Shaw, M. (1987). Education for the future of software engineering (pp. 344-357). In N.E. Gibbs & R.E. Fairley (Eds.), Software engineering education. New York, NY: Springer-Verlag.

Education for the Future of Software Engineering

Mary Shaw
Software Engineering Institute

Abstract. The discipline of software engineering is developing rapidly. Its practitioners must deal with an evolving collection of problems and with new technologies for dealing with those problems. Software engineering education must anticipate new problems and technologies, providing education in the enduring principles of the field in the context of the best current practice. Since changes in the discipline cannot be completely anticipated, software engineers must be able to assume responsibility for their own continuing professional development. This paper describes significant changes now taking place in the field of software engineering and proposes some goals and objectives for the professional education of software engineers.

Software engineering is concerned with finding practical solutions to computational problems. Over the next few years, software engineering will be required

- to respond to society's broadening needs and higher expectations for software
- to deal with constantly increasing expectations for software functionality and performance
- to gain intellectual control over software development and support.

The major challenges that arise from these requirements will be to broaden software engineering's traditional scope of attention and to increase the scale of systems that can be successfully developed and supported. This will require significant changes in the character of the problems that we work on and the methods that we use to solve these problems.

The demand for software is rising more rapidly than our ability to supply the desired capability. For example, Figure 1 uses code size to estimate software demand. The growth rate for this particular application, onboard software in manned spacecraft, is nearly 30% annually. The figure compares this demand growth with the growth of programmer productivity, which is only about 5% annually. We clearly need to find

ways to increase not only the productivity of software engineers but also the rate at which their productivity grows. This problem is one of several software engineering problems aggravated by increasing system complexity. Software engineering education will play a significant role in solving these problems.

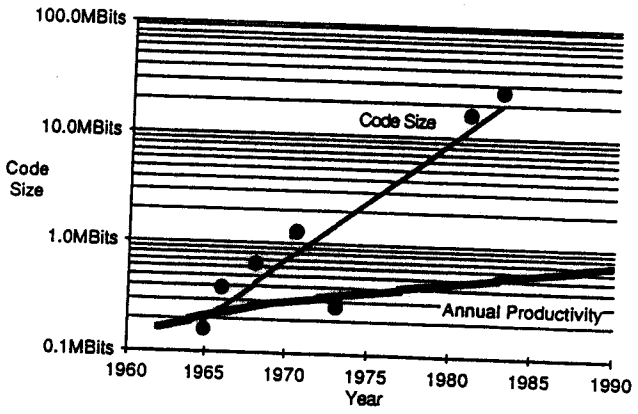

Figure 1. Relative Growth of Software Demand and Productivity

The argument of this paper is as follows. As system complexity increases, the essential character of the most critical problems of software engineering also changes. In order to cope with the complexity of large systems and the new kinds of problems that emerge, software engineering must move from an *ad hoc* basis to a technology-intensive basis rooted in sound models and theories. The principles we use and teach must transcend current practice; they must be codified and teachable. In some cases, such principles can be identified; in other cases we have some systematic understanding that is incompletely codified; in other cases we make do with rules of thumb while trying to develop sound models and theories. Software engineering education must prepare practitioners for future growth by teaching them principles based on sound models in the context of the best current practice.

Effects of Scale on Software Engineering

Software engineering has progressed from solving small problems to solving quite large ones. At each stage in this history, the attention of the software engineering community has been directed at some set of issues that can be understood as characteristic of the major problems of software development at that particular time. Each new generation of systems has been more ambitious than the previous, and new problems emerge as a consequence of this increase in scale. A significant increase in system scale and a corresponding shift in the character of the critical problems seem to take place roughly every decade.

Each time there is a quantum increase in the complexity of software systems, some different aspect of system development becomes the intellectual bottleneck. In the 1960's the problem was writing understandable programs, or *programming-in-the-small*, and the solution was implemented through high-level languages. In the 1970's the problem was organizing large software system development, and the solution was implemented through tools for *programming-in-the-large*. The significance of the distinction between programming-in-the-small and programming-in-the-large is that it is necessary to think about these two kinds of problems in essentially different ways; when the distinction was established, the attention of a significant fraction of the software engineering community was directed to that new problem. When a shift of bottleneck takes place, the problems encountered with smaller systems remain, but the new bottleneck forces the field to attend to a new set of problems in a fashion that may be essentially different from the way we thought about previous problems. The earlier, smaller problems don't disappear, however; they usually remain as subproblems in the larger systems.

In the decade since software engineering recognized programming-in-the-large as a significant issue, the complexity of software systems has grown by another leap, and another shift is now taking place. Software engineers must now deal with complex systems in which software is one of many components in a large heterogeneous system and in which the software is expected to serve as a surrogate for a human programmer, taking an active role in the development and control of software sys-

tems. We will describe those new modes of operation as *program-as-component* and *program-as-deputy*, respectively. This analysis is elaborated in "Beyond Programming in the Large: The Next Challenge for Software Engineering," Tech Memo SEI-86-TM-6, May 1986. [86]

Identification of these new modes recognizes a change in the character of the problems that depend on computational solutions as well as a change in the character of the software development and support process:

- They are not necessarily amenable to algorithmic solution.
- They involve judgmental elements such as selecting among competing, non-absolute preferences.
- They depend on problem-specific knowledge that must be consulted dynamically.
- They are so complex that solutions cannot be specified *a priori* but must be evolved through experience.
- They involve integration of a heterogeneous set of system components including hardware as well as software. They require graceful accommodation of unreliable data and other vagaries of physical systems.

The role of *program-as-component* arises in large heterogeneous systems. Such systems include programs in multiple languages for complex hardware systems; they may have mechanical constraints, produce noisy data, or impose real-time constraints on operation.

The role of *program-as-deputy* arises when large, creative portions of the program development process are delegated to software. This shift has been taking place gradually ever since the first symbolic assembler assigned addresses to variables. As time has passed, more and more expertise about various aspects of the software development process has been incorporated in programs which perform increasingly creative subtasks within the software development and management process.

These shifts reflect only the changes in the technology of software development and support. As system scale has increased, issues from several other areas have also become critical.

- *Professional Issues:* Software engineering will experience a significant personnel shortfall for at least the next 5-10 years. Attention to education, career paths, and professionalism will help to take up the slack.
- *Legal Issues:* Software is unlike either hard products or books. As a result, neither patent law nor copyright law is quite appropriate for software products and tools. Intellectual property law for software must deal with such issues as software protection, product liability, impediments to dissemination of new technology, and rights in technical data.
- *Economic Issues:* Costs of software development arise from many sources, and software consumes an increasing fraction of corporate resources. In addition, accounting rules for software influence corporate decisions about innovation. Software engineers often fail to appreciate cost components other than the ones directly associated with creating the software.
- *Managerial Issues:* Management concerns have interacted with software technology ever since we recognized the issues of programming-in-the-large. As systems grow larger, managerial issues expand to include improved costing and estimating techniques, the visibility into software development necessary for effective control, adequate performance measures for human organizations, and incentives and risk reduction measures to encourage more productive software technology.

Although these areas have not generally been covered in software engineering education, their role now requires attention.

The significance of these shifts is not so much the specific developments I have predicted, but the inevitability of some form of change. Progress in software engineering is a fact of life. Systems and tools will change continually, but more significantly the underlying paradigms will also change as increases in problem scale introduce new bottlenecks requiring essentially new techniques for resolution. As a result, our systems must include plans for change and accommodations for local inconsistency as changes take place. Software engineers must be educated to anticipate and accommodate regular change.

"Engineering" in Software Engineering

Engineering is the application of scientific and technical knowledge to the creation of effective systems that meet practical goals. Engineering disciplines have elements of both synthesis and analysis. In software engineering, synthesis includes design, programming, and integration; analysis includes requirement definition, evaluation, and measurement. Good engineering relies on a combination of underlying scientific principles, technical know-how and experience, and a pragmatic concern with effectiveness and utility. Although the field is gradually maturing, the description "software engineering" is still more an aspiration than an accomplishment.

Traditional methods of software development are *ad hoc* and labor-intensive. They will not be adequate to satisfy the increased demands on computing systems and the complexity of the resulting systems. Software engineering must move to a technology-intensive basis that draws on scientifically-based models and theories; it must be prepared to take advantage of advances in these areas as they become available. The education of software engineers is critical to this progress, for good ideas achieve practical utility only in the hands of people who use them wisely.

Over the past two decades a shift to methods based on scientific models has taken place in many aspects of programming-in-the-small. Algorithms and data structures were originally created in an *ad hoc* fashion, but regular use revealed patterns that could be organized systematically and in time provide a basis for formal theories. Some of the earliest formal models supported the analysis of algorithms. Our understanding of algorithms for certain problem domains is now quite well-structured, we can analyze the performance of specific algorithms, and we know theoretical limits on performance in many cases. Similarly, a theory to support abstract data types emerged during the 1970's. In the late 1960's computer scientists recognized the importance of good representations and their associated data structures. Refining this insight to a theory of abstract data types took about a decade; it required advances in formal specification, programming languages, verification, and programming methodology. Undergraduate computer science stu-

dents should now routinely master algorithmic analysis and abstract data types; it is now reasonable (but not entirely realistic) to expect the material to be applied in routine practice.

Sound theories can also contribute significantly to our ability to construct software systems. For example, the compiler for a programming language is a medium-sized system with a structure that is now well understood. Whereas in the early 1960's the construction of a compiler was a significant achievement, compilers are now often constructed routinely. Good theoretical understanding of syntax developed in the 1960's led to effective techniques for constructing parsers in the 1970's, first manually and more recently automatically. Similarly, good theories for programming language semantics and type structures developed in the 1970's are now leading to automation of other stages of compiler construction.

Although programming-in-the-large has a somewhat shorter history, formal models are beginning to emerge for the information management problems in that domain. For example, configuration management and version control began on an *ad hoc* basis with simple tools for organized (and often massive) recompilation, but at least a few models of system configuration and remanufacture are guiding the construction of software tools. The theoretical basis not only shows how to manage dependency information to reconstruct a system correctly, it also supports more efficient strategies of system reconstruction by avoiding unnecessary steps (e.g., recompilation of modules in which the only changes were comments or which depend only on unchanged portions of modules that were changed).

These examples give the flavor of the progress toward sound foundations for software engineering. There are clearly many areas in which the models, theories, and methodologies are still primitive. However, the power of soundly based theories in at least a few areas offers encouragement for developing and refining theories in other areas.

In Search of Software Engineering Principles

A scientist or engineer instinctively attempts to formalize principles in the form of mathematical laws, and it would be convenient if software

engineering could similarly be derived from a set of primitive equations. However, software engineering includes substantial social and organizational components — both behavioral and aesthetic — and it studies artificial constructs not constrained by the physical laws of materials. As a consequence, the models and theories of the field take many forms. We find good use for

- formal (mathematical) and informal theories
- structural and empirical models
- quantitative and qualitative evaluations
- synthetic and analytic principles
- algorithms and paradigms for design and human behavior
- strong and weak methods
- deep and shallow systems.

In general, the foundations of the field — the principles, models, and theories — should be systematic, codified, and abstracted from the examples where we learned them. These foundations should transcend changes of orders of magnitude in current technology, current problems, or current practice. It should be reasonably easy to teach these foundations to others.

At the current stage of software engineering's development, principles, models and theories are not yet available for all aspects of the discipline. Pragmatics lead us to develop and maintain software through a combination of principles and *ad hoc* techniques:

- principles that transcend current practice and current technology
- rules of thumb that guide current practice by codifying useful patterns
- methodologies that mechanize elements of current practice but do not generalize
- hacks

Good practice calls for drawing on techniques that lie as high on this list as possible.

Precise or detailed description of a technique does not make it a principle. For example, the waterfall model for software development (Figure 2) is a methodology, or mechanization of current practice, and not a principle.

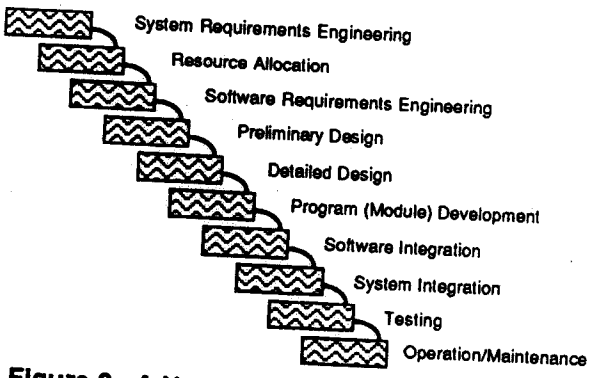

Figure 2. A Non-principle

Software engineering education should accommodate the current state of the field by presenting the strongest principles available in the context of the best current practice. Respect for the students and the state of the field require the material to be presented with honest assessments of the strengths and weaknesses of the techniques. In order for the material to be useful when current practice is obsolete, the selection of material in a software engineering curriculum should favor those areas in which principles have developed; a good curriculum should refrain from simply teaching current practice in the absence of unifying principles.

The position papers for the SEI Education Workshop contained many exhortations about the need for principles, but few concrete examples. Discussions during the workshop brought out some more examples. I will survey some of the suggestions for the guiding ideas that are variously called principles, theories, models, rules, paradigms, laws and methods.

First, most of software engineering seems to share a few attributes. These are often implicit in attitudes and in selection of techniques rather than subjects of explicit discussion. These underlying principles show that software engineering is:

- *Reductionistic:* We believe that problems in software engineering can be decomposed into successively smaller subproblems and that the solutions to the subproblems can be recombined to obtain a solution to the larger problem. We believe that this accounts for the phenomena that we deal with and that no "vital spark" is required or lost in decomposition. We also generally believe in reproducibility of effect — that the same initial conditions and inputs always yield the same result — though this tends to break down in systems so large that the initial condition cannot be precisely specified.
- *Discrete:* The problems and artifacts of software engineering are discrete, rather than continuous. We don't deal with infinitesimals or limits. Although we sometimes model those effects, we do so with definite limits on accuracy. As a consequence, our reasoning relies on case analysis, induction, and abstraction rather than, for example, extrapolation and interpolation.
- *Non-universal:* We believe in the existence of knowledge extrinsic to software engineering. We do not believe that software engineering or computer science is a universal discipline in the sense that it must eventually account for all phenomena in the world. As a result we must deal with transition problems at the boundary of the field; these are often also the boundaries between discrete and continuous phenomena.
- *Incompletely quantifiable:* Although we try to treat software engineering as an engineering discipline and we use quantitative models and measures wherever possible, we recognize that aesthetic considerations must also be respected. This is particularly true of the design aspects of the field.
- *Computationally limited:* Software engineering is incompletely quantifiable not only because of aesthetic requirements but also because of fundamental incompleteness of the underlying mathematics. We have not only theorems about undecidability but also demonstrations of the intrinsic completeness of testing strategies.

Some approaches to problems appear consistently throughout software engineering but appear to be techniques that generally work rather than underlying principles that dictate what solutions must be. These articles of philosophy include:

- *Engineering discipline:* As noted above, calling the field "software engineering" is still more an aspiration than an achievement. Nevertheless, we find that it is worth while to apply soundly-based models and techniques wherever we can.
- *Abstraction:* Abstraction is suppression of detail. Good abstraction is suppression of detail that is, to the current audience, not significant. We use abstraction not only as an approach to managing the complexity of the systems that we develop but also as an approach to designing the interfaces to those systems. We believe that a computer system should allow its user to focus on the user's real problems rather than on the operation of the system.
- *Defect prevention:* We generally follow a strategy of defect prevention rather than of defect removal. This is an approach rooted in utility rather than in principle: it is most often less expensive and less nuisance to build systems correctly in the first place rather than debug them after the fact. However, this may not always be true; for example, the use of rule-based systems to develop prototypes by iteratively adding information about a complex application and testing the prototype seems to be appropriate in many cases.
- *Reusability:* Because of the creative effort involved, we believe that it is better to reuse system components than to build them from scratch. This is sometimes called the "buy-don't-build" philosophy. In fact, this is an observation about economics and utility rather than a universal truth. However, when we start designing with reuse in mind, we will in effect be constructing theories that explain small application domains; the theories will be expressed in whatever form the reusable code takes.

Some areas of software engineering rely extensively on formal theories. These tend to be the older areas in which our understanding of the material has had longer to evolve. Some of these well developed theories include formal syntax and semantics, various kinds of logics, the theory of computation, formal specification and verification, the theory of algorithms, and type theory. Programming languages are often based on these theories, and we now recognize a number of programming paradigms. The more traditional paradigms such as applicative or functional programming and imperative programming are being

joined by object-oriented programming, message-based systems, constraint systems, and rule-based systems.

In other areas, only certain problems have been treated systematically for long enough to develop good models. These models are sometimes structural, as are the queuing-theoretic models for performance evaluation. In other cases the models are empirical, as are certain disk scheduling algorithms and cost estimation models. In the long run the structural models will best meet the test of surviving order-of-magnitude changes in technology or practice, but empirical models are welcome aids in the meanwhile.

Unfortunately, there are many areas of software engineering in which sound models or theories have not yet evolved. In these cases the best practice is *ad hoc*. We should be cautious about the role these practices play in software engineering education. On the one hand, they represent the best of current practice; on the other, they cannot be expected to be durable. Reasonable compromises may involve teaching the nature of the problems without dwelling on the details of the *ad hoc* solutions.

Goals and Objectives for Software Engineering Education

Software engineering is a part of computer science that draws heavily on mathematics, engineering, management, economics, communication, law, cognitive psychology, and design. It inherits a dilemma from computer science: changes in problems, technologies, and methods are an intrinsic part of the field, so the student and teacher are always aiming at a moving target. Since we are constantly assimilating new technologies, we are always on the leading edge of the learning curve for the current technology. This makes it critical for practicing engineers to deal comfortably with change.

Software engineers are educated in colleges and universities, in continuing education programs, and in in-house programs of individual companies. Although these programs reach rather different audiences, they address the same body of knowledge. As a result, a unified curriculum design may suffice to set agendas, but different organizations and presentation styles may be required for different audiences.

Whether software engineers learn this material at the beginning of their careers or afterward, they must be able to function immediately as professionals and to grow as the discipline evolves. Since software engineering is becoming a scientifically-based discipline, students must be educated in the fundamental principles not only of computer science but also of the other fields that contribute heavily to software engineering.

Following the Carnegie Plan for education [27, 78], we can state objectives for any software engineering curriculum, whether it be offered in academia or industry, whether it be a degree program or continuing education. We need a curriculum through which a student can acquire:

- A thorough and integrated understanding of the fundamental conceptual material of software engineering and the ability to apply this knowledge to the formulation and solution of real problems in software engineering.
- A genuine competence in the orderly ways of thinking which scientists and engineers have always used in reaching sound, creative conclusions; with this competence, the student will be able to make decisions in higher professional work and as a citizen.
- An ability to learn independently with scholarly orderliness, so that after graduation the student will be able to grow in wisdom and keep abreast of the changing knowledge and problems of his or her profession and the society in which he or she lives.
- A philosophical outlook, breadth of knowledge, and sense of values which will increase the student's understanding and enjoyment of life and enable each student to recognize and deal effectively with the human, economic, and social aspects of his or her professional problems.
- An ability to communicate ideas to others.

The focus of the curriculum should be on a *liberal professional education with emphasis on design and problem-solving skills.* Describing the education as "liberal" recognizes the importance of exposure to topics outside the student's specialty; at the graduate level this may be somewhat more narrowly directed at material related to software engineering than at the undergraduate level. Liberal education includes communica-

tion skills, both for understanding the work of others and for communicating one's own work. Describing the education as "professional" recognizes the legitimate motivations of students who value education because they can apply it rather than for pure intellectual enjoyment. The "design" component of the education recognizes the synthetic, creative aspect of the profession. "Problem-solving skills" refers to the ability to apply general concepts and methods from a variety of disciplines to all kinds of problems, abstract as well as practical, whose solutions require thought, insight, and creativity. Thus "problems" can range from the proof of a theorem to the design and construction of a specialized computer program and "skills" refers to creative intellectual ability, not merely the ability to perform repetitive routine actions.

A Word of Caution

The greatest danger to software engineering curriculum designers is lack of imagination. If we are too narrow, too shortsighted, or too low in our aspirations, we will deprive the field of the skills it needs to satisfy society's requirements for broader scope and larger scale in computer-based systems.

Acknowledgements

My understanding of software engineering education and the principles that support software engineering has come from many discussions with other computer scientists, especially my colleagues at Carnegie-Mellon. Particular insights in this paper came from discussions with Bill Wulf, Allen Newell, Nico Habermann, and Jim Horning. Jim Tomayko supplied data for the software demand figure. The discussion of software engineering principles evolved substantially during two long sessions with the Principles Working Group at the SEI Education Workshop.