insure that depth of knowledge is obtained in at least one-half of the core material."

The mathematics requirement includes discrete mathematics, differential and integral calculus, and probability and statistics. The science requirement includes a two-semester sequence in a laboratory science, plus two additional courses. These may be either science courses or courses with strong emphasis on quantitative methods. Oral and written communication skills are also required, although not necessarily through a separate course.

4.3. Accreditation Issues Related to Faculty

Both ABET and CSAB specify other accreditation criteria that may affect the development of undergraduate software engineering programs. In particular, both address the issue of the number and competence of the faculty.

ABET's general criteria suggest that three full-time faculty members are necessary for a minimal program; the criteria for computer engineering and similarly named programs say that five full-time faculty are necessary. The criteria for nontraditional engineering programs state [ABET88]:

> In small institutions with strong departments of basic science and no other engineering programs, at least four faculty members educated as engineers or with extensive engineering experience are necessary to provide the engineering philosophy and application in the program.

CSAB criteria suggest a minimum of five full-time-equivalent faculty, of which four should be full-time faculty with primary commitment to the program. Full-time faculty should cover at least 70% of the total classroom instruction.

Professional competence of faculty is addressed by both ABET and CSAB. In addition to the expected requirements that most faculty hold the terminal degree and that they pursue scholarly activities, ABET also mentions the desirability of their being licensed as professional engineers.

These requirements will be impediments to the development of new programs. As far as we know, no United States university offers a doctorate in software engineering, and the number of computer science doctoral students doing research in software engineering is still relatively small. Software engineering professionals with terminal engineering degrees are very much in demand in industry, so it is difficult to recruit them for faculty positions.

What constitutes scholarly activity in software engineering is widely debated. Among funding agencies, software engineering has not yet achieved the status or level of support as have many other engineering disciplines and computer science. Furthermore, empirical research will probably require substantial participation by the software industry, a large segment of which considers its knowledge of software engineering to be proprietary. Much work remains to be done to develop a national infrastructure in support of software engineering research and other scholarly activity.

Given our recent experience with a prolonged shortage of good computer science faculty members as computer science emerged as a discipline, it is not surprising that software engineering faculty are difficult to obtain. The most feasible short-term solution may be to convert computer science faculty into software engineering faculty. Therefore, a school considering the development of an undergraduate software engineering program in the future should begin investing in faculty development now.

5. Professional and Licensing Issues

5.1. Is Software Engineering a Profession?

In the United States, engineering is generally considered a profession subject to a variety of standards and regulations. Whether software engineering can or should eventually be included in the profession has been a topic of much discussion.

The U. S. Government addressed this issue in the 1984 Code of Federal Regulations ([US84], Vol. 29, §541.302 (h)):

> The question arises whether computer programmers and systems analysts in the data processing field are included in the learned professions. At the present time there is too great a variation in standards and academic requirements to conclude that employees employed in such occupations are a part of a true profession recognized as such by the academic community with universally accepted standards for employment in the field.

David Lamb addresses the question "What is a profession?" in [Lamb88]. He considers the classic professions (of perhaps 200 years ago) of doctor, lawyer, and priest. Four key characteristics of these professions were:

1. Extensive schooling to master a body of specialized knowledge
2. A period of apprenticeship
3. A restricted title or license to practice
4. A self-governing professional organization with the power to impose sanctions against unethical or incompetent members

He suggests that today the two key characteristics of a professional are competence and individual responsibility. The report elaborates these characteristics in the context of software engineering, including issues related to education.

[Barnes88] argues that computer science is a profession and describes the attributes of a professional. Much of the discussion can be applied to software engineering.

If we accept the premise that software engineering is or will become a profession, then it is appropriate to consider professional certification and licensing of software engineers. Informally, we describe certification as a voluntary practice administered by the profession itself, and licensing as a mandatory practice administered by government. We examine both of these practices in this chapter.

5.2. Certification of Software Professionals

Licensing or certification of software professionals is a topic being widely debated. We have heard many suggestions that software engineering cannot be considered an engineering profession until its practitioners are licensed like other engineers.

In [Preiss89], the IEEE Computer Society Committee on Public Policy (COPP) reports, "COPP believes that certification of developers of critical-mission software needs a penetrating review and a IEEE Computer Society position." The report also states that "the Computer Society Board of Governors has a standing position, since November 1982, opposing any action by the Institute for Certification of Computer Professionals (ICCP) to establish a certification program for software engineering." We have heard informally that ICCP is now actively developing just such a program.

The Information Systems Security Association believes that one branch of the software profession, information systems security, has matured sufficiently to warrant a certification program. A consortium of organizations is currently developing such a program. They state that "any program to license professionals must include a code of ethics, codes of conduct and good practice, a defined body of knowledge, a uniform examination and certification, an apprenticeship or intern program, an accredited higher education program, a continuing education requirement and/or a recertification procedure, an oversight by society (namely laws), and an image in the mind of the public." [Preiss89]

5.3. Licensing of Engineers

For most computer science educators and students, professional licensing of engineers is not a familiar concept. To help identify and elucidate some of the issues, we examined the Pennsylvania law regarding the licensing of engineers. It is our understanding that the laws of most of the states are similar.

5.3.1. Definitions

The Pennsylvania Professional Engineers Registration Law [Pennsylvania84] includes these definitions:

> "Practice of Engineering" shall mean the application of the mathematical and physical sciences for the design of public or private buildings, structures, machines, equipment, processes, works or engineering systems, and the consultation, investigation, evaluation, engineering surveys, planning and inspection in connection therewith, the performance of the foregoing acts and services being prohibited to persons who are not licensed under this act as professional engineers unless exempt under other provisions of this act. [§2.(a)(1)]

> The term "Practice of Engineering" shall also mean and include related acts and services that may be performed by other qualified persons, including but not limited to, municipal planning, incidental landscape architecture, teaching, construction, maintenance and research but licensure under this act to engage in or perform any such related acts and services shall not be required. [§2.(a)(2)]
>
> The "Practice of Engineering" shall not preclude the practice of other sciences which shall include but not limited to: soil science, geology, physics and chemistry. [§2.(a)(4)]

It is certainly possible to interpret the definition in §2.(a)(1) to include the development of software systems, especially embedded systems. Many techniques of software engineering are applications of mathematical sciences. Algorithms and their implementations could be considered processes, and embedded systems can be considered engineering systems. We are unaware, however, of any case law that establishes precedents regarding the inclusion of software engineering under this act.

5.3.2. Motivation for Licensing

The motivation for professional licensing of engineers is contained in this section of the law:

> In order to safeguard life, health or property and to promote the general welfare, it is unlawful for a person to practice or to offer to practice engineering in this Commonwealth, unless he is licensed and registered under the laws of this Commonwealth as a professional engineer, or for any person to practice or to offer to practice land surveying, unless he is licensed and registered under the laws of this Commonwealth as a professional land surveyor. [§3.(a)]
>
> A person shall be construed to practice or offer to practice engineering or land surveying who practices any branch of the profession of engineering or land surveying, or who, by verbal claim, sign, advertisement, letterhead, card, or in any other way represents himself to be an engineer or land surveyor, or through the use of some other title implies that he is an engineer or land surveyor or that he is registered under this act; or who holds himself out as able to perform, or who does perform any engineering service or work or any other service designated by the practitioner or recognized as engineering or land surveying. [§3.(b)]

The pervasiveness of software-based systems in our society has led to a number of instances of threats to life, health, or property because of software errors. *Software Engineering Notes*, the newsletter of the ACM Special Interest Group on Software Engineering, publishes a column in each issue titled "Risks to the Public in Computers and Related Systems," which contains reports of apparent or proven computer-related risks. Some of the reports describe loss of life or loss of significant amounts of property. As such incidents grow in number and severity, we believe it is

likely that society, as embodied in its legislative and judicial systems, will attempt to regulate the developers of software-based systems.

5.3.3. Licensing Requirements

The section [§4.(b)] in the Pennsylvania law that defines licensing requirements begins:

> The [State Registration Board for Professional Engineers] shall have power– …
>
> **Licensing Professional Engineers.**—To provide for and to regulate the licensing, and to license to engage in the practice of engineering any person of good character and repute who is at least in his twenty-fifth year of age, and who speaks and writes the English language, if such person either– …

The law then specifies four alternative sets of requirements for licensing. The first two are reciprocal agreements based on a person having been licensed by another recognized licensing authority (such as another state or country). The third set of requirements applies to persons with engineering degrees (*§4.(b)(3) below*) and the fourth to persons without such degrees (*§4.(b)(4) below*):

> Has had four or more years' progressive experience in engineering work, under the supervision of a professional engineer or a similarly qualified engineer, of a grade or character to fit him to assume responsible charge of the work involved in the practice of engineering, and is either an engineer-in-training or a graduate in engineering of an approved institution or college having a course in engineering of four or more years, or has had four or more years of progressive experience in engineering work, teaching in an approved institution or college, and who is a graduate of an approved institution or college having a course in engineering of four or more years and who in either event successfully passes written examinations prescribed by the board in engineering subjects. In the case of the examination of an engineer-in-training his examination shall be directed and limited to those matters which will test the applicant's ability to apply the principles of engineering to the actual practice of engineering. In the case of an applicant who is not an engineer-in-training the examinations will be for the purpose of testing the applicant's knowledge of fundamental engineering subjects, including mathematics and the physical sciences and those matters which will test the applicant's ability to apply the principles of engineering to the actual practice of engineering. … [§4.(b)(3)]
>
> Has had twelve or more years of progressive experience in engineering work, at least eight years of which shall have been under the supervision of a professional engineer or similarly qualified engineer, of a grade and character to fit him to assume responsible charge of the work involved in the practice of engineering, and who successfully passes written examinations prescribed by the board for the purpose of testing the applicant's knowledge of fundamental engineering subjects, including mathematics and the physical sciences and those matters which will test the applicant's ability to apply the principles of

> engineering to the actual practice of engineering. To be licensed under this subsection, the person shall be required to successfully pass the examinations prescribed by the board for both professional engineers and engineers-in-training. [§4.(b)(4)]

The typical procedure for students completing an undergraduate engineering degree is to take the state engineer-in-training examination during their senior year. Most students find that this material is freshest in their minds during their senior year rather than after a few years of professional experience. The law defines an engineer-in-training as follows:

> "Engineer-in-Training" means a candidate for licensure as a professional engineer, who has been granted a certificate as an engineer-in-training after successfully passing the prescribed written examination in fundamental engineering subjects, and who shall be eligible upon completion of the requisite years of experience in engineering, under the supervision of a profession engineer, or similarly qualified engineer, for the final examination prescribed for licensure as a professional engineer. [§2.(c)]

The state board examines and certifies engineers-in-training, as prescribed in the law as follows:

> **Examination and Certification of Engineers-in-Training.**–To provide for and to regulate the examination of any person who has produced satisfactory evidence that he has graduated in an engineering curriculum from an approved institution or college having a course of four years or more in engineering or who has had four or more years' experience in engineering work, and who produces satisfactory evidence to show knowledge, skill and education approximating that attained through graduation from an approved institution or college, and to issue to any such person who successfully passes such examination a certificate showing that he has successfully passed this portion of the professional examination and is recognized as an engineer-in-training. The examination of applicants as engineers-in-training shall be designed to permit an applicant for licensure as a professional engineer to take his examination in two stages. The examination for certification as an engineer-in-training shall be for the purpose of testing the applicant's knowledge of fundamental engineering subjects, including mathematics and the physical sciences. Satisfactory passing of this portion of the examination shall constitute a credit for the life of the applicant or until he is licensed under this act as a professional engineer. [§4.(c)]

5.3.4. Code of Ethics

As is the case with most professions, the engineering profession has a code of ethics. Persons seeking licensing as professional engineers in Pennsylvania are required to affirm that they subscribe to and agree to abide by the code of ethics specified in §4.(i) of the law:

It shall be considered unprofessional and inconsistent with honorable and dignified bearing for an professional engineer or professional land surveyor:

1. To act for his client or employer in professional matters otherwise than as a faithful agent or trustee, or to accept any remuneration other than his stated recompense for services rendered.
2. To attempt to injure falsely or maliciously, directly or indirectly, the professional reputation, prospects or business of anyone.
3. To attempt to supplant another engineer or land surveyor after definite steps have been taken toward his employment.
4. To compete with another engineer or land surveyor for employment by the use of unethical practices.
5. To review work of another engineer or land surveyor for the same client, except with the knowledge of such engineer or land surveyor, or unless the connection of such engineer or land surveyor with the work has terminated.
6. To attempt to obtain or render technical services or assistance without fair and just compensation commensurate with the services rendered: Provided, however, the donation of services to a civic, charitable, religious or eleemosynary organization shall not be deemed a violation.
7. To advertise in self-laudatory language, or in any other manner, derogatory to the dignity of the profession.
8. To attempt to practice in any other field of engineering in which the registrant is not proficient.
9. To use or permit the use of his professional seal on work over which he was not in responsible charge.
10. To aid or abet any person in the practice of engineering or land surveying not in accordance with the provisions of this act or prior laws.

Ethics has been a normal part of engineering curricula for many years, and we believe it will be an important pervasive theme in software engineering curricula. It is less common in computer science curricula, so issues of professional ethics in general and software engineering ethics in particular are unfamiliar to most students and educators in computer science. We hope that this situation will improve over the next several years.

6. Strategies for Undergraduate Software Engineering Education

In the previous chapters we have presented a variety of topics that might be considered background or context for discussions of undergraduate software engineering education. We now turn our attention to the development of appropriate educational opportunities.

Even among those who accept the basic premise that better undergraduate education for software professionals is needed, there is disagreement on the best ways to provide that education. Approaches include complete programs in software engineering, adding one or more courses to existing computer science curricula, and adding software engineering topics to existing courses in computer science. A sample of the varied opinions can be found in the position papers of the participants in the 1989 SEI Workshop on an Undergraduate Software Engineering Curriculum (see Appendix 1 and [Gibbs89b]).

We believe that both separate programs in software engineering and software engineering tracks in computer science programs are inevitable. In this chapter we examine strategies for the development of such programs and tracks.

6.1. Program Development Strategies

Two competing strategies for the development of an undergraduate software engineering program are immediately evident:

Creation: design an entire curriculum and install it as a new degree program all at once.

Evolution: build the curriculum over a period of years within an existing degree program.

Both strategies have advantages and disadvantages. To examine these, let us assume that the evolution strategy will be applied to an existing computer science program rather than an engineering program.

The individual courses in a software engineering program are expected to be very different from those in the computer science program. Many of the same topics will be taught, but with different objectives (engineering vs. science) and different combinations of topics making up the courses. The creation strategy allows the new courses to be designed in one major effort, whereas the evolution strategy will almost certainly require redesign of many courses each semester or year over a period of several years. On the other hand, installing all new courses quickly is a

burden for faculty and students, since textbooks and teaching materials will not be readily available for all the courses.

Creation of a new program may require a substantial increase in resources, while an evolutionary approach allows more time for acquiring or shifting resources. In some cases, however, new resources may be available to make creation feasible. A number of major companies that employ large numbers of software engineers have expressed interest in working with their local universities to develop software engineering courses and programs. Federal government agencies such as the National Science Foundation (NSF) are considering increased funding for support of science and engineering education at the undergraduate level. A recent Congressional study urges NSF specifically to increase support for software engineering education, including the creation of pilot degree programs in several universities [Congress89].

We believe that the most practical strategy for most schools will be evolutionary development over a period of three to five years, and that the curriculum will exist for some time as a track within an existing computer science program. This strategy will tend to minimize the problem of availability of resources (and the program name problem described in the next section).

Evolution of the program can be bottom-up or top-down. The bottom-up approach introduces the new courses first at the freshman level, with other new courses being introduced as the first students in the program proceed through their four years. This approach requires a risky commitment on the part of the school, faculty, and students, in that it is difficult to "bail out" if problems are encountered. Students may find it difficult to switch to the existing computer science curriculum in later years because they may not have the proper prerequisites, or because the computer science courses may overlap the software engineering courses too much for the students to receive credit for both, but not enough to permit the students to skip those courses. An advantage of the bottom-up approach is that there is more time available to the faculty to develop the more advanced courses and to adjust the designs of those courses based on experiences with the preceding courses.

The top-down approach introduces new courses first at the senior level, with other new courses brought in later at increasingly lower levels. The beginning stages of this approach are already visible at a number of schools that have introduced one-semester and then two-semester courses in software engineering at the senior level. [Tomayko87, Richardson88, Northrop89]. This approach is somewhat less risky, but it does require continuing development of the higher level courses as the new and presumably better prerequisite courses are introduced.

Regardless of the approach, some of the problems can be lessened by innovative or nontraditional educational techniques. For example, many new courses can be expected to combine topics from existing courses in new ways. Team teaching allows

faculty members most familiar with a topic to present it, thereby decreasing the preparation time of all faculty.

6.2. On the Name "Software Engineering"

A surprisingly significant problem that schools will face when developing a software engineering program is choosing a name for the program. This section discusses that problem and possible solutions.

The computing disciplines have suffered a kind of identity crisis throughout their short history. The identity problem is reflected today in the names of academic departments and programs, as shown in Figure 6.1, which lists the names of departments granting PhD degrees in computing [Gries89].

The introduction of the term "software engineering" has further complicated matters. The term apparently first became widely known as a result of a 1968 conference in Garmisch, Germany, sponsored by the NATO Science Committee [Naur69]. Since that time it has become widely used, but without any real consensus on its meaning. Several definitions that have appeared in the literature are presented in Chapter 3.

University graduate programs in software engineering have existed in the United States for more than 10 years. Not surprisingly, these programs have many different names, as shown in Figure 6.2.

It is in the context of undergraduate software engineering education, however, that the name "software engineering" has been most controversial. No other issue has produced more discussion with fewer meaningful results than that of calling an undergraduate program "software engineering." The discussion seems to be centered on the question, "Is software engineering really engineering?"

The arguments in support of a negative answer to that question are usually in one of two categories. The first includes legalistic arguments that appeal to "formal" definitions of engineering, such as those in a dictionary, in the charter of a professional society, or in the guidelines for accrediting engineering programs or licensing engineers. Such definitions usually evoke the notion of tangible products derived from effective use of the materials and forces of nature. For example, the Accreditation Board for Engineering and Technology provides this definition [ABET88]:

> Engineering is that profession in which knowledge of the mathematical and natural sciences gained by study, experience, and practice is applied with judgment to develop ways to utilize, economically, the materials and forces of nature for the benefit of mankind.

Number of Departments	Department Name
89	Computer Science(s)
22	Electrical and Computer Engineering
11	Computer Science and Engineering
10	Computer and Information Science(s)
7	Electrical Engineering and Computer Science
3	Computer Engineering
2	Computing Science
2	Electrical Engineering
2	Information and Computer Science
1	Advanced Computer Studies
1	Applied Sciences
1	Computational Science
1	Computer Engineering and Science
1	Computer Science and Electrical Engineering
1	Computer Science and Operations Research
1	Electrical, Computer, and Biomedical Engineering
1	Mathematical and Computer Sciences
1	Mathematical Sciences

Figure 6.1. Names of PhD-granting academic departments

Software, it is argued, uses neither the materials nor the forces of nature. However, some engineering disciplines, such as industrial engineering, are very much concerned with the design of processes, which are much closer to software in their intangibility.

The second category of argument is based on the idea that an engineering discipline evolves from a craft, and the software craft has not yet evolved sufficiently far to be called engineering. (Perhaps the definitive discussion of the history of engineering and its application to software engineering is [Shaw89]; we highly recommend it to software engineering educators and students.)

Because definitions of engineering are not mathematically precise, it is not possible to construct a "proof" that software engineering is or is not engineering. Similarly, the boundary between craft and engineering is not clearly drawn, so it is not possible to observe the crossing of that boundary.

Program Title	University
Master of Software Engineering	Carnegie Mellon University Seattle University
Master of Science in Software Engineering	Andrews University Monmouth College University of Pittsburgh The Wichita State University
Master of Science in Software Engineering Sciences	University of Houston-Clear Lake
Master of Computer Science in Software Engineering	The Wichita State University
Master of Science in Software Systems Engineering	Boston University George Mason University
Master of Software Design and Development	College of St. Thomas Texas Christian University
Master of Science in Software Development and Management	Rochester Institute of Technology

Figure 6.2. Software engineering degree program titles

Arguments in support of a positive answer to the question "Is software engineering really engineering?" have appeared in the literature for many years. The first we have found was published in 1969 (!), and the author specifically calls for the establishment of software engineering curricula [Kuo69]. Another early argument is [Jeffery77], and a reasoned argument may also be found in the first chapter of [Jensen79a]. More recently, this question has been addressed in the narrower area of information systems engineering in [Lewis89] and [Nash89].

There are practical implications of this issue for developers of undergraduate software engineering programs. The word "engineering" may be unacceptable in the title of a course or program in a college or university that does not have an engineering school. In some states, the state licensing board for engineers may have the power, directly or indirectly, to limit the use of the word to those disciplines where licensing is available. Furthermore, some schools believe that all undergraduate engineering programs should be accredited, and it is unlikely that either the Accreditation Board for Engineering and Technology (ABET) or the Computing Science Accreditation Board (CSAB) will accredit a program titled "software engineering" any time soon. (Accreditation issues are addressed in more depth in Chapter 4 of this report.)

Furthermore, programs that evolve in computer science departments at schools where the department is outside the engineering school will almost certainly encounter objections to the use of the name software engineering. In such cases it may be possible for the degree to be granted through the engineering school, even though the faculty and administration of the degree program reside in a department outside that school. (The author of this report was involved in the development of computer science programs in the reverse situation: the computer science department was in the college of engineering, but it offered a bachelor of science degree in computer science through the college of liberal arts.)

We believe that the *name* of a potential undergraduate software engineering program is almost insignificant, whereas the *content* of the program is critically important. We expect that as such programs are developed, they will have a variety of names. The majority are likely to be new tracks within existing computer science or computer engineering programs, and thus will officially bear the same name as the existing program.

For schools that want to avoid being bogged down by the "Is it engineering?" and "Is it science?" questions, we suggest a neutral term: *software systems*. At some point in the future, it may be acceptable and desirable to rename the program software engineering. (Note that the University of Houston-Clear Lake avoids the problem in the opposite way; it calls its graduate program *software engineering sciences*.)

6.3. Speculation on Future Trends

There has been much speculation on the future development of the software engineering discipline and the implications for software engineering education. We believe that many of these ideas should be considered in any discussion of the development of undergraduate software engineering education.

Some members of the software engineering community believe that the discipline will partition itself into levels of skill, with different levels requiring different educational backgrounds. They cite the medical profession as a possible model. That profession includes physicians, nurses, physician's assistants, and paramedics, all of whom have reasonably well-defined roles. David Lamb suggests that we should consider a model such as electrical engineering, with a software engineer analogous to an electrical engineer, a programmer to a technician, and a computer scientist to a physicist [Lamb88]. Al Pietrasanta suggests a similar future, with a small number of "super-professionals" and a large number of supporting technicians [Gibbs87, p. 418].

Others see a development somewhat like that of computer science in the 1960s and 1970s. Many schools first offered a computer science degree at the master's level. These degrees were in great demand because many people with degrees in other

areas found themselves being changed into programmers. Additional education was needed, and a second bachelor's degree was not an acceptable approach. As a result, many master's programs contained little more than repackaged undergraduate computer science. In some schools, undergraduate computer science majors were not allowed into their school's graduate program because they already knew all the material.

As we develop a better understanding of the content of software engineering education, we are likely to see the current master's programs in a similar light. We may therefore expect most of today's graduate material to be brought down to the undergraduate level in the next several years. In [Tomayko89], Jim Tomayko discusses this issue and concludes, "[T]he actual material taught in graduate software engineering courses can be easily understood by undergraduates."

Furthermore, there seems to be increasing belief in the software engineering community that domain-specific knowledge must become a part of software engineering education. It is possible that as the generic software engineering concepts are moved to the undergraduate level, master's programs will begin to offer substantial opportunities for domain specialization. For example, master's students may devote at least half of their studies to areas such as real-time embedded systems, commercial systems, decision support systems, or expert systems.

Another concern affects all engineering education: the accelerating growth of knowledge in the sciences and engineering may place impossible demands on students to learn an appropriate part of the knowledge in a four-year baccalaureate program. The president of the Accreditation Board for Engineering and Technology, in [ABET88], identified one of the challenges for the year to be "to study the feasibility of advanced- or dual-level accreditation to address the concern that four-year bachelor's degrees are no longer sufficient for today's world of stiff international competition." The dean of engineering at MIT, in [Wilson89], says, "One of the problems we face is that we cannot do the things we want to do in engineering education in four years. ... What we should be saying to students is that your undergraduate degree is not enough; after you have some experience, you should go back to school. And we should be telling industry that it is in their best interest to allow young engineers to go back to school for a master's degree and to support them while they do so." National studies, such as [NSB86] and [NRC85], discuss this problem and conclude that for some fields, including computer engineering, a master's degree should be the minimum entrance requirement for the profession.

We believe that all these issues must be considered as undergraduate software engineering evolves. The challenge to provide high-quality education for a skilled software work force can be met only by providing appropriate education at a variety of levels for a variety of practitioners.

7. Designing an Undergraduate Curriculum

A school that wants to implement a new undergraduate program in software engineering obviously must design a curriculum. Even when a school plans a long-term evolutionary approach to introducing software engineering into the undergraduate curriculum, a design of the end product of that evolution is desirable. In this chapter we discuss a variety of ideas, concepts, and constraints that might affect such a design.

7.1. Curriculum Objectives

There are many opinions on the purposes and objectives of undergraduate education, ranging along a spectrum from broad, general education to focused vocational, preprofessional, or professional skills. In this section we will examine the professional objectives of a potential software engineering curriculum, but we do *not* wish to imply a lack of support for a broadly based undergraduate program.

7.1.1. Some Ideas from the Literature

In the spring of 1985, the ACM, with the cooperation of the IEEE Computer Society appointed a task force on the core of computer science. Subsequently, the two societies formed the Joint Task Force on Undergraduate Curricula in Computer Science and Engineering to develop guidelines for a common curriculum for all computing programs. Although the latter task force has not yet issued its report, the report of the former [Denning89] contains some general objectives for computing programs that we believe apply to software engineering programs as well.

The report defines *discipline-oriented thinking* as "the ability to invent new distinctions in the field, leading to new modes of action and new tools that make those distinctions available for others to use." It goes on to say, "We suggest that discipline-oriented thinking is the primary goal of a curriculum for computing majors ..." and "Discipline-oriented thinking must be based on solid mathematical foundations, yet theory is not an integral part of most computing curricula." The report also states, "The standard practices of the computing field include setting up and conducting experiments, contributing to team projects, and interacting with other disciplines to support their interests in effective use of computing, but most curricula neglect laboratory exercises, team projects, or interdisciplinary studies."

The report concludes its discussion of curriculum objectives with these statements: "The question of what results should be achieved by computing curricula has not been explored thoroughly in past discussions, and we will not attempt a thorough analysis here. We do strongly recommend that this question be among the first

considered in the design of new core curricula for computing." We believe it should also be the first question considered in the design of software engineering curricula.

ABET gives these overall objectives for an engineering program (including, we believe, a software engineering program):

> Engineering is that profession in which knowledge of the mathematical and natural sciences gained by study, experience, and practice is applied with judgment to develop ways to utilize, economically, the materials and forces of nature for the benefit of mankind. A significant measure of an engineering education is the degree to which it has prepared the graduate to pursue a productive engineering career that is characterized by continued professional growth. ...
>
> Included are the development of: (1) a capability to delineate and solve in a practical way the problems of society that are susceptible to engineering treatment, (2) a sensitivity to the socially-related technical problems which confront the profession, (3) an understanding of the ethical characteristics of the engineering profession and practice, (4) an understanding of the engineer's responsibility to protect both occupational and public health and safety, and (5) an ability to maintain professional competency through life-long learning. [ABET88]

Gerald Wilson, dean of engineering at MIT, suggests that the time is right for the teaching of engineering to be revitalized:

> An understanding of more than one engineering specialty and other undergraduate experiences that prepare students to work in interdisciplinary teams; more design work; and broad exposure to the economic, political, and social issues involved in large engineering projects–these are some of the elements of a refocused engineering education. [Wilson89]

Samuel Florman also argues for increasing the breadth of engineering curricula and discusses some of the history of engineering education that has led to the current state of technology-intensive curricula [Florman86].

Perhaps the best general description of curriculum objectives are implicit in the definition of software engineering in [BCS89] (see Chapter 2 of this report). Explicit objectives for a master's program in software engineering appear in [Ardis89]; these objectives are potentially relevant because of the expectation that much of the material in a graduate curriculum will migrate to undergraduate curricula.

A somewhat different perspective may be found in [Friedman89b], which reports the results of a survey of 100 computer center managers of Fortune 500 companies. These managers were asked about the most important requirements for new graduates entering the computer industry. The most often mentioned requirements, in decreasing order of frequency of mention, included "practical experience; courses strong in analytical, statistical, mathematical, logical skills; a degree in computer science helps, but is *not* that important; strength in oral and written communication skills; good business background; familiarity with a few different programming

languages." Knowledge of data structures was mentioned by only 8% of the respondents.

7.1.2. Describing Educational Objectives

Educational objectives are to curriculum design what software requirements are to software design. We hope that software engineering curriculum designers appreciate this analogy and will devote sufficient energy to defining good objectives before embarking on a major design effort.

A taxonomy of educational objectives that we have found particularly useful in our work in designing a graduate curriculum appears in [Bloom56]. An adaptation of this taxonomy for software engineering education appears in Appendix 2 of this report and is elaborated in [Ford87] and [Ardis89]. Briefly, the taxonomy is a hierarchy of increasingly difficult levels of achievement: knowledge, comprehension, application, analysis, synthesis, and evaluation.

To help illustrate another dimension of the problem of clearly stating objectives, consider the teaching of differential equations. Many, if not most, universities offer different courses in differential equations for students in different programs. In one case familiar to us, the course for pure mathematics majors spent virtually the entire semester proving the uniqueness and existence theorems for differential equations. In the course for applied mathematics and engineering majors, on almost the first day of class the proofs of those theorems were accepted as given, and the rest of the semester was devoted to techniques for solving differential equations and applying them to the solution of common kinds of problems. In both cases students achieved comprehension, application, analysis, and synthesis objectives, but they worked toward different overall goals: mathematics vs. engineering.

Because of these differences, it is not sufficient for an educational objective to state simply "differential equations at the analysis level" nor is it sufficient for a curriculum design to say "a one semester course in differential equations." More specific objectives will be needed. Clearly, this requires substantial effort on the part of the curriculum designer. However, just as for software design, starting with a complete, consistent, and clear requirements specification saves even more effort later and improves the probability that the resulting design really meets the user's needs.

7.1.3. Goals and Objectives

At the highest level, the goals of an undergraduate software engineering curriculum include:

- Preparing students for lifelong learning.

- Making students capable of contributing to an increasingly technological society; such as understanding enough about science and technology to make appropriate political decisions.
- Developing the students' communication and critical reasoning skills.
- Giving students an appropriate set of professional or preprofessional skills.

In this section we consider in detail the last of these four goals. In particular, we attempt to identify professional education objectives of a curriculum designed for students who do not pursue an advanced degree.

One set of objectives for software engineering professional education is implicit in the definition of software engineering in [BCS89]. This definition is reproduced in Chapter 2 of this report.

A second set of objectives can be derived from the objectives for a Master of Software Engineering curriculum in [Ardis89]. These new objectives are presented below in categories based on Bloom's taxonomy[2] [Bloom56]:

Knowledge: In addition to knowledge about all the material described in the subsequent paragraphs, students should be aware of the existence of models, representations, methods, and tools other than those they learn to use in their own studies. Students should be aware that there is always more to learn and that they will encounter more in their professional careers, whatever they may have learned in school.

Comprehension: Students should understand the differences between science and engineering along with the fundamental paradigms of each. They should understand the software engineering process, both in the sense of abstract models and in the various instances of the process as practiced in industry. They should understand the activities and aspects of the process. They should understand the issues (sometimes called the *software crisis*) that are motivating the growth and evolution of the software engineering discipline. They should understand the differences between academic or personal programming and software engineering; in particular, they should understand that software engineering involves the production of software systems under the constraints of control and management activities. They should understand a reasonable set of principles, models, representations, methods, and tools, and the role of analysis and evaluation in software engineering. They should understand the architectures of many common and well-understood classes of software systems. They should know of the existence and comprehend the content of appropriate standards. They should understand the fundamental economic, legal, and ethical issues of software engineering.

[2] See Appendix 2 for a brief description of this taxonomy; see [Ardis89] for definitions of the terms *activity*, *aspect*, and *product* as they are used here.

Application: Students should be able to apply fundamental principles in the performance of the various activities. They should be able to apply appropriate formal methods to achieve results. They should be able to use appropriate tools covering all activities of the software process. They should be able to collect appropriate data for project management purposes, and for analysis and evaluation of both the process and the product. They should be able to execute a plan, such as a test plan, a quality assurance plan, or a configuration management plan; this includes the performance of various kinds of software tests. They should be able to apply documentation standards in the production of all kinds of documents.

Analysis: Students should be able to participate in technical reviews and inspections of various software work products, including documents, plans, designs, and code. They should be able to analyze the needs of customers.

Synthesis: Students should be able to perform the activities leading to various software work products, including requirements specifications, designs, code, and documentation. They should be able to develop plans, such as project plans, quality assurance plans, test plans, and configuration management plans. They should be able to design data for and structures of software tests. They should be able to prepare oral presentations, and to plan and lead software technical reviews and inspections.

Evaluation: Students should be able to evaluate software work products for conformance to standards. They should know appropriate qualitative and quantitative measures of software products, and be able to use those measures in evaluation of products, as in the evaluation of requirements specifications for consistency and completeness, or the measurement of performance. They should be able to perform verification and validation of software. These activities should consider all system requirements, not just functional and performance requirements. Students should be able to apply and validate predictive models, such as those for software reliability or project cost estimation. They should be able to evaluate new technologies and tools to determine which are applicable to their own work.

The word *appropriate* occurs several times in the objectives above. The software engineering discipline is new and changing, and there is not a consensus on the best set of representations, methods, or tools to use. Each curriculum must be structured to match the goals and resources of the school and its students.

7.2. Prerequisites

The fundamental prerequisite for an undergraduate software engineering program is completion of an appropriate secondary school curriculum. This should include four years each of mathematics and English and three years of science. College and university entrance requirements generally address this prerequisite adequately.

We believe that another prerequisite ought to be considered: programming ability. This suggestion is based on an observation about programs in the sciences and engineering. It is perhaps most easily described through analogies.

Consider the introductory sequence in physics. Entering students have developed through life experiences an intuitive understanding of virtually all of the physical phenomena to be studied: motion, velocity, acceleration, gravity, mass, force, heat, light, waves, electricity, magnetism, etc. This permits the first courses to say to the students, "You already know what all these things are; now we will reexamine them scientifically and with mathematical precision." Similar analogies can be drawn for the common engineering fields, such as civil, mechanical, and electrical engineering.

We believe that software engineering is and will continue to be evolving in the direction of increased use of formal or structured methods. To teach the growing body of formal methods for programming, the student needs a knowledge of the programming concepts that are being made precise. This knowledge is unlikely to be acquired through life experience. However, it *can* be acquired through a high school advanced placement course in computer science or through a typical first programming course at the college level.

This prerequisite suggestion is certain to be controversial. It is contrary to the beliefs of some of computer science educators who maintain that precise formal development of programs (and algorithms) should be taught from day one. A few schools have experimented with this approach, with varying degrees of success (see, for example, [Mills89]). These experiments, however, were in computer science curricula. A software engineering curriculum has different objectives, so it is not necessarily the case that any successes (or failures) in a computer science curriculum will carry over.

We note in passing that the report of the ACM task force on the core of computing, a controversial report itself, also suggests programming ability as prerequisite for the introductory course in computing [Denning89]. Furthermore, ABET accreditation guidelines for engineering programs specifically exclude "computer programming skills" from the list of acceptable engineering sciences.

7.3. Technical Content of a Curriculum

There are several possible approaches to selecting the curriculum content. A "curriculum engineering" approach would be to derive the content directly from the objectives. However, the purpose of this report is to promote discussion rather than provide definitive answers, so we believe it is more effective to use another approach. In this section we first examine some curriculum content recommendations that have appeared in the literature. Second, because we believe that the

accreditation criteria of ABET and CSAB represent a significant amount of thought on these issues, we examine the implications of those criteria on curriculum content.

7.3.1. BCS/IEE Curriculum Recommendations

The most significant work to date on identifying the content of an undergraduate software engineering program has been done by a joint working party commissioned by the British Computer Society (BCS) and the Institution of Electrical Engineers (IEE). The working party began its work in February 1988; it produced a preliminary report in March 1989 and a final report in June 1989 [BCS89]. This report is essential reading for anyone interested in software engineering curriculum design.

The report discusses a variety of curriculum issues. A section on context addresses the purpose of the curriculum, the need for flexibility and variation in the curriculum from one school to another, the balance of breadth and depth, prerequisites, and resource requirements. It also discusses what it terms an *engineering ethos*, which includes four themes or components: theory, technology, practice, and application. The report argues that there are "educational advantages in a holistic treatment of Software Engineering which deliberately keeps the four components of engineering interacting throughout a course."

A section titled "Pervading Themes" presents design and quality as themes that must be conveyed throughout a curriculum. An argument is made that design is more critical for software engineering than for the traditional engineering disciplines. A detailed discussion then describes many aspects of design that should be taught. Quality, the second pervading theme, is described in terms of cost, timeliness, reliability, and functionality. The discussion presents a large number of techniques that help achieve quality and suggests how they might be incorporated into courses.

The curriculum content section of the report is organized around three kinds of skills: central software engineering skills, supporting fundamental skills, and advanced skills. These are summarized below.

Central software engineering skills:

- System design and the design of changes to systems
- Requirements analysis, specification, design, construction, verification, testing, maintenance and modification of programs, program components, and software systems
- Algorithm design, complexity analysis, safety analysis, and software verification
- Database design, database administration and maintenance
- Design and construction of human-computer interaction

- Management of projects that accomplish the above tasks, including estimating and controlling their cost and duration, organizing teams, and monitoring quality
- Selection and use of software tools and components
- Appreciation of commercial, financial, legal, and ethical issues arising in software engineering projects.

Supporting fundamental skills:

- Information handling skills: listening, questioning, searching literature, reading documents; oral and written reporting; presentation, and working in teams
- Mathematical skills: methods, notations, and results (mostly from discrete mathematics)
- Knowledge of computer architecture and hardware
- Knowledge of digital communication systems
- Numerical methods
- Knowledge of some major existing components and systems, such as operating systems, communications protocols, programming languages, programming environments, numerical libraries, graphics standards
- Contextual awareness: the changes in hardware and software technologies that are developing, and the forces that drive those changes

Advanced skills (including increased depth in the preceding areas):

- Communications and networks
- Compiler construction and optimization
- Computability
- Computational solid geometry
- Concurrent programming
- Data modeling
- Declarative programming methods and related computer architectures
- Distributed systems
- Formal logics and inference
- Hardware/software interfaces
- High performance parallel computing
- High quality graphical rendition and animation
- Human-computer interaction
- Object-oriented paradigm and related computer architecture
- Mathematical software

- Memory-based reasoning
- Real-time systems
- Requirements analysis and specification
- Safety-critical systems
- Semantics
- Statistical inference and pattern recognition
- Verification

The report also includes an extended discussion of pedagogical issues, including suggestions for alternative ways of approaching the same material. For example, it describes three approaches to teaching programming methodology: starting with abstraction, starting with practice, and starting with rigor.

7.3.2. ACM and IEEE-CS Curriculum Recommendations

The ACM and the IEEE Computer Society have created the Joint Task Force on Undergraduate Curricula in Computer Science and Engineering, which has been addressing undergraduate computing curriculum issues for two years. Their primary goal is to identify the material that is common to all computing programs, including computer science, computer engineering, and possibly information systems and software engineering. We had hoped to be able to include in this report a discussion of the task force's recommended common material and its relationship to potential software engineering curricula. However, the group has not yet issued a public report.

Preliminary versions of the task force's recommendations have been presented at public meetings and have been circulated to reviewers (including the author of this report). The recommendations are structured as *knowledge units* in ten broad areas:

- Algorithms and data structures
- Programming languages
- Computer architecture
- Numerical and symbolic computation
- Operating systems
- Software methodology and engineering
- Database and information retrieval
- Artificial intelligence and robotics
- Human-computer communication
- Social, ethical, and professional issues

Each knowledge unit addresses a relatively small and coherent body of knowledge. Their descriptions include not only suggested lecture topics, but also suggested laboratory exercises, relationships (such as prerequisite) to other units, and an estimate of the total lecture hours needed to cover the material. The total lecture hours for all knowledge units is about 300; this represents 7.5 semester courses (assuming 40 lecture hours per semester).

We believe it is likely that the final recommendations of this task force will be compatible with the ideas for an undergraduate software engineering curriculum presented in the next chapter. The task force's final report, expected late in 1990, will certainly be required reading for undergraduate computing curriculum designers. We plan to examine that report and discuss its implications for software engineering curricula in our next undergraduate curriculum report.

7.3.3. Other Recommendations from the Literature

David Parnas makes an argument for educating "computing professionals" as engineers [Parnas90]; this paper (along with the reports of the professional societies described above) is required reading for anyone interested in software engineering curriculum design. He describes some of the history of computer science curricula, and then sketches a curriculum drawn largely from courses likely to exist already at universities with engineering schools. He chooses 13 courses in mathematics, 3 in science, 8 in engineering, and 8 in computing science. A prerequisite is that the student is a "capable programmer."

We find ourselves in sympathy with Parnas's fundamental premises, but we have three reasons why we cannot support his implementation. First, his required technical courses would constitute at least 80% of an undergraduate program. That leaves insufficient time for coursework to meet curriculum objectives other than career preparation. Second, we do not believe that a curriculum should be simply a collection of independent, existing courses. A curriculum should be designed with specific goals and with coherent course sequences that address those goals. Third, all of the recommended engineering and computing science courses seem to be engineering science rather than engineering design, leaving a large gap in the student's education. In particular, there is a body of knowledge about the engineering processes by which large software systems can be brought into existence that is not reflected in the suggested courses.

7.3.4. Implications of ABET Criteria

If we assume that ABET accreditation will be a goal for some or many undergraduate software engineering programs, then it is useful to examine the implications of

ABET accreditation criteria for curriculum design. The criteria are summarized in Chapter 4 of this report. Some of the implications are described below.

The mathematics requirement includes differential equations, and courses in advanced calculus and linear algebra are recommended. Discrete mathematics is not mentioned in either the required or recommended courses. It is apparent that these mathematics requirements are derived from the traditional engineering disciplines' view of engineering as applying the materials and forces of nature, which are best modeled with continuous mathematics. A software engineering curriculum that satisfies ABET's current criteria would need a large mathematics component to include both the required continuous mathematics and the needed discrete mathematics.

Computer programming skills are specifically excluded from the mathematics, basic science, and engineering science categories in the ABET criteria. A software engineering curriculum will necessarily include material that can be called informally "computer programming." Packaging this material with appropriate computer science and software engineering fundamentals and principles may be critical to its acceptance under current criteria.

The engineering sciences that are the foundations for the traditional engineering disciplines include mechanics, thermodynamics, electrical and electronic circuits, materials science, and transport phenomena. None of these is directly applicable to software engineering. Computer science is also considered an engineering science, so most of the engineering science component of a software engineering curriculum would be computer science. ABET requires, however, that at least one engineering science course be outside the major discipline. Of the remaining choices, a course in electronic circuits may be most valuable.

Other ABET criteria address courses in the humanities and social sciences, and curriculum content in oral and written communication, computer-based experiences, laboratory experiences, and ethical, social, economic, and safety considerations in engineering. It should not be difficult to incorporate all of this kind of material into a software engineering curriculum.

We must keep in mind, however, that the ABET accreditation criteria are not etched in stone; they change to reflect changes in engineering practice and engineering education. Not too many years ago, computer science was not listed among the engineering sciences and computer engineering was not listed as an engineering specialty with separate accreditation guidelines. In designing a software engineering curriculum, it will be better in the long run to choose courses that reflect the spirit of the ABET criteria rather than blindly following all the existing requirements. Well-designed curricula and programs producing highly competent software engineers will influence future ABET criteria.

7.3.5. Implications of CSAB Criteria

Although a software engineering curriculum will probably be closer in spirit and structure to an ABET-accredited engineering curriculum, the CSAB accreditation criteria represent a significant effort and should be considered in curriculum design. The CSAB criteria are somewhat more flexible than those of ABET and include two-thirds year of unspecified coursework. It is likely that a software engineering curriculum designed in the spirit of the ABET criteria may satisfy the CSAB criteria as well.

The mathematics requirement is one-half year including calculus, discrete mathematics, and probability and statistics. All are relevant to a software engineering curriculum. Likewise, the science requirement is reasonable for a software engineering curriculum: a two-semester laboratory science sequence plus two other courses in science or courses with strong emphasis on quantitative methods. The requirement of one and one-third years of computer science courses is also reasonable if we assume that courses in software engineering count toward this requirement.

7.4. Liberal Education Content of a Curriculum

There are a number of opinions on the importance and content of the "liberal" or "non-technical" component of an engineering curriculum. The ABET accreditation criteria state:

> Studies in the humanities and social sciences serve not only to meet the objectives of a broad education, but also to meet the objectives of the engineering profession. Therefore, studies in the humanities and social sciences must be planned to reflect a rationale or fulfill an objective appropriate to the engineering profession and the institution's educational objectives. In the interests of making engineers fully aware of their social responsibilities and better able to consider related factors in the decision-making process, institutions must require coursework in the humanities and social sciences as an integral part of the engineering program. This philosophy cannot be overemphasized. To satisfy this requirement, the courses selected must provide both breadth and depth and not be limited to a selection of unrelated introductory courses.

ABET gives examples of acceptable subject areas including philosophy, religion, history, literature, fine arts, sociology, psychology, political science, anthropology, economics, and foreign languages.

Samuel Florman [Florman86] argues for increased liberal education of engineers: "The fact is that engineers are not receiving essential elements of a traditional college education–and this is occurring at a time when more and more of their fellow citizens are doing so, and when engineers need to be better informed about the world

around them to function effectively." He cites a 1984 report by the National Institute of Education that calls for all recipients of bachelor's degrees to have at least two full years of liberal education and that urges professions such as engineering to extend their programs accordingly. He notes that at its 1985 education conference, the American Society of Civil Engineers formally recommended a five-year program, including more liberal arts courses, for entry into the profession.

The content of the liberal arts component of a software engineering curriculum can vary considerably from school to school and from student to student. The important consideration is that the students have a sufficiently broad understanding of society to permit them to practice their profession effectively.

7.5. Pedagogical Considerations

A good curriculum design must consider more than just objectives, prerequisites, and content. Pedagogy and the kinds of educational experiences provided to the students are also important.

[Denning89] makes a statement about pedagogy and the importance of lifelong learning for computing professionals:

> The curriculum should be designed to develop an appreciation for learning which graduates will carry with them throughout their careers. Many courses are designed with a paradigm that presents "answers" in a lecture format, rather than focusing on the process of questioning that underlies all learning. We recommend that [the task force on undergraduate curricula] consider other teaching paradigms which involve processes of inquiry, an orientation to using the computing literature, and the development of a commitment to a lifelong process of learning.

Often our conceptual models of processes can be fundamentally altered by the words we use to describe them. At the 1986 SEI Software Engineering Education Workshop, Fred Brooks described how his view of software development changed the first time a colleague use the phrase "build a program" instead of the more familiar "write a program." (Brooks himself now recommends we consider the phrase "grow a program.") In the case of the learning process, the eminent Swiss psychologist Jean Piaget suggests that learning is "constructing new knowledge" rather than "receiving new knowledge."

Psychologists distinguish *declarative* knowledge and *procedural* knowledge. The former is easy to write down and easy to teach; the latter is nearly impossible to write down and difficult to teach. It is largely subconscious, and it is best taught by demonstration and best learned through practice. Many of the processes of software engineering depend on procedural knowledge. Thus, it is essential that we design our courses and laboratories to allow our students to experience the engineering

process, to construct their own procedural knowledge. There is certainly an element of truth in the humorous statement quoted in [Weinberg84]: "[The lecture method is] a way of getting material from the teacher's notes into the student's notes– without passing through the brain of either one."

Research on the intellectual development of college-age students also has implications for engineering curricula [Perry70, Culver82]. Perry identified several stages of development in students over their four years in college. In the early stages, the students were more inclined to see all issues as black or white and to expect absolute answers from a suitable authority. In the later stages, they accepted the concept of judgment being applied to select among a range of alternatives. (He notes that students unable to make the transition to the later stages often took refuge by majoring in the physical sciences!) This work suggests that an engineering curriculum should concentrate the fundamental mathematics and engineering science in the early years and wait for the junior or senior years to attempt to teach the application of judgment to engineering design problems.

Finally, although it is beyond the scope of this report to treat in detail, we note that there are a number of efforts underway to reexamine and restructure professional education. These are based on our increased understanding of how students learn. Schein summarizes one of these new approaches as follows:

> The new professional school would start with a learning theory that integrates basic sciences, applied sciences, and professional skills within single learning modules rather than separating them into successive 'core courses,' 'applied courses,' and 'practicum.' The new professional school would be organized around learning modules of varying lengths and would permit the putting together of different patterns of modules, dealing with different professional career foci, leading to different kinds of professional degrees which would require different lengths of time to complete. [Schein72]

An example is the Stanford Law School, which offers four different degrees: Doctor of Jurisprudence, Doctor of Juristic Science, Master of Science in Law, and Master of Jurisprudence. Harvard Medical School, while not offering a variety of medical degrees, has recently begun a major experiment that uses a small team, case study approach rather than traditional lectures and laboratories; this approach integrates many basic sciences, applied sciences, and practical experiences.

Although these approaches are being tried at the graduate professional level, the ideas behind them should be considered as we begin to design the range of undergraduate and graduate professional education for software engineers.

8. An Exercise in Curriculum Design

A major goal of this report is to stimulate discussion in the software engineering community of undergraduate education. To help achieve this goal, we believe it is helpful to sketch the design of an undergraduate curriculum in software engineering. This belief is based on our experiences with the design of a master's level curriculum. Nothing generated more discussion than a strawman curriculum that could be circulated, criticized, dissected, and redesigned.

In undertaking this design exercise, we remember the words of John Hopcroft in his 1986 ACM Turing Award lecture, in which he described his arrival at Princeton in 1964 [Hopcroft87]:

> Princeton asked me to develop a course in automata theory to expand the scope of the curriculum beyond the digital circuit design course then being offered. Since there were no courses or books on the subject, I asked [Edward] McCluskey to recommend some materials for a course on automata theory. He was not sure himself, but he gave me a list of six papers and told me that the material would probably give students a good background in automata theory. ...
>
> At the time, I thought it strange that individuals were prepared to introduce courses into the curriculum without clearly understanding their content. In retrospect, I realize that people who believe in the future of a subject and who sense its importance will invest in the subject long before they can delineate its boundaries.

8.1. Design Constraints

In performing our design exercise, we accept the following constraints on the curriculum:

1. It exhibits the general structure of an engineering curriculum.
2. It is reasonably close to the accreditation guidelines of both ABET and CSAB, to the extent that those two sets of guidelines are compatible.
3. It incorporates the most up-to-date software engineering knowledge that is appropriate at the undergraduate level.
4. It incorporates the material defined by the ACM/IEEE-CS joint curriculum task force to be common to all undergraduate computing curricula. (Because the task force has not yet published its report, our design has been influenced by a partial report distributed confidentially to reviewers, including the author of this report. We have agreed not to publish at this time a detailed discussion of how the task force's material maps into our design.)

5. It reflects appropriate pedagogical considerations, such as those described in the previous chapter.
6. There exists a feasible and reasonable top-down evolutionary strategy for introducing it.

8.2. Curriculum Structure

If we accept the premise that software engineering is or will soon become a true engineering discipline, it is reasonable to expect that an undergraduate curriculum would have the same basic structure as successful curricula in other branches of engineering. Some of that structure is inherent in the ABET accreditation criteria.

It also seems reasonable to consider the CSAB accreditation criteria, which, although not intended to apply to engineering curricula, still represent a significant body of opinion about undergraduate curricula in the computing disciplines. Hence we begin with the first two constraints listed above.

On the other hand, we believe that it is a mistake to try to force a software engineering curriculum to fit accreditation criteria that were not developed with software engineering in mind. It will almost certainly be many years before accreditation of these programs will be possible. As programs evolve, the most important consideration is that they are structured in whatever way best achieves their educational objectives. Nevertheless, it is useful to compare the two sets of criteria to see what common structure, if any, they suggest for a software engineering curriculum. Figures 8.1 and 8.2 summarize the guidelines of ABET and CSAB, respectively.

Requirement	ABET Content Category
25%	Mathematics and Basic Sciences
25%	Engineering Sciences
12.5%	Engineering Design
12.5%	Humanities, Social Sciences
25%	Electives

Figure 8.1. ABET guidelines

Requirement	CSAB Content Category
22.5%	Mathematics and Sciences
33.3%	Core and Advanced Computer Science
27.5%	Humanities, Social Sciences, Arts, Other
16.7%	Electives

Figure 8.2. CSAB guidelines

In light of CSAB's apparent goal of permitting accreditable liberal arts programs, it is not surprising that it is not possible to define a curriculum that simultaneously satisfies both sets of criteria. However, we can come close if we consider the ABET categories of engineering science and engineering design together to be the counterpart of the CSAB categories of core and advanced computer science together, and if we require half of ABET's electives to be in the humanities and social science areas. Figure 8.3 shows this common curriculum structure, including a breakdown by semester hours based on a 120 semester hour degree requirement.

Requirement	Semester Hours	Content Category
22.5%	27	Mathematics and Basic Sciences
37.5%	45	Software Engineering Sciences and Software Engineering Design
25%	30	Humanities, Social Sciences
15%	18	Electives

Figure 8.3. Curriculum structure for software engineering

This curriculum deviates from the ABET requirements by shifting one course from the mathematics and science category to electives. On the other hand, it includes somewhat more technical material in the major field (perhaps two courses) than the CSAB requirements, resulting in a corresponding reduction in the number of free electives. Also, it does not explicitly include the arts along with the humanities. This would probably prevent such a curriculum from fitting into a liberal arts college. We do not believe this is necessarily a flaw in the curriculum, but rather a reflection of the belief that an undergraduate software engineering program is closer in character to other engineering programs than to liberal arts programs in the sciences.

To assess the validity of the software engineering curriculum structure in Figure 8.3, we must examine the potential curriculum content in each of the categories, and we must address the question of whether such a curriculum can provide an integrated educational experience that achieves a reasonable set of educational objectives. Toward this end, we try below to identify some of the appropriate mathematics, science, and software engineering content of the courses in the various categories.

8.3. Curriculum Content Sketch

As we have noted earlier, the design of a curriculum is a complicated task, and the result is more than a list of courses or a description of their content. The material presented in this section is clearly insufficient to be considered a complete curriculum design. Its purpose is to sketch the content and organization of a strawman curriculum and to provide some of the motivation and rationale for the content. We invite comments and hope to incorporate many of the suggestions we receive in subsequent SEI curriculum reports.

8.3.1. Mathematics and Science

The mathematics and science content of the curriculum should help achieve two fundamental objectives. First, it should prepare students to participate competently in an increasingly technological society. This includes the ability to understand science and technology issues well enough to make informed political decisions. Second, the science and mathematics content should provide the students with an appropriate foundation for subsequent software engineering courses.

Mathematics has been particularly useful to engineers in that it allows abstract models of the physical world to be built and analyzed. The results of that analysis can be transferred back to the physical world, especially to the artifacts that the engineer is building. Nearly all the techniques applied by engineers in the traditional disciplines are based on continuous mathematics. At the heart of that branch of mathematics are differential and integral calculus and differential equations.

The models built by software engineers, however, are much more likely to rely on discrete mathematics. It is often used to model the behavior of digital computer and digital systems (above the level of the electronic devices), the behavior of software, and the behavior of systems that include software. Especially useful is the wide range of formalisms lumped together under the title *logic*. Therefore, to achieve the same overall goals that led to inclusion of continuous mathematics in traditional engineering curricula, a significant amount of discrete mathematics is essential in a software engineering curriculum.

Probability and statistics are likely to become increasingly important to software engineers, not only for purposes of modeling systems controlled by or simulated by software, but also for modeling software engineering processes. An example is software reliability modeling.

Calculus still has important roles in software engineering curricula. It is useful in approximation techniques in the analysis of algorithms; it is used in software performance analysis; it is prerequisite to some of the topics in probability and statistics; it is the basis for many concepts in numerical analysis and scientific computing; and it is necessary for the study of physics.

Differential equations, on the other hand, has almost no application to software engineering *per se*. An undergraduate curriculum in software engineering therefore does not require a course (theoretical or applied) in differential equations. Of course, the application software in many areas of science and engineering might involve numerical solution of or other use of differential equations, and students committed to careers in such application domains might choose mathematics, science, and other elective courses accordingly.

A topic that spans the boundaries of discrete mathematics, continuous mathematics, and computer science is numerical methods. An understanding of the limitations of digital computers when performing calculations on (ostensibly) real numbers is a fundamental part of the education of a software engineer.

While the physical and life sciences are fundamental to traditional engineering disciplines, they provide virtually no basis for software engineering. The only significant exception is that electricity and magnetism, common topics in introductory physics courses, support the study of the computer itself, and software engineers need a basic understanding of the machine for which they are developing software. To achieve the first goal stated above, however, it is probably the case that basic knowledge of physics, chemistry, and biology are essential in almost any undergraduate curriculum. Chemistry and biology, in particular, are likely to be increasingly important in understanding society's health care, environmental, and genetic engineering issues in the next century.

An understanding of science necessarily includes an understanding of the methods of science, including laboratory methods. Therefore, we assume that there is a reasonable laboratory component to at least some of the science courses. Appropriate introductory physics and chemistry courses are quite likely to satisfy this requirement.

This discussion leads us to recommend the mathematics and science requirements shown in Figure 8.4

Subject	Courses
Discrete mathematics	2
Probability and statistics	1
Calculus	2
Numerical methods	1
Physics	1
Chemistry	1
Biology	1

Figure 8.4. Mathematics and science requirements

8.3.2. Engineering Science and Engineering Design

An engineering curriculum includes a substantial amount of *engineering science* and *engineering design*. In the traditional engineering disciplines, the engineering science component includes courses is areas such as mechanics, thermodynamics, electrical and electronic circuits, materials science, and transport phenomena. Engineering design courses cover a process that includes establishment of objectives and criteria, synthesis, analysis, construction, testing, and evaluation.

The corresponding topics for a software engineering curriculum will be different. In particular, software engineering *per se* does not involve processes or products for which such sciences as mechanics, thermodynamics, materials, or transport phenomena are relevant. (We recognize, of course, that software may be involved in the development or control of engineered systems for which these sciences are important. Thus these sciences may be important to software engineers who are committed to working in a particular application domain.) To determine appropriate software engineering topics, we must examine the purposes served by engineering science and engineering design in the traditional engineering disciplines.

The engineering sciences are described by ABET as providing a bridge between mathematics or basic science and engineering practice. Knowledge of these sciences permits an engineer to reason about the artifacts he or she intends to build before they are built. It allows the engineer to design a highway bridge that doesn't collapse the first time a truck passes over it and an aircraft that doesn't crash on its first flight. Software engineers need a kind of engineering science that would permit similar kinds of confidence and predictability in software systems. This kind of science should provide analytical tools or capabilities for the software engineer.

The engineering design component of an engineering curriculum is described by ABET as including, among other things, development and use of design methodology, formulation of design problem statements and specifications, consideration of

alternative solutions, feasibility considerations, and detailed system descriptions. These concepts can be incorporated into a software engineering curriculum as stated. The only difference from a traditional engineering discipline is the kind of artifact to be constructed. It may also be noted that this direct applicability of the concepts of traditional engineering design to software engineering is one of the strongest arguments for considering software engineering to be a true engineering discipline.

A curriculum need not have the engineering science and engineering design segregated and placed in different courses. In fact, for software engineering it is not always possible to determine whether a particular topic is in one category or another. In the following four subsections, therefore, we sketch the content of a software engineering curriculum in four categories with more descriptive titles: software analysis, software architectures, computer systems, and software process.

The curriculum structure presented in Section 8.2 suggested 45 semester hours of software engineering courses. The courses material described below totals 14 courses or 42 semester hours. Therefore one software engineering elective course will also be allowed.

8.3.2.1. Software Analysis

The software analysis component of the curriculum provides the student with the knowledge to describe, model, and reason about software and software processes. This allows them to predict properties for software systems, such as reliability, performance, fault-tolerance, and safety. Some of the topics in this area are:

- Formal development of algorithms and programs, including basic concepts of formal verification
- Abstraction and modeling as techniques
- Formal systems that can be applied in software engineering: automata, formal languages, etc.
- Measurement of software processes and products
- Analysis of algorithms
- Performance analysis and prediction
- Computability
- Fundamental concepts of control theory
- Fundamental concepts of information theory

The major goal of this curriculum component is to instill in the student the idea that he or she can reason about software rather than employ the ad hoc or trial-and-error techniques that are prevalent in today's computer science curricula. To achieve this

goal, the material must be taught early in the curriculum (beginning in the freshman year) and reinforced throughout all courses.

We estimate that the material described above would require approximately three one-semester courses. However, in keeping with the idea that professional education might be improved by closer ties between theory and practice, we can foresee curricula that combine this material with some of the discrete mathematics, probability, and statistics material to produce four or five courses. We can also foresee curricula that combine some of this material with some of the software architectures material described below.

8.3.2.2. Software Architectures

A characteristic of the more mature engineering disciplines is *routinization*, which is the ability to solve recurring problems by routine application of known results rather than rederiving those results from basic principles. For example, a civil engineer faced with the problem of designing a highway bridge does not begin by applying differential equations to determine the stress on beams. It is much more likely that the engineer will estimate the average and peak traffic patterns for the bridge, measure the span, determine the soil characteristics at both ends of the proposed bridge, and then order something like a "standard A-304-X highway bridge" from the state highway department's handbook of bridges. This is routine practice, but it is nevertheless real engineering.

Although software engineering is not as mature as civil engineering, there are many recurring problems for which there are widely accepted (and in some cases, provably optimal) solutions or software architectures. An important part of the education of a software engineer is to know how to recognize these recurring problems and how to select among the known architectures for solving them.

In the last century, this kind of knowledge was provided to engineers in the traditional disciplines through apprenticeships. In fact, it was not until 1916 that the majority of engineers in the United States had attended college at all, let alone graduated with an engineering degree. Software engineering over the last 40 years has also depended greatly on informal apprenticeships rather than college degrees in software engineering as the mechanism by which practitioners learned the accepted practices of the profession. A well-designed curriculum can provide much of this knowledge for software engineers while they are still in school. This can be of great value, especially if the curriculum includes the best known practices from throughout the profession, rather than just those of one or two organizations.

It is very important to note that software engineers will not necessarily know any one software architecture as well as a computer scientist who specializes in that area. When faced with the task of building an operating system, database system, compiler, or other system whose basis is in computer science, the software engineer

must know to assemble a project team that includes computer scientists with deep knowledge of the science behind those systems. Similarly, when building an avionics or radar system, the software engineer needs to bring scientists and engineers from those disciplines to the team. The software engineer specializes in building large systems rather than in the science behind all possible application areas.

The structure of this component of the curriculum is still under development. An example may help explain how it differs from courses in a typical computer science curriculum.

Consider the recurring problems of modeling or controlling concurrent processes and managing computing resources such as processor time and memory space. In most computer science curricula, these topics would be discussed in a course on operating systems. A result is that most students will finish the course believing that concurrent programming, process scheduling, and memory management are useful *only* when building a general purpose, timesharing operating system. A software architectures approach places the recurring problems and their solutions at the highest level, with operating systems and other applications brought in as examples. (An application of this idea to the design of an undergraduate computer science curriculum can be found in [Shaw85].)

Some of the topics in this curriculum component are:

- Representation of data, information (including graphics and sounds), and knowledge, from atomic through very large structures (this topic includes traditional areas such as data structures, database systems, and graphics)
- Resource management: time, memory, processors (an application area where these issues are critical is operating systems)
- Expert systems
- Embedded real-time systems
- Concurrent, parallel, and distributed software systems
- Translator systems (this topic includes compilers and assemblers)

We estimate that the material in this component will require four one-semester courses. Because a major emphasis in these courses is analysis of why the particular architectures are good, appropriate analysis techniques are prerequisite. However, it is not appropriate that the students complete all the software analysis courses before beginning the first software architectures course. In many cases, two courses in these two categories might be taken concurrently. It may also be appropriate to design courses that include topics from both categories. An example is a course that combines elementary analysis of algorithms with elementary data structures; without the ability to analyze algorithms, the justification for certain data structures is lost.

8.3.2.3. Computer Systems

Software engineering is not all software. A software engineer must have a thorough understanding of computer systems, including how the computer fits into larger engineered systems. Most existing computer science curricula are deficient in this area.

Topics in this area include:

- Computer organization and assembly language
- Computer architecture
- Digital systems, including laboratory work
- Embedded systems, interfacing computers to other devices
- Data communications
- Networks
- Fundamental concepts of robotics

We estimate that this material will require three one-semester courses. Because these courses rely on some of the mathematics and software analysis material, they are likely to be taken by students in the sophomore and junior years.

8.3.2.4. Software Process

The software process component is the software engineering curriculum counterpart to engineering design. We choose this name partly because it is becoming a widely understood term and partly because the term "design" has a much narrower interpretation in the software community than in engineering in general.

Humphrey [Humphrey89] defines the term this way: "The software process is the set of tools, methods, and practices we use to produce a software product." Some of the topics in this area are:

- Requirements analysis
- Specification concepts and formal methods
- Design
- Implementation techniques and languages
- Verification and validation
- Software evolution
- Evaluation of software products and processes
- Software development team organization and management
- Professional, ethical, and legal issues in software engineering

We estimate that this material will require four one-semester courses. Because these courses will require the students to demonstrate engineering judgment, the pedagogical considerations discussed in Section 7.4 suggest that the courses be taken in the junior and senior years. The software process courses build upon most of the courses in the other three areas, which also argues for placing them late in the curriculum.

One possible structure for the courses is as two year-long project courses. The fundamental methods of the software process could be introduced in the junior year sequence and then reinforced and augmented in the senior year sequence. One of the projects might be a maintenance effort rather than new development. In fact, an argument can be made that the students should experience software maintenance *before* attempting new development. This allows experience with a substantially larger software system than could be developed by student teams. A carefully chosen system can show the students important software architectures; and maintenance projects can better motivate topics such as configuration management and version control. With this experience, students will be better prepared to "design for maintainability" in the subsequent project.

Whatever the course sequence, the students should be given substantial opportunities to experience, rather than just be told about, all parts of the software process. Ways to accomplish this have been described in previous SEI reports [Tomayko87, Engle89a].

8.3.3. Humanities, Social Sciences, and Electives

Section 7.4 of this report presented some ideas about the liberal education part of a software engineering curriculum. Although we cannot give a list of specific courses that are appropriate, we can suggest some goals for those courses.

The students should develop oral and written communication skills. This should include not only the mechanics of these skills, but also the ability to think critically and express ideas creatively. Literature, philosophy, and history courses can contribute to these skills. The students should also develop an understanding of the history and structure of civilization, government, and society, including cultures other than their own, and an understanding of the emerging global economy. This can help place the engineering profession in an appropriate context.

The curriculum structure presented in Section 8.2 allows 30 semester hours, or ten courses, in the humanities and social sciences. It also allows 6 elective courses. This provides considerable latitude for schools and students to develop appropriate programs with both breadth and depth in the liberal arts.

8.4. Descriptions of the Core Courses

In the previous section, we suggested that the material in the software engineering courses might total fourteen courses in the areas of software analysis, software architectures, computer systems, and software process. In this section, we present very rough sketches of those fourteen courses.

8.4.1. Software Analysis 1

This course introduces the basic principles of software analysis. Its goal is to give students an ability to apply formal and mathematically precise reasoning abilities to programming. It introduces the idea of engineering measurement and the kinds of things that can be measured for software: algorithmic complexity (worst case), program performance, memory usage, reliability, etc. It also introduces the concepts of abstraction and modeling. Student exercises include programming, but with emphasis on careful development and on analysis of the algorithms developed.

Prerequisites: programming ability (equivalent to AP computer science)
Discrete Mathematics 1 (logic)

Topics: Reasoning about software products and processes
Formal development of algorithms and programs
Concepts of formal verification
Recursive algorithms
Programming paradigms
Software engineering measurement
Fundamentals of analysis of algorithms

8.4.2. Software Analysis 2

This course continues the development of analytic and formal methods skills. Modeling is stressed, especially finite state machines and similar models that are useful for software systems (as opposed to abstract computability models). Calculus-based complexity theory is introduced to allow analysis of expected behavior (rather than worst case). Important algorithms, such as searching and sorting, are analyzed. Computability and algorithmic intractability are presented to show absolute and practical bounds on computing.

Prerequisites: Software Analysis 1
Discrete Mathematics 2
Calculus 2

Topics: Modeling concepts
Formal models of computation
Basic algorithm design strategies
Analysis of algorithms: advanced concepts, intractability

Computability

8.4.3. Software Analysis 3

This course introduces a variety of advanced theoretical and analytic material appropriate for an engineering curriculum. Much of the material is concepts from traditional engineering disciplines adapted to software.

Prerequisites: Software Analysis 2
Computer Systems 1
Calculus 2
Probability and Statistics

Topics: Performance analysis, including concepts of hardware and software monitors
Software reliability
Fundamental concepts of control theory
Fundamental concepts of information theory
More modeling concepts, including asynchronous and parallel models

8.4.4. Software Architectures 1

Architectures at the smallest scale include representation of fundamental data types and data structures. Programming language structures for modularization and encapsulation are also covered, including data abstraction. This and subsequent software architectures courses are taught with emphasis on the application of the results of computer science rather than the derivation of those results. Analysis techniques are applied to all architecture examples to show why they are good.

Prerequisites: Software Analysis 1
Discrete Mathematics (set theory, graph theory)

Topics: Representation of atomic data
Basic data structures
Data structures for search problems
Fundamental programming language structures

8.4.5. Software Architectures 2

This course presents software architectures related to the recurring problems of concurrent systems and management of time and memory resources. The architectures of several common kinds of system components are presented, including kernels and layered architectures (as are often used in operating systems and programming support environments), and the basics of window managers and user interfaces.

Prerequisites: Software Architectures 1

Topics: Concurrency
Management of time and memory resources
Architecture of operating systems

Architecture of window managers and user interfaces
Architecture of toolboxes and programming support environments

8.4.6. Software Architectures 3

This course addresses the recurring problems of the representation and manipulation of large bodies of information and knowledge. Included are architectures of file systems, database systems, and expert or knowledge-based systems, and common approaches to data security and protection in large systems. As with the other software architectures courses, the emphasis is on presenting the best currently known solutions to these recurring problems and on using engineering analysis to determine why they are good.

Prerequisites: Software Architectures 2
Software Analysis 2
Computer Systems 1

Topics: Representation of information and knowledge
Fundamental representations of graphic data and sound
Architecture of file systems
Architecture of database systems
Architecture of expert or knowledge-based systems
Data security and protection

8.4.7. Software Architectures 4

This course presents a variety of software architectures for common types of systems. Translator systems include compilers, but the emphasis is on the architecture of the systems and its applicability to other kinds of systems. For example, symbol table techniques can be used in systems other than compilers. The students should gain a basic understanding of the importance of formal language theory to compiler construction, but they need not study that theory. Rather, they should understand that computer scientists specializing in compilers are a necessary part of a software engineering project to build a compiler. Similarly, the software architectures of real-time, embedded, distributed, and network systems are presented. Note that the hardware aspects of these complex systems have already been presented in the prerequisite computer systems courses.

Prerequisites: Software Architectures 3
Computer Systems 2, 3

Topics: Architecture of translator systems
Architecture of real-time and embedded systems
Architecture of distributed and network systems

8.4.8. Computer Systems 1

This course covers the material in the computer organization course in a typical computer science curriculum. The assembly language component is relatively small and not intended to give the ability to develop entire programs in assembly language.

Prerequisites: Software Analysis 1
Discrete Mathematics (boolean algebra)

Topics: Computer organization
Memory systems
Assembly language
Basic concepts of computer architecture
Other computer architectures (supercomputers, etc.)

8.4.9. Computer Systems 2

This course is intended to give the software engineer a sufficient hardware background to work with hardware engineers on all kinds of embedded systems. It should include some laboratory work to give the student an appreciation of interfaces, timing, interrupts, and the basic components of digital systems. Engineering measurement and testing fundamentals should be included.

Prerequisites: Computer Systems 1

Topics: Digital logic and systems
Input and output
Interrupt handling
Interfaces between computers and other devices (sensors, effectors, etc.)

8.4.10. Computer Systems 3

Because many of the most complex software systems are embedded in larger complex hardware and communication systems, a software engineer must have a substantial appreciation of many systems engineering concepts. This course presents many of the hardware aspects of such systems. Concepts of distributed systems, networks, and data communications constitute a major portion of the course. Embedded systems, including applications such as robotics, constitute the remainder of the course. Software analysis material on information theory and control theory are applied in this course.

Prerequisites: Computer Systems 2
Software Analysis 3

Topics: Hardware aspects of distributed systems
Computer networks
Data communications

Embedded systems
Fundamentals concepts of robotics

8.4.11. Software Process 1

This course introduces the process by which large software systems are built by teams of developers. It concentrates on the early life cycle phases and is taught with a substantial student project component. A continuing development or maintenance project may be chosen to allow students to work on a very large system without having to create it in its entirety.

Prerequisites: Software Analysis 2
Software Architectures 2

Topics: Software quality issues
Project planning
Software configuration management
Software technical reviews
Software requirements analysis
Software specification
Structure, content, and standards for specification documents
Software design fundamental principles, methods, and representations

8.4.12. Software Process 2

This course completes a first pass through the fundamental software processes. It concentrates on implementation, verification, and validation. This course can continue the student project begun in the previous course.

Prerequisites: Software Process 1

Topics: Implementation considerations: language structures and programming techniques
Software verification and validation
Software maintenance concepts
Regression testing

8.4.13. Software Process 3

This course begins a second pass through the software process, this time with increased depth, formalism, and/or use of tools. Aspects of the software process that are based more on judgment than on hard science are introduced; these include software project management concepts, cost estimation, and human factors. Professional, ethical, and legal issues in software engineering are also presented. This course should also have a substantial student project that can carry over into the next course. The project should involve new development of a substantial sys-

tem and should allow the students to apply some of the advanced software analysis and architectures material.

Prerequisites: Software Process 2
Software Analysis 3
Software Architectures 4
Computer Systems 2
Numerical Methods

Topics: Software project management and team organization
Cost estimation
Systems engineering considerations
Software prototyping
Human factors
Formal specification languages and tools
Software design methods and tools
Professional, ethical, and legal considerations for software engineers

8.4.14. Software Process 4

This course completes the second pass through the software process. It also stresses increased depth, formalism, and/or use of tools for the later phases of the life cycle. Students should be challenged to demonstrate substantial skills in engineering analysis, synthesis, and judgment.

Prerequisites: Software Process 3

Topics: Software implementation: application generators, reuse
Software verification and validation
Software reliability
Integration, system, and acceptance testing

8.5. Course Schedule

Figure 8.5 shows how the courses in the curriculum might be scheduled. Further refinement of this diagram will be necessary after the content of each of the courses is refined.

The freshman year includes both discrete mathematics and calculus. The probability and statistics course and the numerical methods course are scheduled in the junior year. The basic science courses are placed in the sophomore and senior years because they should follow calculus and because they are not direct prerequisites for other courses.

Curriculum	Freshman		Sophomore		Junior		Senior	
Category	Fall	Spring	Fall	Spring	Fall	Spring	Fall	Spring
Mathematics	● ●	● ●			●	●		
Science			●	●			●	
Software Analysis		●		●		●		
Software Architectures			●	●	●	●		
Computer Systems			●	●	●			
Software Process					●	●	●	●
SE Electives								●
Humanities, Social Science	● ●	● ●	●	●	●	●	●	●
Electives	●		●				● ●	● ●

Figure 8.5. Curriculum schedule

The first software analysis course is placed in the second semester in order to accommodate those students who need to satisfy the programming prerequisite during the first semester and to allow the course to build on some of the discrete mathematics from the first semester. Software architectures and computer systems sequences begin in the sophomore year; these are the heart of the engineering science part of the curriculum. This placement allows, for example, the first course in software architectures (which will be mostly data structures) to use the techniques from the first software analysis course (which includes formal development of algorithms). The software process course sequence, which embodies most of the engineering design part of the curriculum, begins in the junior year. It builds upon the first two courses in each of software analysis, software architectures, and computer systems. The senior year software process courses can build on the more advance courses in the other three categories.

Variations of this schedule are possible. There is considerable latitude in rearranging the science, advanced math, elective, and humanities and social science courses to meet the needs of individual students.

8.6. Program Evolution Strategy

The program sketched above can evolve top-down as a track in a computer science department, as discussed in Section 6.1. The major steps might be as follows:

1. Introduce a one-semester software engineering project course as a senior year elective.
2. Expand the project course to two semesters.
3. Increase the discrete mathematics requirement.
4. Increase the amount of formal software analysis material in existing courses on data structures, analysis of algorithms, operating systems, and theory of computation.
5. Add a digital systems course with laboratory.
6. Expand the software engineering project course to four semesters, introducing increased use of disciplined approaches, the use of software engineering tools, and software maintenance.
7. Introduce the first two software analysis and first two software architectures courses as new, coherent courses.
8. Introduce the remainder of the core courses.
9. Introduce advanced software engineering electives.

We would expect that five years might be needed for this evolution. During that time, not only could the new and revised courses be designed, but faculty members could be given opportunities to develop their own knowledge of software engineering. We might also expect that new textbooks for these courses will begin to appear in three to five years, so the timing of the introduction of some courses may be influenced by the appearance of an appropriate book.

9. Summary and Conclusions

The purpose of this report is to provide a variety of information that will stimulate widespread, rational discussion of issues in undergraduate software engineering education. Those issues include the definition of the software engineering discipline and its relationship to computer science, the need for undergraduate software engineering education, program accreditation, professional licensing, and the design and evolution of undergraduate software engineering curricula. In this chapter we summarize the report's discussions of those issues and draw some conclusions about the future.

Definitions of software engineering appearing in the literature exhibit a small number of recurring concepts that are important to understanding the discipline. These include:

Principles: Software engineering can be and is being built on a number of principles. Software need not always be built in an ad hoc manner.

Discipline: The methods of software engineering are systematic and disciplined; this is absolutely essential for large products built by teams.

Quality: Quality must be built into a software product throughout the development process. It cannot be "added on" through a testing and debugging cycle.

Economics: Software engineers must work in the real world and must therefore recognize and appreciate the economics of building useful systems. Engineering judgment is required.

Software engineering education is needed at the undergraduate level because the vast majority (perhaps 80% or more) of software professionals will not pursue an advanced degree. There are now about 20% fewer 18-year olds in the United States than there were ten years ago, and university enrollment trends indicate a substantial decline in the percentage of students majoring in computer science. The 1990s may produce only 25% to 50% as many graduates in computer science per year as the mid-1980s.

There are currently no undergraduate programs in software engineering in the United States, although there are nearly 1000 colleges and universities offering computer science degrees. We believe that 10% to 20% of those schools could eventually offer undergraduate software engineering degrees.

There are two organizations that might ultimately accredit undergraduate software engineering programs, the Accreditation Board for Engineering and Technology (ABET) and the Computing Science Accreditation Board (CSAB). Both require that

a program have already produced graduates before it can be accredited, so it is still several years before accreditation of software engineering programs will be possible. Furthermore, both depend on professional societies for establishing accreditation guidelines. Thus, the Association for Computing Machinery (ACM) and the Computer Society of the Institute for Electrical and Electronic Engineers (IEEE-CS) will most likely be involved in any effort to accredit software engineering programs.

The general accreditation guidelines of the two organizations are somewhat different, and it is unlikely that a software engineering program could be designed that simultaneously satisfies both sets of guidelines. An agreement between ABET and CSAB makes it most likely that any program with the word "engineering" in its title, including software engineering, would have to be accredited by ABET.

Certification of software professionals is a voluntary process administered by the profession itself. The Institute for the Certification of Computer Professionals has awarded certification in computer programming and in data processing for several years. There are a few efforts currently underway to define certification standards for software engineers and specialists in information systems security.

Licensing of engineers is a mandatory process administered by government (in the United States, by the individual states). Unlicensed engineers are subject to significant restrictions on their professional activities. There does not seem to be any current activity to require licensing of software engineers. For now, this is probably the most desirable situation because the discipline of software engineering is not yet sufficiently mature to permit a meaningful legal definition of appropriate standards for professional competence.

In spite of the fact that undergraduate software engineering programs may ultimately be accredited as engineering programs, it is likely that they will evolve as separate tracks within computer science programs. A top-down evolution strategy introduces a software engineering project course at the senior level and then, over a period of several years, brings increasing coverage of software engineering topics earlier into the curriculum. This strategy has fewer risks than bottom-up or all-at-once strategies, and it postpones the need to fight political battles over the use of the word "engineering" in the program title.

Eventually, a bachelor of science in software engineering (BSSE) degree can be separate from a computer science (BSCS) degree. A BSSE curriculum is likely to look very much like a traditional engineering degree in its overall structure, in that it begins with mathematics and basic science, then presents engineering science, and finally presents engineering design. The major differences from a traditional curriculum are in the choices of which mathematics, engineering science, and engineering design subjects are emphasized.

Because software engineering is not concerned with using the forces and materials of nature, discrete mathematics will have a large role and continuous mathematics a

smaller role. Computer science will be a more important engineering science than will be thermodynamics, for example. The engineering design material is still evolving rapidly; we do not yet know very much about the "routine" practices of software engineering.

We have presented a strawman curriculum for a BSSE degree that focuses on four fundamental software engineering subject areas: software analysis, software architectures, computer systems, and the software process. We solicit comments on this strawman and plan to incorporate suggestions in our next report.

Resources will be required to support increased software engineering at the undergraduate level. Faculty resources will be critical to the introduction of better software engineering education, so faculty development should be a high priority for schools considering new courses and programs (graduate or undergraduate). Fortunately, there has been recently widespread discussion of science and engineering educational issues in government and industry, and it is likely that increased funding will soon be available from a variety of sources.

We conclude that undergraduate software engineering programs are a desirable and inevitable part of the spectrum of software engineering education. Even though the discipline is still emerging, we can introduce high-quality software engineering courses into existing computer science programs, and we can begin designing complete software engineering programs. We can anticipate the emergence of these programs in the mid-1990s and accreditation shortly thereafter. We invite the academic and professional communities to cooperate and collaborate with us to make this vision a reality.

Appendix 1. Report on the SEI Workshop on an Undergraduate Software Engineering Curriculum

The SEI Workshop on an Undergraduate Software Engineering Curriculum was held on July 21, 1989, in Pittsburgh, as part of the 1989 SEI Education and Training Week. The participants were selected from those submitting position papers on appropriate topics. They included:

Lionel E. Deimel	*SEI*
Charles B. Engle, Jr.	*U.S. Army Resident Affiliate at SEI (now at the Florida Institute of Technology)*
Gary Ford	*SEI*
Frank L. Friedman	*Temple University*
Norman E. Gibbs	*SEI*
William E. Richardson	*United States Air Force Academy*
David Alex Lamb	*Queen's University*
Jeffrey A. Lasky	*Rochester Institute of Technology*
James R. Lyall	*Embry-Riddle Aeronautical University*
Frances L. Van Scoy	*West Virginia University*
Richard Louis Weis	*University of Hawaii at Hilo*
Stuart H. Zweben	*Ohio State University*

The workshop began with opening statements by each participant, based on the position papers. It is interesting to note that the positions were widely varied, and there was no apparent trend or consensus on any issue. The position papers of the participants are summarized very briefly below to illustrate their variety; the full papers are in [Gibbs89b].

Lionel Deimel [Deimel89] discusses the importance of programming-in-the-small to both computer science and software engineering curricula. He suggests that computer science is more than programming and that it is right for computer science curricula not to spend inordinate amounts of time addressing issues such as documentation, formal techniques, style, and debugging and testing skills. He then argues that such skills are very important to software professionals, and that they should be emphasized in an undergraduate software engineering curriculum. He supports the idea of establishing such programs.

Chuck Engle [Engle89b] argues that computer science and software engineering are different disciplines. He defines the roles of programmer, computer scientist, soft-

ware engineer, and software project manager to help describe the two disciplines. He then describes a number of issues that would distinguish a software engineering curriculum from a computer science curriculum. He concludes that although we can identify the kind of material that should be in a curriculum, our understanding of large-scale software development is insufficient to allow us to establish a meaningful curriculum at this time.

Gary Ford [Ford89a] suggests that undergraduate software engineering programs are inevitable but that the most valuable contribution in the immediate future is not the design of courses. Instead, we should first examine issues such as how well the software engineering community is being served by existing computer science curricula, what appropriate educational objectives are for undergraduate programs in both computer science and software engineering in the 21st century, and how software engineering programs might realistically evolve in computer science departments.

Frank Friedman [Friedman89a] summarizes his views in the title of his position paper, "A Separate Undergraduate Software Engineering Curriculum Considered Harmful." He argues that most science and engineering disciplines have become increasingly obsessed with technical content, to the detriment of the students' general education. A better approach is to broaden the undergraduate program and postpone specialization to the graduate level.

Norm Gibbs [Gibbs89c] discusses the increasing gap between the skills and attitudes of computer science undergraduates and the needs of the software industry. He supports the ideas of the three paradigms of computing and the common basis for all computing curricula, as described in [Denning89], and suggests that it may be time to consider undergraduate programs that emphasize design over abstraction and theory. This might be accomplished by providing four semesters of software engineering study in addition to the computing core. He concludes that the development of new programs may take faculty from computer science programs, resulting in unfortunate fragmentation of the field of computer science.

Bill Richardson [Jones89] argues that it is premature to develop separate software engineering programs and that it is possible to incorporate the most important aspects of software engineering into an existing computer science program. This can be accomplished by introducing the seeds of software engineering concepts in the beginning courses, using appropriate software engineering tools and methods in courses throughout the curriculum, and adding a year-long capstone course in software engineering to bring all the concepts together.

David Lamb [Lamb89] suggests that several concerns must be addressed prior to considering the content of software engineering programs. First is the audience for various programs in computing: students who want only basic familiarity with computers, those wishing to be computing specialists in another discipline, those prepar-

ing for careers in software, those planning graduate study in computer science, etc. He also discusses differences between science and engineering and how those differences affect curriculum design and content. He also considers the practical and political aspects of introducing a new curriculum with the word “engineering” in its title.

Jeff Lasky [Lasky89] identifies several scenarios for increased software engineering education at the undergraduate level: concentrations within computer science programs; concentrations within computer engineering programs; upper-division offerings drawing students from two-year programs in computer science, information systems, or engineering science; and complete undergraduate programs. He discusses the institutional and political impediments to all of these scenarios and suggests a unique solution. An academic unit outside all affected departments offers a two-year computing core, with students then completing degrees in specific disciplines through various departments.

Jim Lyall [Lyall89] describes some of the areas of software engineering identified by industry as being critically important. Almost none are covered by current computer science programs. He argues that software engineering is appropriately named, despite many narrow interpretations of the definition of “engineering” that have appeared. He also describes an effort at his university to determine the feasibility of establishing an undergraduate software engineering program.

Frances Van Scoy [VanScoy89] provides a detailed and wide-ranging plan for evolving a software engineering program within an existing computer science program. The steps are: introducing Ada as a first programming language to permit teaching more modern programming concepts; adopting a broad overview of computer science in the beginning courses (as suggested in [Denning89]); creating a variety of software engineering elective courses in the junior and senior years; splitting the computer science curriculum into two tracks; and, finally, creating distinct programs.

Dick Weis [Weis89] describes efforts at his university to introduce a substantial amount of software engineering material throughout the undergraduate computer science curriculum. The material was selected partly in response to a study made by IBM (where Weis worked before joining the faculty) of the professional knowledge and skill deficiencies of computer science graduates, as perceived by software professionals and managers. The material provided increased emphasis on tasks across the entire software life cycle, more emphasis on analysis and design, large team projects, communications skills, and ethics and professionalism issues.

Stu Zweben [Zweben89] acknowledges that a large percentage of computer science graduates enter careers in software development and that computer science programs have a responsibility to teach appropriate software engineering concepts. He specifically mentions software testing and the role of design as topics that should

receive increased coverage from the beginning of the curriculum. Tool use is another concept deserving increased coverage, but the current lack of consensus on what tools to use and lack of availability of reasonably priced tools impede effective teaching of this topic. Zweben recommends an evolutionary approach that builds on the expected ACM/IEEE-CS curriculum guidelines.

The workshop participants then discussed their perceptions of the deficiencies in current computer science education. The discussion identified five areas for improvement.

- Faculty knowledge and attitudes

  The current faculty in computer science programs are in many cases naive about software engineering, about education, and about curriculum development. They tend to be narrowly specialized and do not see software engineering as intellectually acceptable.

- Institutional attitudes, objectives, and resources

  Academic institutions do not recognize that computer science and software engineering have different objectives. The reward structure does not recognize teaching and curriculum development. There are limited resources for building new expertise among faculty or new programs. Industrial organizations treat software engineering knowledge as proprietary, inhibiting cooperative research efforts with universities.

- The maturity of the software engineering discipline

  The discipline is still in search of basic principles. It still appears that software engineering is a craft requiring talent more than a profession requiring skill.

- Current computer science programs

  Computer science curricula tend to treat programming too informally and stress only small, throw-away programs. There is no treatment of the idea that the methods of programming-in-the-small do not scale up. There is no significant treatment of domain-specific software systems knowledge. There is no focus or integration of topics; many curricula still consist of one course in each of several computer science topic areas. There is a shortage of good educational materials for software engineering.

- Students

  The quality of students entering computer science is declining. The wrong students are choosing computing majors.

The workshop concluded with a discussion of possible tasks for the SEI that would advance the state of undergraduate software engineering education. These included:

- Attempt to change attitudes that computer science and software engineering are narrow or shallow fields.

- Examine the reports of the two ACM/IEEE-CS task forces to determine how well software engineering curricula would fit with their recommendations.
- Examine the curriculum recommendations of the BCS and IEE [BCS89] to determine if they are also applicable to United States universities.
- Develop more opportunities for faculty development. Find a way for computer science educators to experience a large team project.
- Look for opportunities to influence textbook publishers to seek better coverage of software engineering in undergraduate textbooks.
- Investigate and publicize opportunities for faculty enhancement, such as those funded by NSF.
- Provide teaching modules to bring software engineering material into undergraduate computer science programs (perhaps modeled after the SEI's graduate curriculum modules). Include discussions of adding software engineering topics to computer science courses at future SEI Educator Development Workshops.
- Provide "marketing" material, such as videotapes, to attract the best high school students to software engineering.
- Publicize the efforts, especially the successes, of schools to build software engineering programs and to attract good students.
- Provide a forum for university deans to interact with industry and government software managers, in order to begin building the collaboration needed for an educational infrastructure to meet a national need.

At the end of the workshop, there was nearly unanimous agreement that a widely distributed SEI report on undergraduate issues would be an important step in accelerating the development of high-quality undergraduate software engineering programs. It was suggested that the report address issues of the objectives and content of such a program, and it should describe both revolutionary and evolutionary approaches to the creation of these programs. The participants also agreed to serve as reviewers of SEI reports and other materials and as the nucleus of an informal advisory group for undergraduate issues.

Appendix 2. Bloom's Taxonomy of Educational Objectives

Bloom [Bloom56] has defined a *taxonomy of educational objectives* that describes several levels of knowledge, intellectual abilities, and skills that a student might derive from education. An adaptation of this taxonomy for software engineering is shown in Figure A2.1. This taxonomy can be used to help describe the objectives, and thus the style and depth of presentation, of a software engineering curriculum.

Evaluation: The student is able to make qualitative and quantitative judgments about the value of methods, processes, or artifacts. This includes the ability to evaluate conformance to a standard, and the ability to develop evaluation criteria as well as apply given criteria. The student can also recognize improvements that might be made to a method or process, and to suggest new tools or methods.

Synthesis: The student is able to combine elements or parts in such a way as to produce a pattern or structure that was not clearly there before. This includes the ability to produce a plan to accomplish a task such that the plan satisfies the requirements of the task, as well as the ability to construct an artifact. It also includes the ability to develop a set of abstract relations either to classify or to explain particular phenomena, and to deduce new propositions from a set of basic propositions or symbolic representations.

Analysis: The student can identify the constituent elements of a communication, artifact, or process, and can identify the hierarchies or other relationships among those elements. General organizational structures can be identified. Unstated assumptions can be recognized.

Application: The student is able to apply abstractions in particular and concrete situations. Technical principles, techniques, and methods can be remembered and applied. The mechanics of the use of appropriate tools have been mastered.

Comprehension: This is the lowest level of understanding. The student can make use of material or ideas without necessarily relating them to others or seeing the fullest implications. Comprehension can be demonstrated by rephrasing or translating information from one form of communication to another, by explaining or summarizing information, or by being able to extrapolate beyond the given situation.

Knowledge: The student learns terminology and facts. This can include knowledge of the existence and names of methods, classifications, abstractions, generalizations, and theories, but does not include any deep understanding of them. The student demonstrates this knowledge only by recalling information.

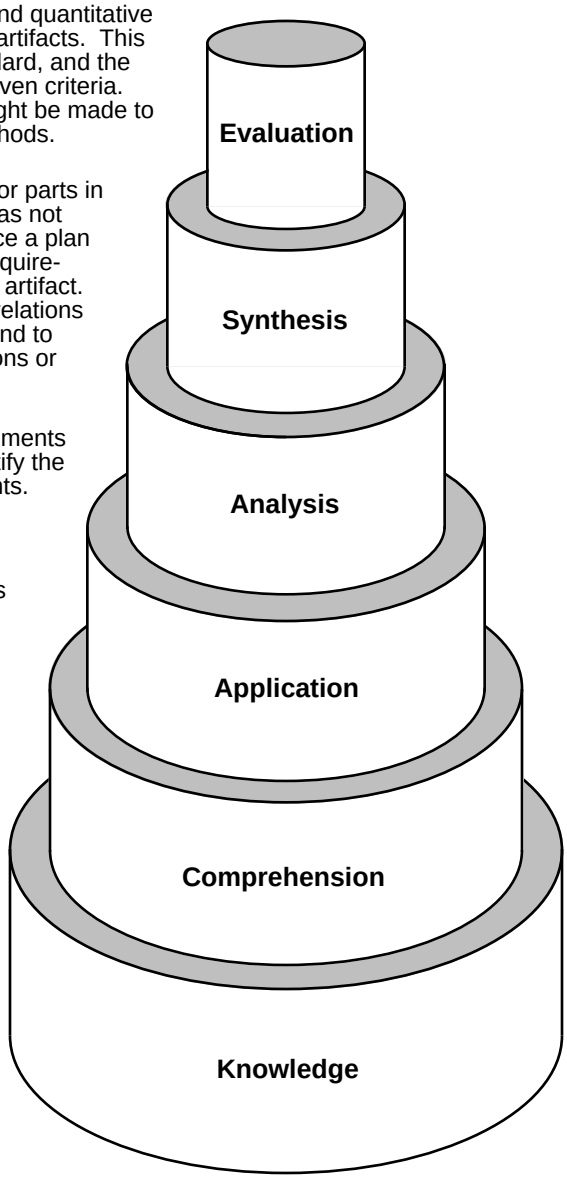

Figure A2.1. Bloom's taxonomy of educational objectives

Bibliography

ABET88 Accreditation Board for Engineering and Technology, Inc. *1988 Annual Report*. ABET, New York, Sept. 1988.

Ardis89 Ardis, M., and Ford, G. *1989 SEI Report on Graduate Software Engineering Education*. Tech. Rep. CMU/SEI-89-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., June 1989.

Barnes88 Barnes, B. H., Bjornson, J. D., Engle, G. L., Gear, C. W., Lewis, P. M., Miller, R. E., and Mulder, M. *Computer Science: The Discipline and the Profession*. Tech. Rep. SPC-TN-88-001, Software Productivity Consortium, Reston, Va., Feb. 1988.

Bauer72 Bauer, F. L. "Software Engineering." *Information Processing 71*. Amsterdam: North Holland, 1972.

BCS89 *A Report on Undergraduate Curricula for Software Engineering Curricula*. The British Computer Society and The Institution of Electrical Engineers, June 1989.

Bloom56 Bloom, B. *Taxonomy of Educational Objectives: Handbook I: Cognitive Domain*. New York: David McKay, 1956.

Cain86 Cain, J. T. "Professional Accreditation for the Computing Sciences." *IEEE Computer* (Jan. 1986), 91-96.

Congress89 U. S. Congress. House Committee on Science, Space, and Technology. *Bugs in the Program; Problems in Federal Government Computer Software Development and Regulation*. 101st Cong., 1st Sess., Serial G, Sept. 1989. Committee Print.

CSAB87 *Criteria for Accrediting Programs in Computer Science in the United States*. Computing Sciences Accreditation Board, Inc., New York, N. Y., Jan. 1987.

Culver82 Culver, R. S., and Hackos, J. T. "Perry's Model of Intellectual Development." *Engineering Education* (Dec. 1982).

Deimel89 Deimel, L. E. "Programming and Its Relation to Computer Science Education and Software Engineering Education." *Software Engineering Education*, Norman E. Gibbs, ed. New York: Springer-Verlag, 1989, 253-256.

Denning89 Denning, P. J., Comer, D. E., Gries, D., Mulder, M. C., Tucker, A., Turner, A. J., and Young, P. R. "Computing as a Discipline." *Comm. ACM 32*, 1 (Jan. 1989), 9-23.

Engle89a Engle, C. B., Jr., Ford, G., and Korson, T. *Software Maintenance Exercises for a Software Engineering Project Course*. Educational Materials CMU/SEI-89-EM-1, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., Feb. 1989. Includes distribution diskettes for software.

Engle89b — Engle, C. B., Jr. "Software Engineering is *Not* Computer Science." *Software Engineering Education*, Norman E. Gibbs, ed. New York: Springer-Verlag, 1989, 257-262.

Fairley85 — Fairley, R. *Software Engineering Concepts*. New York: McGraw-Hill, 1985.

Florman86 — Florman, S. C. "Toward Liberal Learning for Engineers." *Technology Review* (Feb./Mar. 1986), 18-25.

Ford87 — Ford, G., Gibbs, N., and Tomayko, J. *Software Engineering Education; An Interim Report from the Software Engineering Institute*. Technical Report CMU/SEI-87-TR-8, DTIC: ADA 182003, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., May 1987.

Ford89a — Ford, G. "Anticipating the Evolution of Undergraduate Software Engineering Curricula." *Software Engineering Education*, Norman E. Gibbs, ed. New York: Springer-Verlag, 1989, 263-266.

Ford89b — Ford, G. A., and Gibbs, N. E. "A Master of Software Engineering Curriculum; Recommendations from the Software Engineering Institute." *Computer 22*, 9 (Sept. 1989), 59-71.

Friedman89a — Friedman, F. L. "A Separate Undergraduate Software Engineering Curriculum Considered Harmful." *Software Engineering Education*, Norman E. Gibbs, ed. New York: Springer-Verlag, 1989, 267-270.

Friedman89b — Friedman, H. H., and Friedman, L. W. "Myths, Unethical Practices, Personnel Requirements: What Do Computer Industry Professionals Really Believe?" *J. Systems and Software 10*, 2 (Sept. 1989), 151-153.

Gibbs86 — Gibbs, N. E., and Tucker, A. B. "A Model Curriculum for a Liberal Arts Degree in Computer Science." *Comm. ACM 29*, 3 (Mar. 1986), 202-210.

Gibbs87 — *Software Engineering Education: The Educational Needs of the Software Community*. Norman E. Gibbs, Richard E. Fairley, eds. New York: Springer-Verlag, 1987.

Gibbs89a — Gibbs, N. E. "The SEI Education Program: The Challenge of Teaching Future Software Engineers." *Comm. ACM 32*, 5 (May 1989), 594-605.

Gibbs89b — *Software Engineering Education*. Norman E. Gibbs, ed. New York: Springer-Verlag, 1989. Proceedings of the 1989 SEI Conference on Software Engineering Education.

Gibbs89c — Gibbs, N. E. "Is the Time Right for an Undergraduate Software Engineering Degree?" *Software Engineering Education*, Norman E. Gibbs, ed. New York: Springer-Verlag, 1989, 271-274.

Gorgone89 Gorgone, J. T., and McGregor, J. D. "Computing Sciences Accreditation: A Cooperative Effort in CIS." *Computer Science Education 1,* 2 (1989), 99-110.

Gries89 Gries, D., and Marsh, D. "The 1987-1988 Taulbee Survey." *Comm. ACM 32,* 10 (Oct. 1989), 1217-1224.

Hopcroft87 Hopcroft, J. E. "Computer Science: The Emergence of a Discipline." *Comm. ACM 30,* 3 (Mar. 1987), 198-202. Transcription of the 1986 ACM Turing Award Lecture .

Humphrey89 Humphrey, W. S. *Managing the Software Process*. Reading, Mass.: Addison-Wesley, 1989.

IEEE83 *IEEE Standard Glossary of Software Engineering Terminology*. ANSI/IEEE Std. 729-1983, IEEE, 1983.

Jeffery77 Jeffery, S., and Linden, T. A. "Software Engineering is Engineering." *Proc. Computer Science and Engineering Curricula Workshop.* IEEE, June 1977, 112.

Jensen79a *Software Engineering.* Randall W. Jensen; Charles C. Tonies, eds. Englewood Cliffs, N. J.: Prentice-Hall, 1979.

Jensen79b Jensen, R. W., and Tonies, C. C. "Software Engineering Education: A Constructive Criticism." *Software Engineering*, Randall W. Jensen; Charles C. Tonies, eds. Englewood Cliffs, N. J.: Prentice-Hall, 1979, 553-567.

Jones89 Jones, L. G., and Richardson, W. E. "Software Engineering as Part of an Undergraduate Computer Science Program." *Software Engineering Education*, Norman E. Gibbs, ed. New York: Springer-Verlag, 1989, 275-279.

Kuo89 Kuo, F. F. "Let's make our best people into software engineers and not computer scientists." *Computer Decisions 1,* 2 (Nov. 1969), 94.

Lamb88 Lamb, D. A. *Software Engineering: An Emerging Profession?* External Tech. Rep. 88-233, Queen's University, Kingston, Ontario, Canada, Sept. 1988.

Lamb89 Lamb, D. A. "Questions in Planning Undergraduate Software Engineering." *Software Engineering Education*, Norman E. Gibbs, ed. New York: Springer-Verlag, 1989, 280-284.

Lasky89 Lasky, J. A. "Undergraduate Software Engineering Education: Prospects and Opportunities." *Software Engineering Education*, Norman E. Gibbs, ed. New York: Springer-Verlag, 1989, 285-288.

Lewis89 Lewis, P. M. "Information Systems is an Engineering Discipline." *Comm. ACM 32,* 9 (Sept. 1989), 1045-1047.

Lyall89 Lyall, J. R., and Agrawal, J. G. "Position Statement: Software Engineering Undergraduate Education." *Software Engineering Education*, Norman E. Gibbs, ed. New York: Springer-Verlag, 1989, 289-293.

Mills89 Mills, H. D., R.Basili, V., Gannon, J. d., and Hamlet, R. G. "Mathematical Principles for a First Course in Software Engineering." *IEEE Trans. Software Engineering 15*, 5 (May 1989), 550-559.

Nash89 Nash, J. C., and Nash, M. M. "Information Systems is a General Discipline." *Comm. ACM 32*, 12 (Dec. 1989), 1395.

Naur69 *Software Engineering; Report of a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 1968*. Peter Naur, Brian Randell, eds. Brussels, Belgium: NATO Science Committee, 1969.

Northrop89 Northrop, L. M. "Success with the Project-Intensive Model for an Undergraduate Software Engineering Course." *The Papers of the Twentieth SIGCSE Technical Symposium on Computer Science Education*, Robert A. Barrett, Maynard J. Mansfield, eds. New York: ACM, Feb. 1989, 151-155.

NRC85 National Research Council, Commission on Engineering and Technical Systems. *Engineering Education and Practice in the United States: Foundations of Our Techno-Economic Future*. Washington, D.C.: National Academy Press, 1985.

NSB86 NSB Task Committee on Undergraduate Science and Engineering Education. *Undergraduate Science, Mathematics and Engineering Education*. NSB 86-100, National Science Board, Washington, D.C., Mar. 1986.

NSF88 *Profiles–Computer Sciences: Human Resources and Funding.* NSF 88-324, National Science Foundation, Washington, D.C., 1988.

OTA89 U. S. Congress, Office of Technology Assessment. *Holding the Edge: Maintaining the Defense Technology Base.* OTA-ISC-420, U.S. Government Printing Office, Washington, D. C., Apr. 1989.

Parnas90 Parnas, D. L. "Education for Computing Professionals." *Computer 23*, 1 (Jan. 1990), 17-22.

Pennsylvania84 *Professional Engineers Registration Law, Act of 1945, P. L. 913, No. 367, Amended Jan. 1, 1984.* 1984.

Perry70 Perry, W. G., Jr. *Forms of Intellectual and Ethical Development in the College Years: A Scheme*. New York: Holt, Rinehart and Winston, 1970.

Preiss89 Preiss, R. J. "Computer security practitioner certification plan." *Computer 22*, 9 (Sept. 1989), 77-78.

Richardson88 Richardson, W. E. "Undergraduate Software Engineering Education." *Software Engineering Education*, Gary A. Ford, ed. New York: Springer-Verlag, 1988, 121-144.

Schein72 Schein, E. H., and Kommers, D. W. *Professional Education: Some New Directions*. New York: McGraw-Hill, 1972.

Shaw85 *The Carnegie-Mellon Curriculum for Undergraduate Computer Science*. Mary Shaw, ed. New York: Springer-Verlag, 1985.

Shaw89 Shaw, M. *Software and Some Lessons from Engineering*. Videotape TECH-MS-01-01, Software Engineering Institute, Pittsburgh, Pa., Mar. 1989.

Tomayko87 Tomayko, J. E. *Teaching a Project-Intensive Introduction to Software Engineering*. Tech. Rep. CMU/SEI-87-TR-20, DTIC: ADA 200603, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., Mar. 1987.

Tomayko89 Tomayko, J. E. "Is Software Engineering Graduate-Level Material?" *J. Systems and Software 10*, 4 (Nov. 1989), 231-234.

US84 *Code of Federal Regulations*. 1984.

VanScoy89 Van Scoy, F. L. "Developing an Undergraduate Software Engineering Curriculum within an Existing Computer Science Program." *Software Engineering Education*, Norman E. Gibbs, ed. New York: Springer-Verlag, 1989, 294-303.

Weinberg84 Weinberg, D., and Weinberg, G. "Constructing New Knowledge: The Experiential Learning Model." *Data Training* (Nov. 1984), 26-28.

Weis89 Weis, R. L. "Software Engineering in a BS in Computer Science." *Software Engineering Education*, Norman E. Gibbs, ed. New York: Springer-Verlag, 1989, 304-309.

Wilson89 Wilson, G. L. "Designing a Better Engineer." *Technology Review* (1989), 3,13.

Zweben89 Zweben, S. H. "Integrating Software Engineering into an Undergraduate Computer Science Curriculum." *Software Engineering Education*, Norman E. Gibbs, ed. New York: Springer-Verlag, 1989, 310-312.