[redacted]

Carnegie-Mellon University
Software Engineering Institute

[redacted]

by

Gary Ford
Software Engineering Institute
James Tomayko
Software Engineering Institute

May 1987



[redacted]

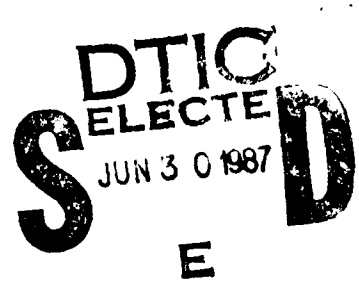
2

**Technical Report**
**SEI-87-TR-8**
**ESD-TR-87-109**
**May 1987**

# Software Engineering Education
# An Interim Report from the Software Engineering Institute



**Gary Ford**
**Norman Gibbs**
**James Tomayko**




**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

# Software Engineering Education

## An Interim Report from the Software Engineering Institute[1]

Gary Ford, Norman Gibbs, James Tomayko

*Abstract:* The goals and activities of the Software Engineering Institute's Education Program are described. Two curriculum recommendations are presented, one for a professional Master of Software Engineering degree program, and the other for an undergraduate project course in software engineering. Also presented is an organizational structure for software engineering curriculum content.

## The SEI Education Program

The Software Engineering Institute is a federally funded research and development center operated by Carnegie-Mellon University. Its principal responsibility is to accelerate the reduction to practice of modern software engineering techniques and methods (see [DOD85, Barbacci85]). Included in this responsibility are the identification, assessment, development, dissemination, and insertion of promising methods, techniques, and tools to support software engineering.

Recognizing that education is the foundation for substantial improvements in developing and using technology, the SEI charter also includes the sentence, "[The SEI] shall also influence software engineering curricula development throughout the education community." Our experiences to date indicate that the education community is very interested in software engineering education, and that the SEI can play an important role in focusing activities in the development of courses and curricula, in catalyzing the production of textbooks, educational software, and other course support materials, and in providing for widespread distribution of information and materials.

The SEI Education Program is one of the original programs at the Institute. It has broad responsibilities, based on the sentence quoted above. Those include identifying the educational needs of the software engineering community, providing leadership to meet those needs, and working directly with all interested parties in disseminating information and materials that address those needs.

The Education Program began its efforts in the summer of 1985 with two staff members. We plan to grow to approximately fourteen technical staff by 1990, with a corresponding growth of the scope of our activities. An earlier report [Gibbs86b] detailed the challenges of software engineering education and presented our strategy for meeting them. This report contains our first specific recommendations, but they should be considered preliminary, with more refined and more detailed recommendations appearing later. Toward that end, we solicit comments on this report from the software engineering and education communities.

The report is organized into five major parts. The first describes the scope and goals of the SEI Graduate Curriculum Project. Next is a description of an organizational structure for the content of a software engineering curriculum. The third and fourth parts are the curriculum recommendations for a Master of Software Engineering degree and for a one semester undergraduate project course. The final part contains some thoughts on the future of software engineering education.

## The Graduate Curriculum Project

The SEI Education Program encompasses several projects and activities. Currently, our major effort is the Graduate Curriculum Project, which is identifying and documenting the body of knowledge appropriate for master's level programs in software engineering. The project is also developing a wide range of educational materials that will support educators and students in such programs. The individual aspects of the project are discussed below.

We are not the first to attempt to define a curriculum for software engineering. Several others have made recommendations, and reading them in chronological order provides an interesting history of the growth of the discipline [Ardis85, Comer86, Fairley78, Fairley79a, Fairley79b,

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

Fairley80, Freeman76, Freeman78, Hoffman78a, Hoffman78b, Jensen78, Jensen79, Lehman86a, McGill84, Mills86, Mulder75, Nance80, Stucki78, Warner82, Wasserman76]. However, our goal is not just to propose another curriculum, but to create and *support* a *dynamic* curriculum for a dynamic discipline. We rely on the innovators in software engineering to point the direction (see [Brooks86, Goldberg86, Lehman86b]), and on a large number of persons outside the SEI, including software engineers and educators, to develop the curriculum and support materials and to revise them to reflect the state of the art of software engineering.

We also understand that there is anything but concensus on what software engineering education should encompass. We find ourselves in the same situation as the developers of computer science curricula in the 1960s. In his 1986 ACM Turing Award Lecture, John Hopcroft remembered his arrival at Princeton University in the fall of 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.

For many of the courses recommended in this report, we cannot now clearly delineate their boundaries. We believe that teaching them now is valuable to today's students and will help improve them for tomorrow's students.

## Curriculum Modules

The audience for software engineering education is large and diverse, including students at both undergraduate and graduate levels, colleges and universities whose offerings range from a part of one course to an entire degree program in software engineering, and practitioners in industry and government. Each audience needs a different curriculum. Therefore, we made an early decision to define the content of software engineering education in *modules*. Each module presents a relatively small and highly focused topic. Modules vary in size, but most are smaller than a typical university course. Courses and programs can then be constructed in many ways from the modules.

We intend for a module to be a resource for an instructor who is designing or revising a course. With this goal in mind, we have structured modules to contain most of the information that an instructor would gather in the preliminary stages of course design, including a bibliography and a topic outline. By providing this information, we hope that instructors find it possible to develop better courses in much shorter time.

A module is embodied in a document of approximately 20 pages that begins with a capsule description of the material (similar to a college catalog description), and brief discussions of the philosophy of the module, the prerequisites, and the educational objectives. The module content is then described in two forms, a brief topic outline (similar to a course syllabus), and a detailed, annotated outline, with references to the appropriate literature. Teaching considerations are described, including such things as suggestions for exercises, projects, or exams, the amount of time to spend on each topic, appropriate textbooks, and suggestions for support materials. An annotated bibliography completes the module, and includes both a brief synopsis of the content of each reference and an indication of how the work might be used by the instructor and/or the student.

The Education Program has a small permanent staff, so we have relied on visiting educators and software engineers to develop the curriculum modules (see *SEI Affiliate Programs* on page 22). During 1986, ten visiting educators spent periods of from one week to several months at the SEI, mostly during the summer. They researched module topics, organized the material, and shared their knowledge and expertise with us and with each other. Five modules were completed and published [Budgen86b, Cohen86, Collofello86, Jorgensen86, Tomayko86a].

Curriculum modules are considered to be living documents. Each is subject to continual review, both inside and outside the SEI, and to revision. Especially useful are the experiences of educators who have taught from them, and these educators will often be invited to come to the SEI to prepare revisions. It is our expectation that each module will represent the best current ideas of the software engineering community, and that the set of modules as a whole will be a state-of-the-art snapshot of the discipline.

## Curriculum Packages

Different audiences need different curriculum content. The documentation of curriculum content in modules facilitates creation of many different curriculum *packages*. We have identified two important packages for immediate develop-

ment: a one semester course in software engineering at the senior year level in a computer science degree program, and a complete professional degree program at the master's level. Each of these is presented in detail later in this report.

These packages were targeted first because they represent the two ends of the most likely spectrum of university offerings in software engineering. Additional planned packages are intermediate points on that spectrum, and include two semester sequences (especially project oriented courses) and concentrations of four to six courses for specialty tracks within a master of science program in computer science. This latter package is presently being developed and tested in cooperation with The Wichita State University, which is the SEI's first designated *graduate curriculum test site*. We also expect to develop packages of individual or groups of modules for industry short courses. Some modules have already been tested successfully in this form.

## Support Materials

The quality of a course depends on the quality of the support materials available as well as the quality of the instructor. Textbooks and other reference materials are obvious examples of support materials, but software engineering education also depends on projects and exercises, software tools and programming environments, and examples of large-scale software systems and the processes that produce them. The SEI has undertaken to produce or cause the production of a wide range of such materials.

We have an agreement with Addison-Wesley Publishing Company to produce *The SEI Series on Software Engineering*, a series of monographs and textbooks on all aspects of software engineering. The series is under control of an editorial board of leaders in the field, the majority of whom are software engineers from industry rather than university faculty members. The first books in the series are expected in 1988. Authors are chosen from throughout the software engineering community, and we intend to work with them to capture their expertise in appropriate curriculum modules as well as in the books.

Both developers and users of curriculum modules are encouraged to share with the SEI the materials that they and their students produce for software engineering courses. One package of support materials has been published [Tomayko86b]. As with curriculum modules, we work with visiting staff for the production of support materials and welcome proposals for such projects.

The SEI is also identifying and collecting large, production-quality artifacts from industry. We believe that a delivered, working software system of perhaps 50,000 lines of code, along with all requirements specifications, design documentation, test plans and data, and user documentation would make an unusually valuable object of study for software engineering students. It could also serve as the basis for software testing or enhancement projects on a scale almost never before possible in universities.

## Educational Software Tools

The software engineering community has long recognized the utility of software systems to help with the daily activities of clerical and data processing workers. More recently, many systems have begun to appear that help with the daily activities of software engineers. The days when a text editor and a compiler made up a complete tool set are long gone.

The Wang Institute of Graduate Studies offered a Master of Software Engineering degree in which more than one hundred software tools, almost all of which are commercially available, were used by its students in their course work. Major software development organizations in industry often have many more, usually proprietary products developed for their own use. Because this is the kind of environment in which today's students will soon find themselves, it is important that tool usage be included in the curriculum.

Many of the modern tools run on relatively expensive hardware systems, often beyond the resources of most universities in the quantities required to accommodate large numbers of students. Therefore, the SEI Education Program is planning to develop, again with the help of visiting educators and their students, a set of software tools that have two important properties. First, the tools will run on the kinds of hardware configurations that are commonly available in universities. Second, the tools will demonstrate the major features, if not the full functionality, of the kinds of tools that students are likely to use as practicing software engineers. Because the SEI is in the business of identifying, assessing, and developing promising new tools, we are ideally positioned to produce limited versions of new tools for education almost as fast as the full tools appear in industry.

A second variety of tools includes those that are specific to a single project, such as test case generators and simulators to aid in requirements analysis. It is beyond the abilities of individual instructors to build such tools for each student project, so we see a role here for the SEI. Rather than

attempting to build generalized versions of such tools, we plan to define possible student projects, develop appropriate support materials for each project, including tools, and offer these to educators for their use.

## A Curriculum Content Organizational Structure

The body of knowledge called *software engineering* consists of a large number of interrelated topics. We thought it impractical to attempt to capture this knowledge as an undifferentiated mass, so an organizational structure was needed. The structure described below *is not intended to be a taxonomy of software engineering*, but rather a guide for the SEI in collecting and documenting software engineering knowledge, and for describing the content of some recommended courses for a graduate curriculum. (Others have considered the organization of software engineering topics; for example, see [Babb79, AFIPS80, IEEE86].)

Discussions of software engineering frequently describe the discipline in terms of a software life cycle: requirements analysis, specification, design, implementation, testing, and maintenance. Although these life cycle phases are worthy of presentation in a curriculum, we found this one-dimensional structure inadequate for purposes of organizing all the topics in software engineering, and for describing the curriculum.

A good course, whether a semester course in a university or a one day short course in industry, must have a central thread or idea around which the presentation is focused. Not every course can or should focus on one life cycle phase. In an engineering course (including software engineering), we can look either at the engineering *process* or at the *product* that is the result of the process. Therefore we have chosen these two views as the highest level partition of the curriculum content. Each is elaborated below.

### The Process View

The process of software engineering includes several *activities* that are performed by software engineers. The range of activities is broad, but there are many *aspects* of each activity that are similar across the range. Thus we organize those topics whose central thread is the process in two dimensions: activity and aspect.

**The Activity Dimension.** Activities are divided into four groups: *development, control, management, and operational*. Each is defined and discussed below.

*Development* activities are those that create or produce the artifacts of a software system. These include requirements analysis, specification, design, implementation, and testing. Because a software system is usually part of a larger system, we sometimes will distinguish system activities from software activities; for example, *system design* from *software design*. We expect that many large projects will include both systems engineers and software engineers, but an appreciation of the systems aspects of the project is important for software engineers, and thus should be included in a curriculum.

*Control* activities are those that exercise restraining or directing influence over software development. These activities are more concerned with controlling the way in which the development activities are done than with the production of artifacts. Two major kinds of control activities are those related to software evolution and those related to software quality.

A software product evolves in the sense that it exists in many different forms as it moves through its life cycle, from initial concept, through development and use, to eventual retirement. Change control and configuration management are activities related to evolution. We also consider software maintenance in this category, rather than as a separate development activity, because the difference between development and maintenance is not in the activities performed (both involve requirements analysis, specification, design, implementation, and testing), but in the way those activities are controlled.

Software quality activities include quality assurance, test and evaluation (T&E), and independent verification and validation (IV&V). These activities, in turn, incorporate such tasks as software technical reviews and performance evaluation.

*Management* activities are those involving executive, administrative, and supervisory direction of a software project, including technical activities that support the executive decision process. Typical management activities are project planning (schedules, establishment of milestones), resource allocation (staffing, budget), development team organization, cost estimation, and legal aspects (contracting, licensing). This is an appropriate part of a software engineering curriculum for several reasons: there is a body of knowledge about managing software projects that is different from that about managing other kinds of projects, many software engineers are likely to assume software

management positions at some point in their careers, and knowledge of this material by all software engineers improves their ability to work together as a team on large projects.

*Operational* activities are those related to the use of a software system by an organization. These include training personnel to use the system, planning for the delivery and installation of the system, the transition from the old (manual or automated) system to the new, operation of the software, and retirement of the system. Although a software engineer may not have primary responsibility for any of these activities, an awareness of these activities will often impact the development of a software system, and software engineers are often parts of teams that perform these activities.

Software engineering support tools provide a case of special interest. These tools are software systems, and the users of those systems are the software engineers themselves. Operational activities for these systems can be observed and experienced directly. An awareness of the issues related to the use of software tools can not only help software engineers develop systems for others but can also help them adopt and use new tools for their own activities.

**The Aspect Dimension.** Engineering activities traditionally have been partitioned into *analytic* activities and *synthetic* activities. We have chosen instead to consider an axis orthogonal to activities that captures some of this kind of distinction, but that recognizes six *aspects* of these activities: *abstractions, representations, methods, tools, assessment,* and *communication.*

*Abstractions* include fundamental principles and formal models. For example, software development process models (waterfall, iterative enhancement, etc.) are models of software evolution. Finite state machines and Petri nets are models of sequential and concurrent computation, respectively. COCOMO is a software cost estimation model. Modularity and information hiding are principles of software design.

*Representations* include notations and languages. The Ada®[2] language thus fits into the organization as an implementation language, while decision tables and dataflow diagrams are design notations. PERT charts are a notation useful in project planning.

*Methods* include formal methods, current practices, and methodologies. Proofs of correctness are examples of formal methods for verification. Object-oriented design is a design method, and structured programming may be considered a current practice of implementation.

*Tools* include individual software tools as well as integrated tool sets (and, implicitly, the hardware systems on which they run). Examples include general purpose tools such as electronic mail and word processing, tools related to design and implementation, such as compilers and syntax directed editors, and project management tools. Other kinds of software support for process activities are also included; these are sometimes described by such terms as *infrastructure, scaffolding,* or *harnesses.*

Sometimes the term *environment* is used to describe a set of tools, but we prefer to reserve this term to mean a collection of related representations, tools, methods, and objects. Software objects are abstract, so we can only manipulate representations of them. Tools to perform manipulations are usually designed to help automate a particular method or way of accomplishing a task. Typical tasks involve many objects (code modules, requirements specification, test data sets, etc.), so those objects must be available to the tools. Thus we believe all four parts are necessary for an environment.

*Assessment* aspects include measurement, analysis, and evaluation of both software products, software processes, and the impact of software on organizations. Metrics and standards are also placed in this category. This is an area where we feel considerable emphasis is needed in the curriculum. Software engineers, like engineers in the traditional fields, need to know what to measure, how to measure it, and how to use the results to analyze, evaluate, and ultimately improve processes and products.

*Communication* is the final aspect. All software engineering activities include written and oral communication. Most produce documentation. A software engineer must have good general technical communication skills, as well as an understanding of forms of documentation appropriate for each activity.

By considering the activity dimension and the aspect dimension as orthogonal, we have a matrix of ideas that might serve as the central thread in a course. It is likely that individual cells in that matrix represent too specialized a topic for a full semester course. Therefore we recommend that courses be designed around part or all of a horizontal or vertical slice through that matrix.

[2]Ada is a registered trademark of the U.S. government, Ada Joint Program Office.

## The Product View

Often it is appropriate to discuss many activities and aspects in the context of a particular kind of software system. For example, concurrent programming has a variety of notations for specification, design, and implementation that are not needed in sequential programming. Instead of inserting one segment or lecture on concurrent programming in each of several courses, it is probably better to gather all the appropriate information on concurrent programming into one course. A similar argument can be made for information related to various system requirements; for example, achieving system robustness involves aspects of requirements definition, specification, design, and testing.

Therefore we have added two additional categories to the curriculum content organizational structure, *software system classes* and *pervasive system requirements*. Although these may be viewed as being dimensions orthogonal to the activity and aspect dimensions, it is not necessarily the case that every point in the resulting four-dimensional space represents a topic for which there exists a body of knowledge, or for which a course should be taught.

Any of the various system classes or pervasive requirements described below might be the central thread in a course in a software engineering curriculum. We emphasize that the material taught might also be taught in courses whose central thread is one of the activities mentioned earlier. For example, techniques for designing real-time systems could be taught in a design course or in a real-time systems course. Testing methods to achieve system robustness could be taught in a testing course or in a robustness course. The purpose of adding these two new dimensions to the structure is to allow better descriptions of possible courses.

**Software System Classes.** Several different classes can be considered. One group of classes is defined in terms of a system's relationship to its environment, and has members described by terms such as *batch*, *interactive*, *reactive*, *real-time*, and *embedded*. Another group has members described by terms such as *distributed*, *concurrent*, or *network*. Another is defined in terms of internal characteristics, such as *table-driven*, *process-driven*, or *knowledge-based*. We also include generic or specific applications areas, such as *avionics systems*, *communications systems*, *operating systems*, or *database systems*.

Clearly, these classes are not disjoint. Each class is composed of members that have certain common characteristics, and there is or may be a body of knowledge that directly addresses the development of systems with those characteristics. Thus each class may be the central theme in a software engineering course.

**Pervasive System Requirements.** Discussions of system requirements generally focus on functional requirements. There are many other categories of requirements that also deserve attention. Identifying and then meeting those requirements is the result of many activities performed throughout the software engineering process. As with system classes, it may be appropriate to choose one of these requirement categories as the central thread for a course, and then to examine those activities and aspects that affect it.

Examples of pervasive system requirements are *accessibility*, *adaptability*, *availability*, *compatibility*, *correctness*, *efficiency*, *fault tolerance*, *integrity*, *interoperability*, *maintainability*, *performance*, *portability*, *protection*, *reliability*, *reusability*, *robustness*, *safety*, *security*, *testability*, and *usability*. Definitions of these terms may be found in the ANSI/IEEE Glossary of Software Engineering Terminology [IEEE83].

## Educational Objectives

An additional refinement of the curriculum content can be achieved by considering educational objectives. Modest objectives can be achieved by a superficial presentation, while more ambitious objectives will require greater breadth or depth (or both) in the presentation of a topic. A clear statement of the educational objectives of a course is valuable to the instructor, both for choosing what material to present and how to present it.

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. We found it useful to adapt this taxonomy to help describe the objectives, and thus the style and depth of presentation, of a software engineering curriculum.

The six classes of objectives below are presented in increasing order of difficulty, in that each requires education beyond the previous ones for the student to achieve the objective.

**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.

**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.

**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.

**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.

**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.

**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.

Because the body of knowledge called software engineering is very large, it is unreasonable to expect a software engineer to know all of it, even at the lowest or knowledge level, as defined above. On the other hand, some material must be known at the synthesis level if the engineer is to be productive. The design of a curriculum, therefore, must identify carefully the overall educational objectives, as well as the objectives of each course. The backgrounds, capabilities, and career goals of the students, the faculty and equipment resources, and the amount of time available are all factors to be considered in defining objectives.

In the following section, we invoke this taxonomy in the description of the curriculum. Knowing the objectives of a course often helps guide the choice of teaching methods and student exercises. Most educators agree that realistic software engineering experiences are often the best mechanism for achieving the higher levels of objectives. In a later section, we discuss examples of how to provide such experiences.

## A Master of Software Engineering Curriculum

Throughout the short history of computer science and software engineering, the size and complexity of software systems have grown steadily. Advances in hardware and software technology have made more ambitious systems feasible, but each significant increase in system size has presented software engineers with new problems to be solved. For example, in the 1950s the system bottleneck was often getting access to the machine for a batch run. Operating systems with job schedulers, foreground and background processing, and timesharing helped solve that problem. Coding bottlenecks were alleviated by the development of higher-level programming languages. Structured programming and data abstraction helped simplify software design problems. Microprocessors led to more complex embedded systems.

With each advance came the need for increased education of computer scientists and software engineers. We are now at a point where there is a substantial body of useful knowledge about programming-in-the-small. A software engineer needs much of this knowledge before beginning serious study of programming-in-the-large, meaning the controlled development of large, complex systems by teams of developers. A good undergraduate program in computer science provides the necessary prerequisite knowledge for a software engineering degree program. Thus it is presently inappropriate to try to develop an *undergraduate* degree program in software engineering. We believe that for the immediate future, software engineering education should be at the master's level.

The academic community distinguishes two master's level technical degrees. The Master of Science in *Discipline* is a

research-oriented degree, and often leads to doctoral level study. The Master of *Discipline* is a terminal professional degree intended to produce a practitioner who can rapidly assume a position of substantial responsibility in an organization. The former degree often requires a thesis, while the latter requires a project or practicum as a demonstration of the level of knowledge acquired. The Master of Business Administration (MBA) degree is perhaps the most widely recognized example of such a professional degree.

The SEI was chartered partly in response to the perceived need for a greatly increased number of highly skilled software engineers. It is our belief that this need can be best addressed by encouraging and helping academic institutions offer a Master of Software Engineering (MSE) degree program.

## Objectives

As described above, the goal of the MSE degree is to produce a software engineer who can rapidly assume a position of substantial responsibility within an organization. To achieve this goal, the curriculum we propose is designed to give the student a body of knowledge that includes balanced coverage of the software engineering process activities, their aspects, and the products they produce, and to give the student sufficient experience to bridge the gap between undergraduate programming and professional software engineering.

Specific educational objectives are summarized below, and may be found in greater detail in the descriptions of individual curriculum units in the Curriculum Content section beginning on page 11.

**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 whatever techniques they learned in school, they will encounter more in their professional careers.

**Comprehension.** The students 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 a reasonable set of principles, models, representations, methods, and tools, and the role of analysis and evaluation in software engineering. They should understand the existing design paradigms for well-understood systems, such as compilers. 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.

**Application.** The students should be able to apply fundamental principles in the performance of the various activities. They should be able to apply a reasonable set of formal methods to achieve results. They should be able to use a reasonable set of 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.** The 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.** The 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.** The 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 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 activites should consider all system requirements, not just functional and performance requirements. They 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 words *appropriate* and *reasonable* occur several times in the objectives above. The software engineering discipline is new and changing, and there is not a concensus on the best set of representations, methods, or tools to use. Each implementation of the MSE curriculum must be structured to match the goals and resources of the school and its students. In subsequent reports, the SEI will offer recommendations on the most promising methods and technologies for many of the software engineering activities.

## Prerequisites

An undergraduate degree in computer science is the most desirable prerequisite for the MSE degree (see [Austing78, Koffman84, Koffman85, Gibbs86a] for models of computer science programs). We recognize that most practitioners do not have such a degree but still wish to pursue the MSE degree. Furthermore, students with a bachelor's degree in computer science from different schools, or from the same school but five years apart, are likely to have substantially different knowledge. Thus the prerequisites for the MSE degree must be defined carefully, and must be enforceable and enforced.

The primary prerequisite, therefore, is substantial knowledge of programming-in-the-small. This includes a working knowledge of at least one modern, high-level language (for example, Pascal, Modula-2, Ada) and at least one assembly language. Also important is a knowledge of fundamental concepts of programming, including control and data structures, modularity, data abstraction and information hiding, and language implementations (runtime environments, procedure linkage, and memory management). Students should also be familiar with the *tools of the trade*, meaning a user's knowledge (not a designer's knowledge) of computer organization and architecture, operating systems, and typical software tools (editor, assembler, compiler, linking loader, etc.). An appreciation for formal methods and models is also essential, including analysis of algorithms and the fundamentals of computability, automata, and formal languages. Most or all of this material is likely to be found in the first three years of an undergraduate computer science degree program.

Knowledge of one or more other major areas of computer science is highly desirable, but not absolutely necessary. Examples are functional and declarative languages, numerical methods, database systems, compiler construction, computer graphics, or artificial intelligence. This material is usually found in senior level electives in a computer science degree program. Some schools may choose to allow advanced computer science courses as electives in the MSE program. Knowledge of major applications areas in the sciences and engineering may also be useful.

The mathematics prerequisites are those areas commonly required in an undergraduate computer science degree: discrete mathematics and some calculus. Some software engineering topics may require additional mathematical prerequisites, such as probability and statistics. A student planning a career in a particular application area may want additional mathematics, such as linear algebra or differential equations, but these are not essential prerequisites for any of the mainstream software engineering courses.

Enforcing the prerequisites can be difficult. A lesson may be learned from experience with master's degree programs in computer science. For many years in the 1960s and 1970s, these programs often served almost exclusively as retraining programs for students with undergraduate degrees in other fields (notably mathematics and engineering), rather than as advanced degree programs for students who already had an undergraduate computer science degree. In several schools, undergraduate computer science majors were not eligible for the master's program because they had already taken all or nearly all of the courses as undergraduates.

These programs existed because there was a clearly visible need for more programmers and computer scientists, and the applicants for these programs did not want a second bachelor's degree. There were not enough applicants who already had a computer science degree to permit enforcement of substantial prerequisities.

For the proposed MSE program to achieve its goals, it must take students a great distance beyond the undergraduate computer science degree. This, in turn, requires that students entering the program have approximately that level of knowledge. Because of the widely varying backgrounds of potential students, this is very difficult to assess. Standardized examinations, such as the Graduate Record Examination in Computer Science, provide only part of the solution.

We recommend that schools wishing to establish the MSE program consider instituting a *leveling* or *immigration* course to help establish prerequisite knowledge. Such a course almost never fits into the normal school calendar. Rather it is an intensive two to four week course that is scheduled just before or just after the start of the normal

school year. Students receive up to 20 hours a week of lectures summarizing all of the prerequisite material. The value of this course is not that the students become proficient in all the material, but that they become aware of deficiencies in their own preparation. Self-study in parallel with the first semester's courses can often remove most of these deficiencies.

Another important part of the immigration course is the introduction of the computing facilities, especially the available software tools, to students with varying backgrounds. Ten to 20 hours each week can be devoted to demonstrations and practice sessions. Because proficiency with tools can greatly increase the productivity of the students in later courses, the time spent in the immigration course can be of enormous value.

Finally, the immigration course can be used to help motivate the study of software engineering. The faculty, and sometimes the students themselves, can present some of their own or others' experiences that led to improved understanding of some of the significant problems of software engineering or their solutions.

The Computer Science Department at Carnegie-Mellon University has a very successful immigration course for its doctoral program, which may be useful as a model for other schools. It is described in [CMU80] and [Shaw73]. A current course schedule may be requested from the Department.

Another kind of prerequisite has been adopted by four existing MSE programs (at the College of St. Thomas, Seattle University, Texas Christian University, the and Wang Institute of Graduate Studies[3]) All four require the student to have at least one year of professional experience as a software developer. This requirement has the benefit of giving the students increased motivation for software engineering, since it exposes them to the problems of developing systems that are much larger than those seen in the university, and exposes them to economic and technical constraints on the software development process. It is also the case that schools cannot control the quality of that experience, and students may acquire bad habits that must be unlearned.

We have not found the arguments for an experience prerequisite sufficiently compelling to recommend it for all MSE programs. Other engineering disciplines have successful master's level programs, and even undergraduate programs, without such a prerequisite. Most graduate professional degrees do not require it.

As a discipline grows and evolves, it is a common phenomenon in education for new material to be incorporated into courses that are added to the end of an existing curriculum. Over time, the new material is assimilated into the curriculum in a process called *curricula compression.* Obsolete material is taken out of the curriculum, but much of the compression is accomplished by reorganization of material to get the most value in the given amount of time.

In a rapidly growing and changing discipline, new material is added faster than curricular compression can accommodate it. In some engineering disciplines the problem is acute. There is a growing sentiment that the educational requirement for an entry level position in engineering should be a master's degree or a five-year undergraduate degree [NRC85]. This is especially true for a computer science/software engineering career.

If this level of education is needed for a meaningful entry level position, then we question the value of sending students out with a bachelor's degree, hoping they will return sometime later for a software engineering degree. The professional experience achieved during that time will not necessarily be significant. Also, the percentage of students intending to return to school who actually do return to school declines rapidly as time since graduation increases. Therefore we believe that an MSE curriculum structured to follow immediately after a good undergraduate curriculum will offer the best chance of achieving the goals of rapid increases in the quality and quantity of software engineers. Of course, such a program does not preclude admission of students with professional experience.

We do recognize that work experience can be valuable. The experience component of the MSE curriculum, which is discussed later in this report, might be structured to include actual work experience. It may be that the overall educational experience is significantly enhanced if the work component is a coordinated part of the program, rather than an interlude between undergraduate and graduate studies.

We also recognize that we must motivate many of the activities in the software engineering process. We see a great need to raise the level of awareness on the part of both students and educators of the differences between undergraduate programming and professional software engineering. The SEI Education Program is working at the undergraduate level to help accomplish this. The undergraduate

[3]As of April 1, 1987, the Wang Institute became part of Boston University, and its MSE program will be terminated at the end of August, 1987.

software engineering course described in this report is one example.

## Curriculum Content

We use a broad view of software engineering when choosing the content of the curriculum, and include several topics that are not part of a typical engineering curriculum. This statement of the National Research Council about engineering curricula reflects the views of many engineers and educators [NRC85]:

> Another element of the problem is that to make the transition from high school graduate to a competent practicing engineer requires more than just the acquisition of technical skills and knowledge. It also requires a complex set of communication, group-interaction, management, and work-orientation skills.
>
> ... For example, education for management of the engineering function (as distinct from MBA-style management) is notably lacking in most curricula. Essential nontechnical skills such as written and oral communication, planning, and technical project management (including management of the individual's own work and career) are not sufficiently emphasized.

On the other hand, we have narrowed the curriculum by concentrating exclusively on *software* engineering (including some aspects of *system* engineering) and omitting applications area knowledge. Two major reasons for this is are pragmatic: first, the body of knowledge known as software engineering is sufficiently large to require all the available time in a typical master's degree program (and then some), and second, students cannot study all of the applications areas in which they might eventually work. We believe that a student at the graduate level should have acquired the skills for self-education that will allow acquisition of some knowledge in a needed application area.

More important, however, is our strong belief that the variety of applications areas and the level of sophistication of hardware and software systems in each of those areas mandate a development *team* with a substantial range of knowledge and skills among its members. Some members of the team must understand the capabilities of hardware and software components of a system in order to do the highest level specification, while other members must have the skills to design and develop individual components. Software engineers will have responsibility for software components just as electrical, mechanical, or aeronautical engineers, for example, will have responsibility for the hardware components. Scientists, including computer scientists, will often also be needed on the development teams, and all the scientists and engineers must be able to work together toward a common goal.

The content of the MSE curriculum is described in *units*, each covering a major topic area, rather than courses. There are three reasons for this. First, not every topic area contains enough material for a typical university course. Second, combining units into courses can be accomplished in different ways for different organizations. (An example that maps the units into semester courses follows the description of the curriculum content.) Third, this structure more easily allows each unit to evolve to reflect the changes in software engineering knowledge and practice, while maintaining the stability of the overall curriculum structure.

Because of strong relationships among topics and subtopics, we were unable to find a concensus on an appropriate linear presentation order of topics. We do, however, recommend a top-down approach that focuses on the software engineering process first because this overall view is needed to put the individual activities in context. Software management and control activities are presented next, followed by the development activities and product view topics.

This approach is similar to the common pedagogical technique known as the *spiral approach*, in which interrelated topics are presented repeatedly in increasing depth. Viewed this way, there are three relatively distinct levels in the curriculum:

| | |
|---|---|
| Level 1 | The Software Engineering Process<br>Software Evolution<br>Software Generation<br>Software Maintenance<br>Technical Communication |
| Level 2 | Software Configuration Management<br>Software Quality Issues<br>Software Quality Assurance<br>Software Project Organizational and Management Issues<br>Software Project Economics<br>Software Operational Issues |
| Level 3 | Requirements Analysis<br>Specification<br>System Design<br>Software Design<br>Software Implementation<br>Software Testing<br>System Integration<br>Embedded Real-time Systems<br>Human Interfaces |

The first level is an overview of the software engineering process, including how software evolves, how it is

generated, and how it is maintained. The educational objective is to give the students a minimal degree of knowledge and comprehension. In addition, a segment on technical communication is recommended at this level. Although this is not strictly software engineering material, it is important in later courses, and is unlikely to have been covered in the student's undergraduate curriculum. Implementations of the curriculum may choose either to have a separate short course or to integrate this material into other courses.

The second level stresses the control and management activities of software engineering. These are the activities that most clearly distinguish software engineering from programming; they also complement the student's knowledge of programming to provide the total conceptual framework for more detailed study.

The third level presents in-depth coverage of development activities in the context of programming-in-the-large. The students should achieve a level of skill in using tools, applying methods, and synthesizing software work products. Project courses or other forms of software experience are most appropriate at this level.

Social and ethical issues are also important to the education and development of a professional software engineer. Examples are privacy, data security, and software piracy. We do not recommend a course or unit specifically on these issues, but rather encourage instructors to find opportunities to discuss them in appropriate contexts in all courses and to set an example for students.

The curriculum topics are described below in units of unspecified size. Nearly all have a software engineering activity as the focus. For each, we provide a short description of the subtopics to be covered, the aspects of the activity that are most important, and the educational objectives of the unit. Curriculum modules and detailed course descriptions in each of these areas are under development at the SEI.

### 1. The Software Engineering Process

*Topics:* The software engineering process and software products. All of the software engineering activities. The concepts of software process model and software product life cycle model.

*Aspects:* All aspects, as appropriate for the various activities.

*Objectives:* Knowledge of activities and aspects. Some comprehension of the issues, especially the distinctions among the various classes of activities. The students should begin to understand the substantial differences between programming, as they have done in an undergraduate program, and software engineering, as it is practiced professionally.

### 2. Software Evolution

*Topics:* The concept of a software product life cycle. The various forms of a software product, from initial conception through development and operation to retirement. Controlling activities and disciplines to support evolution. Planned and unplanned events that affect software evolution. The role of changing technology.

*Aspects:* Models of software evolution, including development life cycle models such as the waterfall, iterative enhancement, phased development, spiral.

*Objectives:* Knowledge and comprehension of the models. Knowledge and comprehension of the controlling activities.

### 3. Software Generation

*Topics:* Various methods of software generation, including designing and coding from scratch, use of reusable components (including examples such as mathematical procedure libraries, packages designed specifically for reuse, Ada generic program units, and program concatenation as with pipes), use of program or application generators and very high level languages, role of prototyping. Factors affecting choice of a software generation method. Effects of generation method on other software development activities, such as testing and maintenance.

*Aspects:* Models of software generation. Representations for software generation, including design and implementation languages, very high level languages, and application generators. Tools to support generation methods, including application generators.

*Objectives:* Knowledge and comprehension of the various methods of software generation. Ability to apply each method when supported by appropriate tools. Ability to evaluate methods and choose the appropriate ones for each project.

### 4. Software Maintenance

*Topics:* Maintenance as a part of software evolution. Reasons for maintenance. Kinds of maintenance (perfective, adaptive, corrective). Comparison of development activities during initial product development and during maintenance. Controlling activities and disciplines

that affect maintenance. Designing for maintainability. Techniques for maintenance.

*Aspects:* Models of maintenance. Current methods.

*Objectives:* Knowledge and comprehension of the issues of software maintenance and current maintenance practice.

**5. Technical Communication**

*Topics:* Fundamentals of technical communication. Oral and written communications. Preparing oral presentations and supporting materials. Software project documentation of all kinds.

*Aspects:* Principles of communication. Document preparation tools. Standards for presentations and documents.

*Objectives:* Knowledge of fundamentals of technical communication and of software documentation. Application of fundamentals to oral and written communications. Ability to analyze, synthesize, and evaluate technical communications.

**6. Software Configuration Management**

*Topics:* Concepts of configuration management. Its role in controlling software evolution. Maintaining product integrity. Change control and version control. Organizational structures for configuration management.

*Aspects:* Fundamental principles. Tools (such as *sccs* or *rcs*). Documentation, including configuration management plans.

*Objectives:* Knowledge and comprehension of the issues. Ability to apply the knowledge to develop a configuration management plan and to use appropriate tools.

**7. Software Quality Issues**

*Topics:* Definitions of quality. Factors affecting software quality. Planning for quality. Quality concerns in each phase of a software life cycle, with special emphasis on the specification of the pervasive system attributes. Quality measurement and standards. Software correctness assessment principles and methods. The role of formal verification and the role of testing.

*Aspects:* Assessment of software quality: appropriate measures. Tools to help perform measurement. Correctness assessment methods, including testing and formal verification. Formal models of program verification.

*Objectives:* Knowledge and comprehension of software quality issues and correctness methods. Ability to apply formal verification methods.

**8. Software Quality Assurance**

*Topics:* Software quality assurance as a controlling discipline. Organizational structures for quality assurance. Independent verification and validation teams. Test and evaluation teams. Software technical reviews. Software quality assurance plans.

*Aspects:* Current industrial practice for quality assurance. Documents including quality assurance plans, inspection reports, audits, and validation test reports.

*Objectives:* Knowledge and comprehension of quality assurance planning. Ability to analyze and synthesize quality assurance plans. Ability to perform technical reviews. Knowledge and comprehension of the fundamentals of program verification, and its role in quality assurance. Ability to apply concepts of quality assurance as part of a quality assurance team.

**9. Software Project Organizational and Management Issues**

*Topics:* Project planning: choice of process model, project scheduling and milestones. Staffing: development team organizations, quality assurance teams. Resource allocation.

*Aspects:* Fundamental concepts and principles. Scheduling representations and tools. Project documents.

*Objectives:* Knowledge and comprehension of concepts and issues. It is not expected that a student, after studying this material, will immediately be ready to manage a software project.

**10. Software Project Economics**

*Topics:* Cost estimation, cost/benefit analysis, risk analysis for software projects. Factors that affect cost.

*Aspects:* Models of cost estimation. Current techniques and tools for cost estimation.

*Objectives;* Knowledge and comprehension of models and techniques. Ability to apply the knowledge to tool use.

**11. Software Operational Issues**

*Topics:* Organizational issues related to the use of a software system in an organization. Training, system installation, system transition, operation, retirement. User documentation.

*Aspects:* User documentation and training materials.

*Objectives:* Knowledge and comprehension of the major issues.

### 12. Requirements Analysis

*Topics:* The process of interacting with the customer to determine system requirements. Defining software requirements. Identifying functional, performance, and other requirements: the pervasive system requirements. Techniques to identify requirements, including prototyping, modeling, and simulation.

*Aspects:* Principles and models of requirements. Techniques of requirement identification. Tools to support these techniques, if available. Assessing requirements. Communication with the customer.

*Objectives:* Knowledge and comprehension of the concepts of requirements analysis and the different classes of requirements. Knowledge of requirements analysis techniques. Ability to apply techniques and analyze and synthesize requirements for simple systems.

### 13. Specification

*Topics:* Objectives of the specification process. Form, content, and users of a specifications document. Specifying functional, performance, reliability, and other requirements of systems. Formal models and representations of specifications. Specification standards.

*Aspects:* Formal models and representations. Specification techniques and tools that support them, if available. Assessment of a specification for attributes such as consistency and completeness. Specification documents.

*Objectives:* Knowledge and comprehension of the fundamental concepts of specification. Knowledge of specification models, representations, and techniques, and the ability to apply or use one or more. Ability to analyze and synthesize a specification document for a simple system.

### 14. System Design

*Topics:* The role of system design and software design. How design fits into a life cycle. Software as a component of a system. Hardware vs. software tradeoffs for system performance and flexibility. Subsystem definition and design. Design of high level interfaces, both hardware to software and software to software.

*Aspects:* System modeling techniques and representations. Methods for system design, including object oriented design, and tools to support those methods. Iterative design techniques. Performance prediction.

*Objectives:* Comprehension of the issues in system design, emphasizing engineering tradeoffs. Ability to use appropriate system design models, methods, and tools, including those for specifying interfaces. Ability to analyze and synthesize small systems.

### 15. Software Design

*Topics:* Principles of design, including abstraction and information hiding, modularity, reuse, prototyping. Levels of design. Design representations. Design practices and techniques. Examples of design paradigms for well-understood systems.

*Aspects:* Principles of software design. One or more design notations or languages. One or more widely used design methods and supporting tools, if available. Assessment of the quality of a design. Design documentation.

*Objectives:* Knowledge and comprehension of one or more design representations, design methods, and supporting tools, if available. Ability to analyze and synthesize designs for software systems. Ability to apply methods and tools as part of a design team.

### 16. Software Implementation

*Topics:* Relationship of design and implementation. Features of modern procedural languages related to design principles. Implementation issues including reusable components and application generators. Programming support environment concepts.

*Aspects:* One or more modern implementation languages and supporting tools. Assessment of implementations: coding standards and metrics.

*Objectives:* Ability to analyze, synthesize, and evaluate the implementation of small systems.

### 17. Software Testing

*Topics:* The role of testing and its relationship to quality assurance. The nature of and limitations of testing. Levels of testing: unit, integration, acceptance, etc. Detailed study of testing at the unit level. Formal models of testing. Test planning. Black box and white box testing. Building testing environments. Test case generation. Test result analysis.

*Aspects:* Testing principles and models. Tools to support

specific kinds of tests. Assessment of testing; testing standards. Test documentation.

*Objectives:* Knowledge and comprehension of the role and limitations of testing. Ability to apply test tools and techniques. Ability to analyze test plans and test results. Ability to synthesize a test plan.

### 18. System Integration

*Topics:* Testing at the software system level. Integration of software and hardware components of a system. Uses of simulation for missing hardware components. Strategies for gradual integration and testing.

*Aspects:* Methods and supporting tools for system testing and system integration. Assessment of test results and diagnosing system faults. Documentation: integration plans, test results.

*Objectives:* Comprehension of the issues and techniques of system integration. Ability to apply the techniques to do system integration and testing. Ability to develop system test and integration plans. Ability to interpret test results and diagnose system faults.

### 19. Embedded Real-time Systems

*Topics:* Characteristics of embedded real-time systems. Existence of hard timing requirements. Concurrency in systems and representing concurrency in requirements specifications, designs, and code. Issues related to complex interfaces between devices and between software and devices. Criticality of embedded systems and issues of robustness, reliability, and fault tolerance. Input and output considerations, including unusual data representations required by devices. Issues related to the cognizance of time. Issues related to the inability to test systems adequately.

*Objectives:* Comprehension of the significant problems in the analysis, design, and construction of embedded real-time systems. Ability to produce small systems that involve interrupt handling, low level input and output, concurrency, and hard timing requirements, preferably in a high level language.

### 20. Human Interfaces

*Topics:* Software engineering factors: applying design techniques to human interface problems, including concepts of device independence and virtual terminals. Human factors: definition and effects of screen clutter, assumptions about the class of users of a system, robustness and handling of operator input errors, uses of color in displays.

*Objectives:* Comprehension of the major issues. Ability to apply design techniques to produce good human interfaces. Ability to design and conduct experiments with interfaces, to analyze the results, and to improve the design as a result.

## The Software Engineering Experience Component

In addition to coursework covering the units described above, the curriculum should incorporate a significant software engineering experience component representing at least 30% of the student's work.

One form of experience is a cooperative program with industry, which has been common in undergraduate engineering curricula for many years. The University of Stirling uses this form in their Master of Science in Software Engineering program [Budgen86a]. Students enter the program in the fall semester of a four semester program. Between the first and second semesters they spend two to three weeks in industry, as an introduction to that company. They return to the company in July for a six month stay, during which time they participate in a professionally managed project. The fourth semester is devoted to a thesis or project report, based in part on their industrial experience.

Imperial College of Science and Technology has a similar industry experience as part of a four year program leading to a Bachelor of Science in Engineering degree [Lehman86a]. For this purpose, the College has set up Imperial Software Technology, Ltd. (IST), in partnership with the National Westminster Bank PLC, The Plessey Company PLC, and PA International. IST is an independent, technically and commercially successful company providing software technology products and services.

The more common form of experience, however, is one or more project courses as part of the curriculum. Two forms are common: a project course as a capstone following all the lecture courses, and a project that is integrated with one or more of the lecture courses.

The Wang Institute of Graduate Studies, Texas Christian University, and Seattle University have each offered the MSE degree for several years, and the College of St. Thomas is in its third year of offering such a degree (the TCU and St. Thomas degrees are actually named Master of Software Design and Development). Each incorporates a capstone project course into its curriculum [McKeeman86, Comer86, Mills86]. In recent years, the Wang Institute has chosen projects related to software tools that could be useful

to future students. TCU takes the professional backgrounds of its students into consideration when choosing projects. Seattle sometimes solicits real projects from outside the university.

It is worth noting that the project course descriptions for all three of these institutions do not mention software maintenance. Educators and practitioners alike have long recognized that maintenance requires the majority of resources in most large software systems. The minimal coverage of maintenance in software engineering curricula may be attributed to several factors: there does not appear to be a coherent, teachable body of knowledge on software maintenance; current thinking on improving the maintenance process is primarily based on improving the development process including the capturing of development information for maintenance purposes; and that giving students maintenance experience requires that there already exists a significant software system with appropriate documentation and change requests, the preparation of which is completely beyond the capabilities of an instructor in the normal time allotted to course preparation (the SEI has a small project underway to address this final problem).

The University of Southern California has built an infrastructure for student projects that continue beyond the boundaries of semesters and groups of students. The System Factory Project [Scacchi86] has created an experimental organizational environment for developing large software systems that allows students to encounter many of the problems associated with professional software engineering and to begin to find effective solutions to the problems. To date, over 250 graduate students have worked on the project and have developed a large collection of software tools.

The University of Toronto has added the element of software economics to its project course [Horning76, Wortman86]. The Software Hut (a small software house) approach requires student teams to build modules of a larger system, to try to sell their module to other teams (in competition with teams that have developed the same module), to evaluate and buy other modules to complete the system, and to make changes in purchased modules. At the end of the course, systems were "sold" to a "customer" at prices based on the system quality (as determined by the instructor's letter grade for the system). The instructor reports that this course had a very different character from previous project courses. The students' attempts to maximize their profits gave the course the flavor of a game and helped motivate the use of many techniques for increased software quality.

Arizona State University has attempted to build the project experience into a sequence of courses, combining lectures with practice [Collofello82]. The four courses were Software Analysis (requirements and specifications), Software Design, Software Testing, and Software Maintenance. The courses were offered in sequence so that a single project could be continued through all four. However, the students could take the courses in any order, and although many students did take them in the normal order, the turnover in enrollment from one semester to the next gave a realistic experience.

We do not believe that there is only one *correct* model for providing software engineering experience. It can be argued that experience is the basis for understanding the abstractions of processes that make up formal methods and that allow reasoning about processes. Therefore we should give the students experience first, with some guidance, and then show them that the formalisms are abstractions of what they have been doing. It can also be argued that we should teach "theory" and formalisms first, and then let the students try them in capstone project courses.

No matter what form the experience component takes, it should provide as broad an experience as possible. It is especially important for the students to experience, if not perform, the control activities and the management activities. Without these, the project can be little more than advanced programming.

## The MSE Curriculum Structured as Semester Courses

A typical master's degree curriculum requires 30 to 36 semester hours[4] credit. The units described in the previous section can be structured as semester courses of three hours each (except for Technical Communication, which may be a one or two hour course), totalling approximately 21 semester hours, leaving time for the software engineering experience component and for some electives. This structure is described below.

[4]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 graduate student with teaching or research responsibilities might take three courses (nine semester hours) each semester; a student without such duties might take five courses.

**Introduction to Software Engineering**

*Units:* The Software Engineering Process, Software Evolution, Software Generation, Software Maintenance, Software Configuration Management

**Software Quality**

*Units:* Software Quality Issues, Software Quality Assurance

**Software Project Management**

*Units:* Software Project Organizational and Management Issues, Software Operational Issues

**Software Requirements Specification**

*Units:* Requirements Analysis, Specification

**System Design**

*Units:* System Design, Embedded Real-time Systems, Human Interfaces

**Software Design and Implementation**

*Units:* Software Design, Software Implementation

**Software Testing and Integration**

*Units:* Testing, System Integration

The unit on technical communication should be presented early in the curriculum but need not be a separate course. It may be better integrated into other courses at appropriate places. Oral presentation skills might be taught along with software technical reviews, and writing skills might be taught in the first course where significant documents are required. We recommend that instructors look for possibilities to collaborate in the development of this unit with a university's English or Communications Department.

The material learned in the technical communication course should be reinforced throughout the curriculum by requiring the students to produce written documents and to make oral presentations. In the past, many instructors in the sciences and engineering have shown a reluctance to make technical communication a factor in student evaluations and grades. Because of its importance in software engineering, we strongly urge instructors to make it an integral part of all appropriate courses.

The courses outlined here focus almost exclusively on the software engineering process. It is recommended that the electives focus on the product. Courses based on particular system classes (such as distributed systems or expert systems) or on pervasive system attributes (such as fault tolerance or maintainability) will complement the process courses.

Because of the wide range of choices for electives, students can be well served by creative course design. For example, several small units of material (roughly one semester hour each) might be prepared by several different instructors. Three of these could then be offered sequentially in one semester, under the title *Topics in Software Engineering*, with different units offered in different semesters.

## Resources

Any new degree program will require a commitment of resources by a university. We cannot give a prescription for the exact kinds and levels of resources needed for an MSE program, but we can make some observations (in large part based on the experience of the schools that currently offer an MSE).

**Faculty.** Because software engineering is a new discipline, universities cannot simply recruit faculty from those with doctorates in software engineering—there are none. Furthermore, because most of the expertise in software engineering is found among the best practitioners in industry, the majority of whom do not have a doctorate in any field, universities should give very careful consideration to the selection of software engineering faculty and to the academic reward structure for those faculty.

There is a particularly difficult dilemma here. The faculty needs to be formed of persons committed to the academic profession, including curriculum development, teaching, and scholarly activity. At the same time, the expertise in software engineering may be outside the university, in persons committed to the software engineering profession.

It may be tempting to try to build a new program with a substantial number of adjunct faculty from local industry. We recommend against this approach, primarily because we believe there must be a coherence and continuity to the core curriculum that can only be achieved with faculty that live with it every day and from year to year. The curriculum must be more than a set of independent courses.

Adjunct faculty can be used effectively in seminar and topics courses, but the subjects to be covered in those

courses should be chosen by the regular faculty and should serve a specific purpose in the curriculum. We estimate that ten to twenty percent of courses might be taught by adjunct faculty.

Most software engineering courses, especially project courses, will require a large commitment of time by the instructor in addition to traditional teaching activities. This time must be devoted to activities such as developing the computing environment, creating software tools or acquiring and rehosting tools, and playing a project management role for student project teams. In addition, there are additional time costs in starting a new degree program and in developing the relationships with the industrial community that are useful for software engineering education. Therefore, we recommend that the teaching load for faculty, at least in the early years of a new program, be one course per semester.

In selecting and rewarding faculty, we also recommend that significant professional experience in software engineering be considered along with academic and research experience. Most advances in software engineering are likely to be found in industry, rather than in university laboratories, so faculty should be encouraged to be part of that industry environment whenever possible. The National Science Board [NSB86] suggests a variety of faculty professional development efforts for engineering faculty, and we recommend that they be studied by schools considering an MSE program.

The number of faculty needed depends on the number of courses to be offered, which in turn depends on the number of students in the program. We estimate that a program based on the curriculum described here and that serves about twenty graduates per year would require a minimum of five full time faculty.

**Staff.** Because of the substantial computing resources needed to support an MSE program, the students and faculty will need the support of a facilities staff. Wang Institute has reported needing two full time staff members to maintain the software tools used by the students. A hardware maintenance staff may also be needed.

As software engineering becomes more tool-intensive, there will be an increasing need for training in tool use. This will be true in universities also. It is recommended that the staff include a person responsible for offering the appropriate training in a structured way, separate from the normal courses for degree credit.

**Software.** New software tools will continue to shape the way software engineers work, so the use of tools should be a major part of the curriculum. Some examples of important classes of tools are:

- those that support communication, such as word or document processing systems, electronic mail and bulletin boards, and text editors;
- language-oriented tools, such as compilers, profilers, cross reference generators, syntax analyzers, syntax-directed editors, path analyzers, and prettyprinters;
- project management tools, including those for project scheduling, resource management, and cost estimation;
- application generators and other very high level language tools.

The SEI plans to work with software engineering researchers and educators to identify, develop, and disseminate additional software tools for use in education.

**Hardware.** Even a university with existing computer science degree programs should plan a significant increase in hardware to support an MSE program. The trend in software engineering is toward a local network of powerful workstations for developers. The cost of such workstations continues to fall, with typical examples now available for $4,000 to $12,000 with educational discounts. Both larger and smaller machines are also likely to be useful because there is a growing set of useful software tools on these machines. High quality printers are also needed to support the substantial amount of documentation produced in software engineering courses.

Part of the process of planning a new degree program is the development of a detailed resource requirements list and a plan for acquiring those resources. University committees and state boards involved in the approval process need this information. The SEI, when requested, will work closely with its Academic Affiliates to do this planning.

## A One Semester Course In Software Engineering

Establishing a number of MSE programs will eventually have an impact on the number and quality of professional software engineers, but for the next five years we expect very few such programs producing very few graduates. A

more moderate goal, but one that can be achieved very quickly, is to provide some software engineering education within existing undergraduate computer science programs. For many schools, the starting point is a one semester elective course, probably at the senior level, that helps students understand the major differences between computer science and software engineering, and that helps prepare the students for careers or further education in software engineering.

The emphasis of the course should be on those things that are most different from typical student programming: project teams in which different members play very different roles, and the discipline imposed on the process by configuration management, quality assurance, and resource constraints. To accomplish this, the course should be structured as a software development project, with students as the project team. Lectures will be mixed with project meetings and the various reviews and inspections of work products.

A good course, like a good project, needs a good manager. Instructors will have to be the project managers because only they will have sufficient authority and knowledge to keep the project on schedule. The students will still experience project management, although from the point of view of the managed rather than the manager. (In the future, students in an MSE program may play the role of project manager for the undergraduate course.)

Because all students have already had a lot of experience writing code, we believe it is not necessary for all of them to write more code in this course. Instead, it is much more important for them to be part of an entire project team. Not every student can play every role on the team, but each student will see each role being played and will interact with students in other roles. A suggested set of roles is outlined below.

Most of the products of the course, like most of the products of a software project, are documents. Therefore it is important that all students have skills using some kind of document production software. In addition, it is critically important that the instructor be prepared to exemplify all the necessary kinds of documents to be produced, perhaps by giving copies of documents from another project and by preparing document skeletons or templates for student use. Instructors can be guided by existing standards for these documents, such as those established by the DoD, NASA, and IEEE.

In order for students to understand the budget aspects of the software engineering process, it is necessary to track all resources used in the course of the project. This includes both student time and machine time. Students should be asked to keep accurate records of their activities and to make appropriate reports at weekly intervals. Since most student computing environments (personal computers or workstations) do not do extensive accounting of time and resources used, the students should also be asked to record and report their machine use. The student project administrator should convert these reports to project costs using typical industry values for salaries and overhead costs.

In order for the project to be completed, it must be of appropriate size for the skills of the students. If they will need to learn a new computer system or programming language for the course, a smaller project should be attempted. In any case, some flexibility can be achieved if the life cycle models segment of the first lecture emphasizes phased development or iterative enhancement models for this project. It is then possible to modify the project completion goals several weeks into the project.

The SEI has prepared a detailed (750 pages) description of this course including a full, annotated syllabus, recommended texts and readings, descriptions of the student roles and responsibilities, sample examinations, and examples of all the project documents expected [Tomayko87].

**Project Team Roles.** The exact structure of the student project team will necessarily be affected by the class size, but the roles described below are essential. The numbers in parentheses indicate a suggested number of students for the particular role. If the class is substantially larger than this model, then two parallel development teams (design, implementation, and testing) might be used.

*Project Manager* (1): The instructor usually plays this role.

*Project Administrator* (1): This student is responsible for tracking resource usage and other administrative activities.

*Principal Architect* (1): This student is responsible for the overall specification of the system and writes the requirements and specification documents.

*Configuration Management* (2-3): These students are responsible for writing the configuration management plan, establishing a configuration control board, creating necessary forms for configuration management procedures, maintaining a repository of current and previous software configuration items, receiving change requests and discrepancy reports and presenting them to the configuration control

board, ensuring that approved changes are made, and conducting audits of the configuration management process.

*Quality Assurance* (2-3): These students are responsible for establishing documentation and source code standards, planning and conducting reviews, audits, and inspections of documentation, source code, and testing procedures, and participating in the activities of the configuration control board.

*Design* (2-4): These students are responsible for detailed system design, including choice of a design method and representation, and for writing the design documents.

*Implementation* (2-4): These students implement the design as provided by the design team. Deviations from the design must be submitted to and approved by the configuration control board.

*Test and Evaluation* (2-3): These students develop the test plan and strategy for unit and system testing, perform those tests, and record the results of the tests. They lead the implementors in integration testing.

*Verification and Validation* (3-5): These students are responsible for development of test plans and test data, and for planning and conducting the final acceptance tests of the system. This includes functional, performance, stress, and validation testing.

*Document Manager* (1): This student is responsible for user-level documentation.

**Course Outline.** The course outline below is structured as 22 class meetings of 75 to 90 minutes each, and thus can easily be taught in a semester with classes twice each week. The remaining class time should be used for examinations, for additional emphasis on topics of the instructor's choice, and for recovering from unexpected delays.

1. *Software Engineering: Programs as Products/Life Cycle Models:* Introduction to the concept of software engineering as opposed to computer science or programming. Discussion of software as products to be used by people other than the developers. Presentation of different life cycle models, such as waterfall, rapid prototype, incremental development, etc. Introduction to the class project.

2. *Development Standards/Project Organization:* Standards for software development, including government standards, IEEE standards, and corporate standards. Models of team organization, such as surgical team, democratic, and chief programmer.

3. *Requirements Engineering:* How requirements are determined. Interactions with customers, marketing, and development organizations. Stating requirements and developing the requirements document.

4. *Controlling Disciplines: Quality Assurance and Configuration Management:* What software QA and configuration management organizations do in a software project. Relationship of their activities to the developers. Concept of independent verification and validation.

5. *Cost, Size, and Manpower Planning:* These topics relate to project management. Cost estimation techniques and methods such as COCOMO. Software size estimating and its relation to schedule and cost. Staff loading on a project over the life cycle. Mythical man-month discussion.

6. Review of requirements document; quality assurance and configuration management plans due.

7. *Specification Techniques:* Formal specification tools and techniques such as on-line tools and data flow diagrams. Functional specification development.

8. *Design Concepts and Methods:* Survey of design methods: Top-down structured, Jackson, Warnier-Orr, object-oriented, etc. Advantages and disadvantages of each in differing problem domains.

9. *Design Concepts and Methods:* Continued discussion.

10. Review of specification document.

11. Preliminary design review; test plans and user document outline due.

12. *Design Representation:* Using structure charts, HIPO charts, data flow diagrams, pseudocode, and other tools in implementing designs.

13. *Structured Programming and Implementation Considerations:* Review of the concepts of structured programming. Discussion of Bohm and Jacopini paper [Bohm66]. Applying structure to non-structured tools such as assembly languages. Coding considerations in FORTRAN and COBOL versus Pascal and Ada.

14. Configuration control board meeting to close discrepancy reports and change requests against the requirements and specification.

15. Critical design review.

16. *Software Testing and Integration Concepts:* Unit

testing techniques, white-box versus black-box testing. Concept of coverage. Integration methods, such as top-down, bottom-up, and Big Bang. Development of testing and integration suites.

17. *Verification and Validation:* Formal verification, concept of validation and acceptance testing. Development of validation suites. Automated testing.

18. *Verification and Validation:* Continued discussion.

19. Code inspections; release of code to test and evaluation team.

20. *Post-Development Software Evolution:* The software maintenance problem. Designing for maintenance. Developing a maintenance handbook for a software product. Reverse-engineering of software product documentation to improve maintainability of existing code.

21. *User Documentation:* Characteristics of good user documentation. Writing user documentation if you are a developer. Document organization and style; ways to assist the reader.

22. Final product review and acceptance.

## The Future of Software Engineering Education

Software engineering is an emerging discipline, and it will be a challenge to educators to keep pace. The next ten to fifteen years can be expected to bring many changes in the structure of software engineering academic programs.

We expect that computer science and software engineering will continue to distinguish themselves from each other. Also, even though we expect an overall increase in the resources devoted to education in these two fields, it is probable that software engineering education will grow faster than the two combined, thus siphoning off some of the resources currently devoted to computer science education. However, we do not expect many universities to establish separate academic departments anytime soon, so the resource tradeoffs will be intradepartmental and relatively easy to handle.

### Undergraduate Education

We believe there are two immediate trends in undergraduate software engineering education. The first is the adoption of a senior level software engineering course, such as the one described in the previous section, in existing computer science degree programs. The second is continued evolution of lower level programming courses in the direction of increased consideration of concepts and language features that support programming-in-the-large, primarily data abstraction, information hiding, concurrency, and exception handling.

There are also two longer term possibilities in undergraduate software engineering education to be explored. The first is the development of an undergraduate degree with the same kind of expectations of its graduates as is now the case with traditional undergraduate engineering degrees. The SEI Education Program will investigate such degree programs, although our best recommendation today is that it is premature. The body of knowledge needed by software engineers depends heavily on undergraduate computer science knowledge, and both cannot easily be taught in a four year program.

As the distinctions between computer science and software engineering become clearer, it is feasible that separate undergraduate programs will emerge. We would expect that they would have two years of common studies, much as the traditional engineering disciplines share an *engineering core*. The third year would also probably have some common courses, while the fourth year might be entirely different between the two programs.

A second, quite different possibility for undergraduate education is based on the expectation of some members of the software engineering community that the profession will partition itself into at least two levels of skill. Software engineers at the higher level will be project leaders, system architects, and designers, while those at the lower levels will be skilled in the use of particular software tools or in performing particular tasks, such as testing. The two levels of practitioners will require different levels of education and experience, the higher level requiring an MSE degree and the lower level perhaps a bachelor's degree.

There are precedents for this kind of partitioning of effort. The data processing community has long distinguished systems analysts from coders, and the engineering community is supported by engineering technicians. It is too soon to see clear trends in software engineering, but the SEI Education Program plans to continue monitoring the profession.

### Graduate Education

A professional master's degree in software engineering

(MSE) is our current recommendation for the most effective graduate education. Our preliminary suggestions for the content of such a degree program have been presented in this report.

The software community generally agrees that experience still plays a very large role in the development of a good software engineer. Some persons have suggested that software engineering may want to look to medicine for educational models, such as internships, residencies, and teaching hospitals. Thus the structure of the experiential component of the MSE program will continue to be an area of investigation for the SEI.

Doctoral level education in software engineering is also possible. The Wang Institute of Graduate Studies has done preliminary development of a program leading to a Doctor of Philosophy degree in software engineering. The rapid growth of software engineering as an academic discipline leads us to believe such programs will begin to appear in the 1990s.

## Certification and Accreditation

Although it is certainly premature to begin, suggestions that software engineers be certified and that software engineering degree programs be accredited have been made. There is certainly precedent for both in the traditional engineering disciplines. The SEI does not intend to be in the business of doing either but expects to serve as a resource to the software engineering community in these matters.

A related area that the SEI does expect to investigate is the assessment of student backgrounds relative to prerequisites for an MSE program. Such assessments are likely to be useful to instructors for purposes of choosing appropriate starting points for the first courses in the curriculum, and to students for purposes of identifying areas were extra work is needed. We also expect to offer self-study guidelines and materials for students wishing to remove prerequisite deficiencies.

## SEI Affiliate Programs

The SEI has established a number of affiliate programs to promote direct interactions between the Institute and the software engineering community. These interactions are critical to the success of the SEI, since they are the sources of new ideas and the best path for new technology from the SEI to reach potential users.

Separate affiliate programs serve academic, industry, and government organizations. All provide for exchange of information, but their most important aspect is that they encourage staff from affiliated organizations to come to the SEI for an extended period to work with us in the investigation and development of new software engineering technology.

**Academic Affiliates Program.** The Academic Affiliates Program provides a means whereby educational institutions can join the SEI in cooperative efforts of mutual interest. Faculty members from affiliate institutions have been major contributors to the Graduate Curriculum Project and may also apply for visiting positions in any of the SEI's research or development projects.

Academic affiliates normally participate in the educational activities of the SEI, including faculty development workshops and the annual Conference on Software Engineering Education. We also ask them to share their experiences and ideas with us, which we in turn share with other interested parties.

Affiliate institutions wishing to develop a graduate level curriculum in software engineering may be designated *Graduate Curriculum Test Sites*. The SEI works closely with these schools to tailor the curriculum, courses, and materials to their particular needs.

**Industry Affiliates Program.** The Industry Affiliates Program provides for direct interactions between the SEI technical staff and the commercial segment of the software community. These organizations are the sources of many of the important new ideas in software engineering, which the SEI can then investigate and develop. The companies can send *resident affiliates* to the SEI to participate in that development, to absorb information from other SEI projects, and finally return to their companies with new technology.

Industry affiliates also interact with the SEI Education Program. Software engineers from several companies have already come to the SEI to share their knowledge with us for the development of curriculum modules, and some modules have been packaged and taught as short courses at the affiliates' sites.

**Government Affiliates Program.** The Government Affiliates Program is similar to the Industry Affiliates Program, but the interactions are with government organiza-

tions. The Department of Defense is perhaps the world's larger purchaser and maintainer of software, and the SEI plays an important role as a resource for the various defense agencies involved.

For additional information on the SEI affiliate programs, contact the Director of Affiliate Relations at the SEI.

## Acknowledgements

The curriculum recommendations in this report have benefitted from the suggestions of a large number of persons. We had valuable discussions with many members of the SEI technical staff, including Mario Barbacci, Clyde Chittister, Lionel Deimel, Larry Druffel, Peter Feiler, Priscilla Fowler, Dick Martin, John Nestor, Joe Newcomer, Mary Shaw, Nelson Weiderman, Chuck Weinstock, and Bill Wood, and with visiting staff members Bob Aiken, Brad Brown, David Budgen, Fred Cohen, Jim Collofello, Bob Glass, Paul Jorgensen, Nancy Leveson, Ev Mills, Rich Sincovec, Joe Turner, and Peggy Wright.

Earlier versions of the MSE recommendations were written by Jim Collofello and Jim Tomayko, and reviewed by Evans Adams, David Barnard, Dan Burton, Phil D'Angelo, David Gries, Ralph Johnson, David Lamb, Manny Lehman, John Manley, John McAlpin, Richard Nance, Roger Pressman, Dieter Rombach, George Rowland, Viswa Santhanam, Walt Scacchi, Roger Smeaton, Joe Touch, and K. C. Wong.

An early version of the MSE curriculum was the subject of discussion at the Software Engineering Education Workshop, which was held at the SEI in February, 1986 [Gibbs86c]. In addition to several of the persons mentioned above, the following participants at the workshop contributed ideas to the current curriculum recommendations: Bruce Barnes, Victor Basili, Jon Bentley, Gordon Bradley, Fred Brooks, James Comer, Dick Fairley, Peter Freeman, Susan Gerhart, Nico Habermann, Bill McKeeman, Al Pietrasanta, Bill Richardson, Bill Riddle, Walter Seward, Ed Smith, Dick Thayer, David Wortman, and Bill Wulf.

## References

**[AFIPS80]**
AFIPS Taxonomy Committee. *Taxonomy of Computer Science & Engineering.* AFIPS Press, Arlington, VA, 1980.

**[Ardis85]**
Ardis, Mark, James Bouhana, Richard Fairley, Susan Gerhart, Nancy Martin, and William McKeeman. *Core Course Documentation: Master's Degree Program in Software Engineering.* TR-85-17, Wang Institute of Graduate Studies, Sept., 1986.

**[Austing78]**
Austing, Richard, Bruce Barnes, Della Bonnette, Gerald Engle, and Gordon Stokes. Curriculum '78: Recommendations for the Undergraduate Program in Computer Science. *Comm. ACM 22*, 3 (March 1979), 147-166.

**[Babb79]**
Babb, Robert G., and Leonard L. Tripp. An Approach to Defining Areas Within the Field of Software Engineering. *ACM Software Engineering Notes 4*, 4 (Oct. 1979), 9-17.

**[Barbacci85]**
Barbacci, Mario R., A. Nico Habermann, and Mary Shaw. The Software Engineering Institute: Bridging Practice and Potential. *IEEE Software 2*, 6 (Nov. 1985), 4-21.

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

**[Bohm66]**
Bohm, C., and G. Jacopini. Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules. *Comm. ACM 9*, 5 (May 1966), 366-371.

**[Brooks86]**
Brooks, Frederick P. No Silver Bullet—Essence and Accidents of Software Engineering. In *Information Processing 86*, H.-J. Kugler, ed. IFIP, 1986.

**[Budgen86a]**
Budgen, David, Peter Henderson, and Chic Rattray. Academic/Industrial Collaboration in a postgraduate MSc course in Software Engineering. In *Software Engineering Education: The Educational Needs of the Software Community*, Gibbs, Norman E., and Richard E. Fairley, eds. Springer-Verlag, New York, 1986, 201-211.

**[Budgen86b]**
Budgen, David, and Richard Sincovec. *Introduction to Software Design.* Curriculum Module SEI-CM-2.0, Software Engineering Institute, Carnegie-Mellon University, Sept., 1986.

**[CMU80]**
*The Computer Science Ph.D. Program at Carnegie-Mellon University.* Department of Computer Science, Carnegie-Mellon University. Dec., 1980.

**[Cohen86]**
Cohen, Fred. *Information Protection.* Curriculum Module SEI-CM-5.0, Software Engineering Institute, Carnegie-Mellon University, Sept., 1986.

**[Collofello82]**
Collofello, James S. A Project-Unified Software Engineering Course Sequence. *Proc. Thirteenth SIGCSE Technical Symposium on Computer Science Education.* 1982, 13-19.

**[Collofello86]**
Collofello, James. *The Software Technical Review Process.* Curriculum Module SEI-CM-3.0, Software Engineering Institute, Carnegie-Mellon University, Sept., 1986.

**[Comer86]**
Comer, James R., and David J. Rodjak. Adapting to Changing Needs: A New Perspective on Software Engineering Education at Texas Christian University. In *Software Engineering Education: The Educational Needs of the Software Community*, Gibbs, Norman E., and Richard E. Fairley, eds. Springer-Verlag, New York, 1986, 149-171.

**[DOD85]**
Department of Defense. Contract F19628-85-C-003. Contract with Carnegie-Mellon University for the creation of the Software Engineering Institute.

**[Fairley78]**
Fairley, Richard E. Educational Issues in Software Engineering. *Proc. 1978 ACM National Conf.* 1978, 58-62.

**[Fairley79a]**
Fairley, Richard E. MSE79: First Draft of a Masters Curriculum in Software Engineering. *ACM Software Engineering Notes 4*, 1 (Jan. 1979), 12-17.

**[Fairley79b]**
Fairley, Richard E. MSE79: Second Draft of a Masters Curriculum in Software Engineering. *ACM Software Engineering Notes 4*, 2 (April 1979), 13-16.

**[Fairley80]**
Fairley, Richard E. Software Engineering Education. *Proc. Thirteenth Hawaii Intl. Conf. System Sciences.* 1980, 70-75.

**[Freeman76]**
Freeman, Peter, Anthony I. Wasserman, and Richard E. Fairley. Essential Elements of Software Engineering Education. *Proc. 2nd Intl. Conf. on Software Engineering.* 1976, 116-122.

**[Freeman78]**
Freeman, Peter, and Anthony I. Wasserman. A Proposed Curriculum for Software Engineering Education. *Proc. 3rd Intl. Conf. on Software Engineering.* May, 1978, 56-62.

**[Gibbs86a]**
Gibbs, Norman, and Allen Tucker. A Model Curriculum for a Liberal Arts Degree in Computer Science. *Comm. ACM 29*, 3 (March 1986), 202-210.

**[Gibbs86b]**
Gibbs, Norman, and Gary Ford. *The Challenges of Educating the Next Generation of Software Engineers.* SEI-86-TM-7, Software Engineering Institute, Carnegie-Mellon University, June, 1986.

**[Gibbs86c]**
*Software Engineering Education: The Educational Needs of the Software Community.* Gibbs, Norman E., and Richard E. Fairley, eds. Springer-Verlag, New York, 1986.

**[Goldberg86]**
Golberg, Robert. Software engineering: An emerging discipline. *IBM Systems J. 25*, 3/4 (1986), 334-353.

**[Hoffman78a]**
Hoffman, A. A. J. A Proposed Masters Degree in Software Engineering. *Proc. 1978 ACM National Conf.* 1978, 54-57.

**[Hoffman78b]**
Hoffman, A. A. J. A Survey of Software Engineering Courses. *ACM SIGCSE Bulletin 10*, 3 (Aug. 1978), 80-83.

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

**[Horning76]**
Horning, J. J. The Software Project as a Serious Game. In *Software Engineering Education: Needs and Objectives: Proceedings of an Interface Workshop*, Anthony Wasser-

man and Peter Freeman, eds. Springer-Verlag, New York, 1976, 71-75.

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

**[IEEE86]**
IEEE Computer Society Software Engineering Standards Subcommittee. Draft Standard Taxonomy for Software Engineering Standards. IEEE Computer Society. Final version not yet published.

**[Jensen78]**
Jensen, Randall W., Charles C. Tonies, and W. I. Fletcher. A Proposed 4-Year Software Engineering Curriculum. *Proc. Ninth SIGCSE Tech. Symp. on Computer Science Education.* 1978, 84-92.

**[Jensen79]**
Jensen, Randall W., and Charles C. Tonies. Software Engineering Education: A Constructive Criticism. In *Software Engineering*, Jensen, R. W., and C. C. Tonies, eds. Prentice-Hall, Englewood Cliffs, 1979, 553-567.

**[Jorgensen86]**
Jorgensen, Paul. *Requirements Specification Overview.* Curriculum Module SEI-CM-1.0, Software Engineering Institute, Carnegie-Mellon University, Sept., 1986.

**[Koffman84]**
Koffman, E. B., P. L. Miller, and C. E. Wardle. Recommended Curriculum for CS1, 1984. *Comm. ACM 27*, 10 (Oct. 1984), 998-1001.

**[Koffman85]**
Koffman, E. B., D. Stemple, and C. E. Wardle. Recommended Curriculum for CS2, 1984. *Comm. ACM 28*, 8 (Aug. 1985), 815-818.

**[Lehman86a]**
Lehman, Manny M. The Software Engineering Undergraduate Degree at Imperial College, London. In *Software Engineering Education: The Educational Needs of the Software Community*, Gibbs, Norman E., and Richard E. Fairley, eds. Springer-Verlag, New York, 1986, 172-181.

**[Lehman86b]**
Lehman, M. M. Advanced Software Technology—Development and Introduction to Practice. In *Information Processing 86*, H.-J. Kugler, ed. IFIP, 1986.

**[McGill84]**
McGill, J. P. The Software Engineering Shortage: A Third Choice. *IEEE Trans. Software Eng. SE-10*, 1 (Jan. 1984), 42-49.

**[McKeeman86]**
McKeeman, William M. Experience with a Software Engineering Project Course. In *Software Engineering Education: The Educational Needs of the Software Community*, Gibbs, Norman E., and Richard E. Fairley, eds. Springer-Verlag, New York, 1986, 234-262.

**[Mills86]**
Mills, Everald. The Master of Software Engineering [MSE] Program At Seattle University After Six Years. In *Software Engineering Education: The Educational Needs of the Software Community*, Gibbs, Norman E., and Richard E. Fairley, eds. Springer-Verlag, New York, 1986, 182-200.

**[Mulder75]**
Mulder, M. C. Model Curricula for Four-Year Computer Science and Engineering Programs: Bridging the Tar Pit. *IEEE Computer 8*, 12 (Dec. 1975), 28-33.

**[Nance80]**
Nance, Richard E., and Walter P. Warner. *Anticipating the Software Engineer: The Academic Preparation.* NSWC TR 80-108, Naval Surface Weapons Center, Dahlgren, Virginia, May, 1980.

**[NRC85]**
National Research Council, Commission on Engineering and Technical Systems. *Engineering Education and Practice in the United States: Foundations of Our Techno-Economic Future.* National Academy Press, Washington, D.C., 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., March, 1986.

**[Scacchi86]**
Scacchi, Walter. The Software Engineering Environment for the System Factory Project. *Proc. Nineteenth Hawaii Intl. Conf. Systems Sciences.* 1986, 822-831.

**[Shaw73]**
Shaw, Mary. Immigration Course in Computer Science: Teaching Materials and 1972 Schedule. *ACM SIGCSE Bulletin 5*, 2 (June 1973), 26-32.

**[Stucki78]**
Stucki, Leon G., and Lawrence J. Peters. A Software Engineering Graduate Curriculum. *Proc. 1978 ACM National Conf.* 1978, 63-67.

**[Tomayko86a]**
Tomayko, James. *Software Configuration Management.* Curriculum Module SEI-CM-4.0, Software Engineering Institute, Carnegie-Mellon University, Sept., 1986.

**[Tomayko86b]**
Tomayko, James, ed. *Support Materials for Software Configuration Management.* Support Materials SEI-SM-4.0, Software Engineering Institute, Carnegie-Mellon University, Sept., 1986.

**[Tomayko87]**
Tomayko, James E. *Teaching a Project-Intensive Introduction to Software Engineering.* SEI-87-SR-1, Software Engineering Institute, Carnegie-Mellon University, March, 1987.

**[Warner82]**
Warner, Walter P., and Richard E. Nance. The Development of Software Engineers: A View from a User. *Proc. AFIPS National Computer Conference.* 1982, 293-300.

**[Wasserman76]**
*Software Engineering Education: Needs and Objectives: Proceedings of an Interface Workshop.* Wasserman, Anthony I., and Peter Freeman, eds. Springer-Verlag, New York, 1976.

**[Wortman86]**
Wortman, David B. Software Projects in an Academic Environment. In *Software Engineering Education: The Educational Needs of the Software Community*, Gibbs, Norman E., and Richard E. Fairley, eds. Springer-Verlag, New York, 1986, 292-305.