Prospects for an Engineering Discipline of Software

Mary Shaw, *Carnegie Mellon University*

Software engineering is not yet a true engineering discipline, but it has the potential to become one. Older engineering fields suggest the character software engineering might have.

The term "software engineering" was coined in 1968 as a statement of aspiration — a sort of rallying cry. That year, the North Atlantic Treaty Organization convened a workshop by that name to assess the state and prospects of software production. Capturing the imagination of software developers, the NATO phrase "software engineering" achieved popularity during the 1970s. It now refers to a collection of management processes, software tooling, and design activities for software development. The resulting practice, however, differs significantly from the practice of older forms of engineering.

What is engineering?

"Software engineering" is a label applied to a set of current practices for development. But using the word "engineering" to describe this activity takes considerable liberty with the common use of that term. The more customary usage refers to the disciplined application of scientific knowledge to resolve conflicting constraints and requirements for problems of immediate, practical significance.

Definitions of "engineering" abound. Although details differ, they share some common clauses:

- *Creating cost-effective solutions* ... Engineering is not just about solving problems; it is about solving problems with economical use of all resources, including money.
- ... *to practical problems* ... Engineering deals with practical problems whose solutions matter to people outside the engineering domain — the customers.
- ... *by applying scientific knowledge* ... Engineering solves problems in a particular way: by applying science, mathematics, and design analysis.
- ... *to building things* ... Engineering emphasizes the solutions, which are usually tangible artifacts.
- ... *in the service of mankind.* Engineering not only serves the immediate

0740-7459/90/1100/0015/$01.00 © 1990 IEEE

customer, but it also develops technology and expertise that will support the society.

Engineering relies on codifying scientific knowledge about a technological problem domain in a form that is directly useful to the practitioner, thereby providing answers for questions that commonly occur in practice. Engineers of ordinary talent can then apply this knowledge to solve problems far faster than they otherwise could. In this way, engineering shares prior solutions rather than relying always on virtuoso problem solving.

Engineering practice enables ordinary practitioners so they can create sophisticated systems that work — unspectacularly, perhaps, but reliably. The history of development is marked by both successes and failures. The successes have often been virtuoso performances or the result of diligence and hard work. The failures have often reflected poor understanding of the problem to be solved, mismatch of solution to problem, or inadequate follow-through from design to implementation. Some failed by never working, others by overrunning cost and schedule budgets.

In current software practice, knowledge about techniques that work is not shared effectively with later projects, nor is there a large body of development knowledge organized for ready reference. Computer science has contributed some relevant theory, but practice proceeds largely independently of this organized knowledge. Given this track record, there are fundamental problems with the use of the term "software engineer."

Routine and innovative design. Engineering design tasks are of several kinds. One of the most significant distinctions separates routine from innovative design. Routine design involves solving familiar problems, reusing large portions of prior solutions. Innovative design, on the other hand, involves finding novel solutions to unfamiliar problems. Original designs are much more rarely needed than routine designs, so the latter is the bread and butter of engineering.

Most engineering disciplines capture, organize, and share design knowledge to make routine design simpler. Handbooks and manuals are often the carriers of this organized information. But current notations for software designs are not adequate for the task of both recording and communicating designs, so they fail to provide a suitable representation for such handbooks.

Software in most application domains is treated more often as original than routine — certainly more so than would be necessary if we captured and organized what we already know. One path to increased productivity is identifying applications that could be routine and developing appropriate support.

The current focus on reuse emphasizes capturing and organizing existing knowledge of a particular kind: knowledge expressed in the form of code. Indeed, subroutine libraries — especially of system calls and general-purpose mathematical routines — have been a staple of programming for decades. But this knowledge cannot be useful if programmers do not know about it or are not encouraged to use it. Furthermore, library components require more care in design, implementation, and documentation than similar components that are simply embedded in systems.

Given our track record, there are fundamental problems with the use of the term "software engineer."

Practitioners recognize the need for mechanisms to share experience with good designs. This cry from the wilderness appeared on the Software Engineering News Group, a moderated electronic mailing list:

"In Chem E, when I needed to design a heat exchanger, I used a set of references that told me what the constants were ... and the standard design equations. ...

"In general, unless I, or someone else in my [software-] engineering group, has read or remembers and makes known a solution to a past problem, I'm doomed to recreate the solution. ... I guess ... the critical difference is the ability to put together little pieces of the problem that are relatively well known, without having to generate a custom solution for every application. ...

"I want to make it clear that I am aware of algorithm and code libraries, but they are incomplete solutions to what I am describing. (There is no *Perry's Handbook for Software Engineering.*)"

This former chemical engineer is complaining that software lacks the institutionalized mechanisms of a mature engineering discipline for recording and disseminating demonstrably good designs and ways to choose among design alternatives. (*Perry's Chemical Engineering Handbook*, published by McGraw-Hill, is the standard design handbook for chemical engineering; it is about four inches thick and printed in tiny type on 8.5″ × 11″ tissue paper.)

Model for the evolution of an engineering discipline. Historically, engineering has emerged from ad hoc practice in two stages: First, management and production techniques enable routine production. Later, the problems of routine production stimulate the development of a supporting science; the mature science eventually merges with established practice to yield professional engineering practice. Figure 1 shows this model.

The exploitation of a technology begins with craftsmanship: A set of problems must be solved, and they get solved any which way. They are solved by talented amateurs and by virtuosos, but no distinct professional class is dedicated to problems of this kind. Intuition and brute force are the primary movers in design and construction. Progress is haphazard, particularly before the advent of good communication; thus, solutions are invented and reinvented. The transmission of knowledge between craftsmen is slow, in part because of underdeveloped communications, but also because the talented amateurs often do not recognize any special need to communicate.

Nevertheless, ad hoc practice eventually moves into the folklore. This craft stage of development sees extravagant use of available materials. Construction or manufacture is often for personal or local use or for barter, but there is little or no large-scale production in anticipation of resale. Community barn raisings are an example

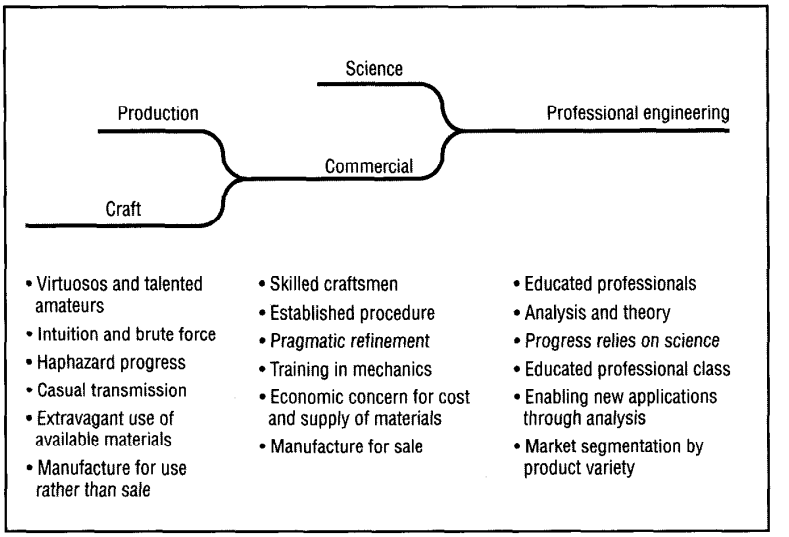

Figure 1. Evolution of an engineering discipline. The lower lines track the technology, and the upper lines show how the entry of production skills and scientific knowledge contribute new capability to the engineering practice.

of this stage; so is software written by application experts for their own ends.

At some point, the product of the technology becomes widely accepted and demand exceeds supply. At that point, attempts are made to define the resources necessary for systematic commercial manufacture and to marshal the expertise for exploiting these resources. Capital is needed in advance to buy raw materials, so financial skills become important, and the operating scale increases over time.

As commercial practice flourishes, skilled practitioners are required for continuity and for consistency of effort. They are trained pragmatically in established procedures. Management may not know why these procedures work, but they know the procedures *do* work and how to teach people to execute them.

The procedures are refined, but the refinement is driven pragmatically: A modification is tried to see if it works, then incorporated in standard procedure if it does. Economic considerations lead to concerns over the efficiency of procedures and the use of materials. People begin to explore ways for production facilities to exploit the technology base; economic issues often point out problems in commercial practice. Management strategies for controlling development fit at this point of the model.

The problems of current practice often stimulate the development of a corresponding science. There is frequently a strong, productive interaction between commercial practice and the emerging science. At some point, the science becomes sufficiently mature to be a significant contributor to the commercial practice. This marks the emergence of engineering practice in the sense that we know it today — sufficient scientific basis to enable a core of educated professionals so they can apply the theory to analysis of problems and synthesis of solutions.

For most disciplines, this emergence occurred in the 18th and early 19th centuries as the common interests in basic physical understandings of natural science and engineering gradually drew together. The reduction of many empirical engineering techniques to a more scientific basis was essential to further engineering progress. And this liaison stimulated further advances in natural science. "An important and mutually stimulating tie-up between natural and engineering science, a development [that] had been discouraged for centuries by the long-dominant influence of early Greek thought, was at long last consummated," wrote historian James Kip Finch.[1]

The emergence of an engineering discipline lets technological development pass limits previously imposed by relying on intuition; progress frequently becomes dependent on science as a forcing function. A scientific basis is needed to drive analysis, which enables new applications and even market segmentation via product variety. Attempts are made to gain enough control over design to target specific products on demand.

Thus, engineering emerges from the commercial exploitation that supplants craft. Modern engineering relies critically on adding scientific foundations to craft and commercialization. Exploiting technology depends not only on scientific engineering but also on management and the marshaling of resources. Engineering and science support each other: Engineering generates good problems for science, and science, after finding good problems in the needs of practice, returns workable solutions. Science is often not driven by the immediate needs of engineering; however, good scientific problems often follow from an understanding of the problems that the engineering side of the field is coping with.

The engineering practice of software has recently come under criticism for lacking a scientific basis. The usual curriculum has been attacked for neglecting mathematics[2] and engineering science.[3] Although current software practice does not match the usual expectations of an engineering discipline, the model described here suggests that vigorous pursuit of applicable science and the reduction of that science to practice *can* lead to a sound engineering discipline of software.

Examples from traditional engineering. Two examples make this model concrete: the evolution of engineering disciplines as demonstrated by civil and chemical engineering. The comparison of the two is also illuminating, because they have very different basic organizations.

Civil engineering: a basis in theory. Originally so-called to distinguish it from military engineering, civil engineering included all of civilian engineering until the middle of the 19th century. A divergence of interests led engineers specializing in other technologies to break away, and today civil engineers are the technical experts of the construction industry. They are concerned primarily with large-scale, capital-intensive construction efforts, like buildings, bridges, dams, tunnels, canals, highways, railroads, public water supplies, and sanitation. As a rule, civil-engineering efforts involve well-defined task groups that use appropriate tools and technolo-

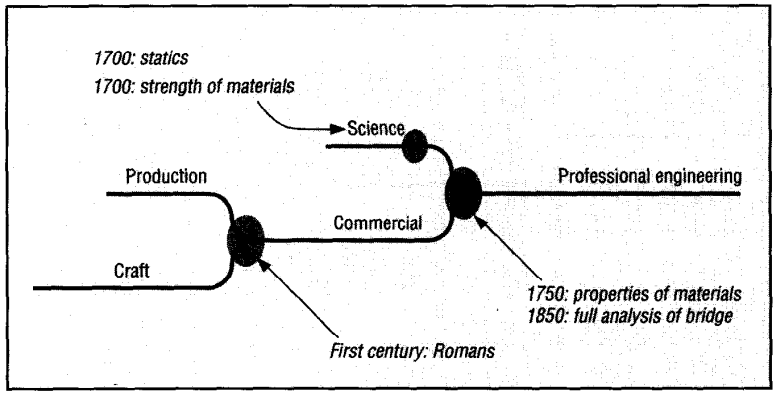

Figure 2. Evolution of civil engineering.

gies to execute well-laid plans.

Although large civil structures have been built since before recorded history, only in the last few centuries has their design and construction been based on theoretical understanding rather than on intuition and accumulated experience. Neither the artisans of the Middle Ages nor of the ancient world showed any signs of the deliberate quantitative application of mathematics to determine the dimensions and shapes that characterizes modern civil engineering. But even without formal understanding, they documented pragmatic rules for recurring elements. Practical builders had highly developed intuitions about statics and relied on a few empirical rules.

The scientific revolution of the Renaissance led to serious attempts by Galileo Galilei, Filippo Brunelleschi, and others to explain structures and why they worked. Over a period of about 200 years, there were attempts to explain the composition of forces and bending of a beam. However, progress was slowed for a long time by problems in formulating basic notions like force, particularly the idea that gravity could be treated as just another force like all the others. Until the basic concepts were sorted out, it was not possible to do a proper analysis of the problem of combining forces (using vector addition) that we now teach to freshmen, nor was it possible to deal with strengths of materials.

Around 1700, Pierre Varignon and Isaac Newton developed the theory of statics to explain the composition of forces and Charles Augustin de Coulomb and Louis Marie Henri Navier explained bending with the theory of strength of materials. These now provide the basis for civil engineering. By the middle of the 18th century, civil engineers were tabulating properties of materials.

The mid-18th century also saw the first attempts to apply exact science to practical building. Pope Benedict ordered an analysis of St. Peter's dome in 1742 and 1743 to determine the cause of cracks and propose repairs; the analysis was based on the principle of virtual displacement and was carried out precisely (although the model is now known to fail to account properly for elasticity). By 1850, it was possible for Robert Stephenson's Britannia Tubular Bridge over the Menai Strait between Wales and England to be subjected to a formal structural analysis.

Thus, even after the basic theories were in hand, it took another 150 years before the theory was rich enough and mature enough to have direct utility at the scale of a bridge design.

Civil engineering is thus rooted in two scientific theories, corresponding to two classical problems. One problem is the composition of forces: finding the resultant force when multiple forces are combined. The other is the problem of bending: determining the forces within a beam supported at one end and weighted at the other. Two theories, statics and strength of materials, solve these problems; both were developed around 1700. Modern civil engineering is the application of these theories to the problem of constructing buildings.

"For nearly two centuries, civil engineering has undergone an irresistible transition from a traditional craft, concerned with tangible fashioning, towards an abstract science, based on mathematical calculation. Every new result of research in structural analysis and technology of materials signified a more rational design, more economic dimensions, or entirely new structural possibilities. There were no apparent limitations to the possibilities of analytical approach; there were no apparent problems in building construction [that] could not be solved by calculation," wrote Hans Straub in his history of civil engineering.[4]

You can date the transition from craft to commercial practice to the Romans' extensive transportation system of the first century. The underlying science emerged about 1700, and it matured to successful application to practice sometime between the mid-18th century and the mid-19th century. Figure 2 places civil engineering's significant events on my model of engineering evolution.

Chemical engineering: a basis in practice. Chemical engineering is a very different kind of engineering than civil engineering. This discipline is rooted in empirical observations rather than in a scientific theory. It is concerned with practical problems of chemical manufacture; its scope covers the industrial-scale production of chemical goods: solvents, pharmaceuticals, synthetic fibers, rubber, paper, dyes, fertilizers, petroleum products, cooking oils, and so on. Although chemistry provides the specification and design of the basic reactions, the chemical engineer is responsible for scaling the reactions up from laboratory scale to factory scale. As a result, chemical engineering depends as heavily on mechanical engineering as on chemistry.

Until the late 18th century, chemical production was largely a cottage industry. The first chemical produced at industrial scale was alkali, which was required for the manufacture of glass, soap, and textiles. The first economical industrial process for alkali emerged in 1789, well before the atomic theory of chemistry explained the underlying chemistry. By the mid-19th century, industrial production of dozens of chemicals had turned the British Midlands into a chemical-manufacturing district. Laws were passed to control the resulting pollution, and pollution-control inspectors, called alkali inspectors, monitored plant compliance.

One of these alkali inspectors, G.E. Davis, worked in the Manchester area in the late 1880s. He realized that, although the plants he was inspecting manufactured dozens of different kinds of chemicals, there were not dozens of different

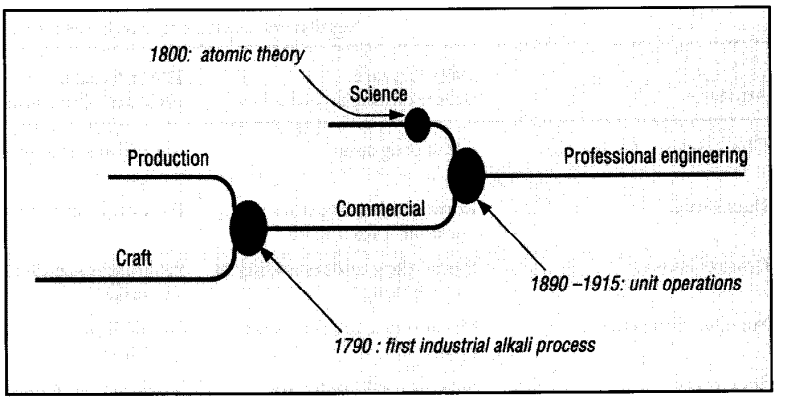

Figure 3. Evolution of chemical engineering.

procedures involved. He identified a collection of functional operations that took place in those processing plants and were used in the manufacture of different chemicals. He gave a series of lectures in 1887 at the Manchester Technical School. The ideas in those lectures were imported to the US by the Massachusetts Institute of Technology in the latter part of the century and form the basis of chemical engineering as it is practiced today. This structure is called *unit operations*; the term was coined in 1915 by Arthur D. Little.

The fundamental problems of chemical engineering are the quantitative control of large masses of material in reaction and the design of cost-effective industrial-scale processes for chemical reactions.

The unit-operations model asserts that industrial chemical-manufacturing processes can be resolved into a relatively few units, each of which has a definite function and each of which is used repeatedly in different kinds of processes. The unit operations are steps like filtration and clarification, heat exchange, distillation, screening, magnetic separation, and flotation. The basis of chemical engineering is thus a pragmatically determined collection of very high-level functions that adequately and appropriately describe the processes to be carried out.

"Chemical engineering as a science ... is not a composite of chemistry and mechanical and civil engineering, but a science of itself, the basis of which is those unit operations [that] in their proper sequence and coordination constitute a chemical process as conducted on the industrial scale. These operations ... are not the subject matter of chemistry as such nor of mechanical engineering. Their treatment is in the quantitative way, with proper exposition of the laws controlling them and of the materials and equipment concerned in them," the American Institute of Chemical Engineers Committee on Education wrote in 1922.[5]

This is a very different kind of structure from that of civil engineering. It is a pragmatic, empirical structure — not a theoretical one.

You can date the transition from craft to commercial practice to the introduction of the LeBlanc process for alkali in 1789. The science emerged with the British chemist John Dalton's atomic theory in the early 19th century, and it matured to successful merger with large-scale mechanical processes in the 1890s. Figure 3 places chemical engineering's significant events on my model.

Software technology

Where does software stand as an engineering discipline? For software, the problem is appropriately an engineering problem: creating cost-effective solutions to practical problems, building things in the service of mankind.

Information processing as an economic force. The US computer business — including computers, peripherals, packaged software, and communications — was about $150 billion in 1989 and is projected to be more than $230 billion by 1992. The packaged-software component is projected to grow from $23.7 billion to $37.5 billion in this period, according to the Data Analysis Group's fourth-quarter 1989 forecasts. Services, including systems integration and in-house development, are not included in these figures.

Worldwide, software sales amounted to about $65 billion in 1989. This does not include the value of in-house development, which is a much larger activity. World figures are hard to estimate, but the cost of in-house software in the US alone may be in the range of $150 billion to $200 billion.[6] It is not clear how much modification after release (so-called "maintenance") is included in this figure. Thus, software is coming to dominate the cost of information processing.

The economic presence of information processing also makes itself known through the actual and opportunity costs of systems that do *not* work. Examples of costly system failures abound. Less obvious are the costs of computing that is not even tried: development backlogs so large that they discourage new requests, gigabytes of unprocessed raw data from satellites and space probes, and so on. Despite very real (and substantial) successes, the litany of mismatches of cost, schedule, and expectations is a familiar one.

Growing role of software in critical applications. The US National Academy of Engineering recently selected the 10 greatest engineering achievements of the last 25 years.[7] Of the 10, three are informatics achievements: communications and information-gathering satellites, the microprocessor, and fiber-optic communication. Two more are direct applications of computers: computer-aided design and manufacturing and the computerized axial tomography scan. And most of the rest are computer-intensive: the Moon landing, advanced composite materials, the jumbo jet, lasers, and the application of genetic engineering to produce new pharmaceuticals and crops.

The conduct of science is increasingly driven by computational paradigms standing on equal footing with theoretical and experimental paradigms. Both scientific and engineering disciplines require very sophisticated computing. The demands are often stated in terms of raw processing power — "an exaflop (10^{18}) processor with teraword memory," "a petabyte (10^{15}) of storage," as one article put it[8] — but the supercomputing community is increasingly recognizing development, not mere raw processing, as a critical bottleneck.

Because of software's pervasive presence, the appropriate objective for its developers should be the effective delivery of computational capability to real users in forms that match their needs. The dis-

Table 1.
Significant shifts in research attention.

Attribute	1960 ± 5 years: programming any-which-way	1970 ± 5 years: programming-in-the small	1980 ± 5 years: programming-in-the-large
Characteristic problems	Small programs	Algorithms and programming	Interfaces, management system structures
Data issues	Representing structure and symbolic information	Data structures and types	Long-lived databases, symbolic as well as numeric
Control issues	Elementary understanding of control flows	Programs execute once and terminate	Program assemblies execute continually
Specification issues	Mnemonics, precise use of prose	Simple input/output specifications	Systems with complex specifications
State space	State not well understood apart from control	Small, simple state space	Large, structured state space
Management focus	None	Individual effort	Team efforts, system lifetime maintenance
Tools, methods	Assemblers, core dumps	Programming language, compilers, linkers, loaders	Environments, integrated tools, documents

tinction between a system's computational component and the application it serves is often very soft — the development of effective software now often requires substantial application expertise.

Maturity of development techniques. Our development abilities have certainly improved over the 40 or so years of programming experience. Progress has been both qualitative and quantitative. Moreover, it has taken different forms in the worlds of research and practice.

One of the most familiar characterizations of this progress has been the shift from programming-in-the-small to programming-in-the-large. It is also useful to look at a shift that took place 10 years before that, from programming-any-which-way to programming-in-the-small. Table 1 summarizes these shifts, both of which describe the focus of attention of the software research community.

Before the mid-1960s, programming was substantially ad hoc; it was a significant accomplishment to get a program to run at all. Complex software systems were created — some performed very well — but their construction was either highly empirical or a virtuoso activity. To make programs intelligible, we used mnemonics, we tried to be precise about writing comments, and we wrote prose specifications. Our emphasis was on small programs, which was all we could handle predictably.

We did come to understand that computers are symbolic information processors, not just number crunchers — a significant insight. But the abstractions of algorithms and data structures did not emerge until 1967, when Donald Knuth showed the utility of thinking about them in isolation from the particular programs that happened to implement them.

A similar shift in attitudes about specifications took place at about the same time, when Robert Floyd showed how attaching logical formulas to programs allows formal reasoning about the programs. Thus, the late 1960s saw a shift from crafting monolithic programs to an emphasis on algorithms and data structures. But the programs in question were still simple programs that execute once and then terminate.

You can view the shift that took place in the mid-1970s from programming-in-the-small to programming-in-the-large in much the same terms. Research attention turned to complex systems whose specifications were concerned not only with the functional relations of the inputs and outputs, but also with performance, reliability, and the states through which the system passed. This led to a shift in emphasis to interfaces and managing the programming process.

In addition, the data of complex systems often outlives the programs and may be more valuable, so we learned that we now have to worry about integrity and consistency of databases. Many of our programs (for example, the telephone switching system or a computer operating system) should *not* terminate; these systems require a different sort of reasoning than do programs that take input, compute, produce output, and terminate. In systems that run indefinitely, the sequence of system states is often much more important than the (possibly undesirable) termination condition.

The tools and techniques that accompanied the shift from programming-any-which-way to programming-in-the-small provided first steps toward systematic, routine development of small programs; they also seeded the development of a science that has matured only in the last decade. The tools and techniques that accompanied the shift from programming-in-the-small to programming-in-the-large were largely geared to supporting groups of programmers working together in orderly ways and to giving management a view into production processes. This directly supports the commercial practice of development.

Practical development proceeded to large complex systems much faster than the research community did. For example, the Sage missile-defense system of the 1950s and the Sabre airline-reservation system of the 1960s were successful interactive systems on a scale that far exceeded the maturity of the science. They appear to have been developed by excellent engineers who understood the requirements well and applied design and development methods from other (like electrical) engineering disciplines. Modern development methodologies are management procedures intended to guide large numbers of developers through similar disciplines.

The term "software engineering" was introduced in 1968 to name a conference

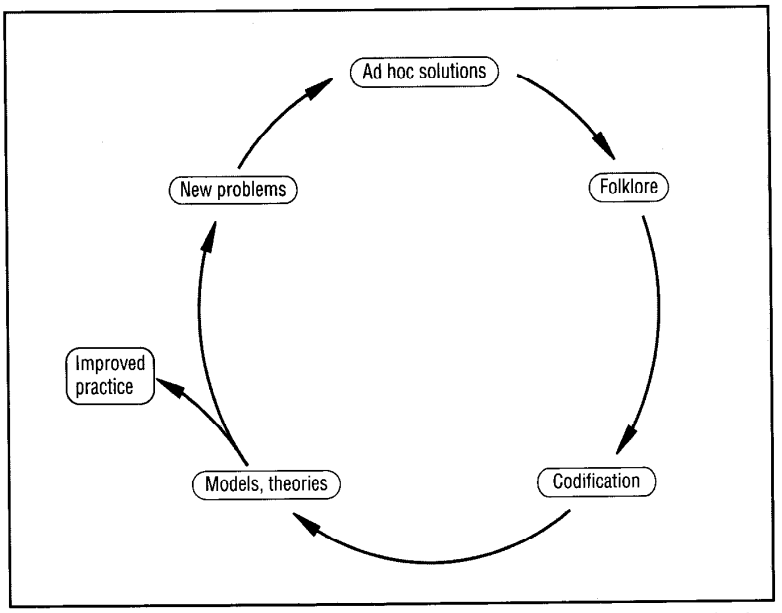

Figure 4. Cycle of how good software models develop as a result of the interaction between science and engineering.

convened by NATO to discuss problems of software production.[9] Despite the label, most of the discussion dealt with the challenge of progressing from the craft stage to the commercial stage of practice. In 1976, Barry Boehm proposed the definition of the term as "the practical application of scientific knowledge in the design and construction of computer programs and the associated documentation required to develop, operate, and maintain them."[10] This definition is consistent with traditional definitions of engineering, although Boehm noted the shortage of scientific knowledge to apply.

Unfortunately, the term is now most often used to refer to life-cycle models, routine methodologies, cost-estimation techniques, documentation frameworks, configuration-management tools, quality-assurance techniques, and other techniques for standardizing production activities. These technologies are characteristic of the commercial stage of evolution — "software management" would be a much more appropriate term.

Scientific basis for engineering practice. Engineering practice emerges from commercial practice by exploiting the results of a companion science. The scientific results must be mature and rich enough to model practical problems. They must also be organized in a form that is useful to practitioners. Computer science has a few models and theories that are ready to support practice, but the packaging of these results for operational use is lacking.

Maturity of supporting science. Despite the criticism sometimes made by software producers that computer science is irrelevant to practical software, good models and theories *have* been developed in areas that have had enough time for the theories to mature.

In the early 1960s, algorithms and data structures were simply created as part of each program. Some folklore grew up about good ways to do certain sorts of things, and it was transmitted informally. By the mid-1960s, good programmers shared the intuition that if you get the data structures right, the rest of the program is much simpler. In the late 1960s, algorithms and data structures began to be abstracted from individual programs, and their essential properties were described and analyzed.

The 1970s saw substantial progress in supporting theories, including performance analysis and correctness. Concurrently, the programming implications of these abstractions were explored; abstract-data-type research dealt with such issues as:

- Specifications: abstract models and algebraic axioms.
- Software structure: bundling representation with algorithms.
- Language issues: modules, scope, and user-defined types.
- Information hiding: protecting the integrity of information not in the specification.
- Integrity constraints: invariants of data structures.
- Composition rules: declarations.

Both sound theory and language support were available by the early 1980s, and routine good practice now depends on this support.

Compiler construction is another good example. In 1960, simply writing a compiler at all was a major achievement; it is not clear that we really understood what a higher level language was. Formal syntax was first used systematically for Algol-60, and tools for processing it automatically (then called compiler compilers, but now called parser generators) were first developed in the mid-1960s and made practical in the 1970s. Also in the 1970s, we started developing theories of semantics and types, and the 1980s have brought significant progress toward the automation of compiler construction.

Both of these examples have roots in the problems of the 1960s and became genuinely practical in the 1980s. It takes a good 20 years from the time that work starts on a theory until it provides serious assistance to routine practice. Development periods of comparable length have also preceded the widespread use of systematic methods and technologies like structured programming, Smalltalk, and Unix, as Sam Redwine and colleagues have shown.[11] But the whole field of computing is only about 40 years old, and many theories are emerging in the research pipeline.

Interaction between science and engineering. The development of good models within the software domain follows this pattern:

We engineers begin by solving problems any way we can. After some time, we distinguish in those ad hoc solutions things that usually work and things that do not usually work. The ones that do work enter the folklore: People tell each other about them informally. As the folklore becomes more and more systematic, we codify it as written heuristics and rules of procedure. Eventually, that codification becomes crisp enough to support models and theories, together with the associated mathematics. These can then help improve practice,

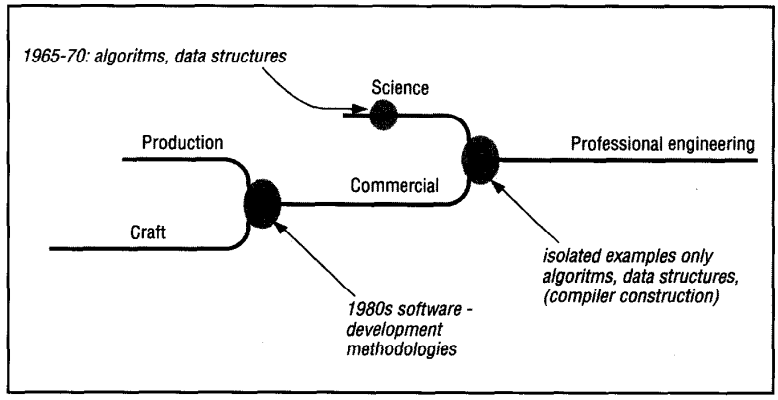

Figure 5. Evolution of software engineering.

and experience from that practice can sharpen the theories. Furthermore, the improvement in practice let us think about harder problems — which we first solve ad hoc, then find heuristics for, eventually develop new models and theories for, and so on. Figure 4 illustrates this cycle.

The models and theories do not have to be fully fleshed out for this process to assist practice: The initial codification of folklore may be useful in and of itself.

This progression is illustrated in the use of machine language for control flow in the 1960s. In the late 1950s and the early 1960s, we did not have crisp notions about what an iteration or a conditional was, so we laid down special-purpose code, building each structure individually out of test and branch instructions.

Eventually, a small set of patterns emerged as generally useful, generally easy to get right, and generally at least as good as the alternatives. Designers of higher level languages explicitly identified the most useful ones and codified them by producing special-purpose syntax. A formal result about the completeness of the structured constructs provided additional reassurance.

Now, almost nobody believes that new kinds of loops should be invented as a routine practice. A few kinds of iterations and a few kinds of conditionals are captured in the languages. They are taught as control concepts that go with the language; people use them routinely, without concern for the underlying machine code.

Further experience led to verifiable formal specifications of these statements' semantics and of the programs that used them. Experience with the formalization in turn refined the statements supported in programming languages. In this way, ad hoc practice entered a period of folklore and eventually matured to have conventional syntax and semantic theories that explain it.

Where is software? Where, then, does current software practice lie on the path to engineering? It is still in some cases craft and in some cases commercial practice. A science is beginning to contribute results, and, for isolated examples, you can argue that professional engineering is taking place. (Figure 5 shows where software practice fits on my model.)

That is not, however, the common case.

There are good grounds to expect that there will eventually be an engineering discipline of software. Its nature will be technical, and it will be based in computer science. Although we have not yet matured to that state, it is an achievable goal.

The next tasks for the software profession are

- to pick an appropriate mix of short-term, pragmatic, possible purely empirical contributions that help stabilize commercial practice and
- to invest in long-term efforts to develop and make available basic scientific contributions.

The profession must take five basic steps on its path to becoming a true engineering discipline:

Understand the nature of expertise. Proficiency in any field requires not only higher order reasoning skills but also a large store of facts together with a certain amount of context about their implications and appropriate use. Studies have demonstrated this across a wide range of problem domains, including medical diagnosis, physics, chess, financial analysis, architecture, scientific research, policy decision making, and others, as Herbert Simon described in the paper "Human Experts and Knowledge-Based Systems" presented at the 1987 IFIP Working Group 10.1 Workshop on Concepts and Characteristics of Knowledge-Based Systems.

An expert in a field must know about 50,000 chunks of information, where a chunk is any cluster of knowledge sufficiently familiar that it can be remembered rather than derived. Furthermore, in domains where there are full-time professionals, it takes no less than 10 years for a world-class expert to achieve that level of proficiency.[11]

Thus, fluency in a domain requires content and context as well as skills. In the case of natural-language fluency, E.D. Hirsch has argued that abstract skills have driven out content; students are expected (unrealistically) to learn general skills from a few typical examples rather than by a "piling up of information"; and intellectual and social skills are supposed to develop naturally without regard to the specific content.[12]

However, Hirsch wrote, specific information is important at all stages. Not only are the specific facts important in their own right, but they serve as carriers of shared culture and shared values. A software engineer's expertise includes facts about computer science in general, software design elements, programming idioms, representations, and specific knowledge about the program of current interest. In addition, it requires skill with tools: the language, environment, and support software with which this program is implemented.

Hirsch provided a list of some 5,000 words and concepts that represent the information actually possessed by literate Americans. The list goes beyond simple vocabulary to enumerate objects, concepts, titles, and phrases that implicitly invoke cultural context beyond their dictionary definitions. Whether or not you agree in detail with its composition, the list and accompanying argument demonstrate the need for connotations as well as denotations of the vocabulary.

Similarly, a programmer needs to know not only a programming language but also the system calls supported by the environment, the general-purpose libraries, the application-specific libraries, and how to combine invocations of these definitions effectively. The programmer must

Table 2.
Cost distributions for the thee ways to get a piece of information.

Method	Infrastructure cost	Initial-learning cost	Cost of use in practice
Memory	Low	High	Low
Reference	High	Low	Medium
Derivation	Medium-high	Medium	High

be familiar with the global definitions of the program of current interest and the rules about their use. In addition, a developer of application software must understand application-area issues.

Simply put, the engineering of software would be better supported if we knew better what specific content a software engineer should know. We could organize the teaching of this material so useful subsets are learned first, followed by progressively more sophisticated subsets. We could also develop standard reference materials as carriers of the content.

Recognize different ways to get information. Given that a large body of knowledge is important to a working professional, we as a discipline must ask how software engineers should acquire the knowledge, either as students or as working professionals. Generally speaking, there are three ways to get a piece of information you need: You can remember it, you can look it up, or you can derive it. These have different distributions of costs, as Table 2 shows.

Memorization requires a relatively large initial investment in learning the material, which is then available for instant use.

Reference materials require a large investment by the profession for developing both the organization and the content; each student must then learn how to use the reference materials and then do so as a working professional.

Deriving information may involve ad hoc creation from scratch, it may involve instantiation of a formal model, or it may involve inferring meaning from other available information. To the extent that formal models are available, their formulation requires a substantial initial investment. Students first learn the models, then apply them in practice. Because each new application requires the model to be applied anew, the cost in use may be very high.[13]

Each professional's allocation of effort among these alternatives is driven by what he has already learned, by habits developed during that education, and by the reference materials available. Today, general-purpose reference material for software is scarce, although documentation for specific computer systems, languages, and applications may be extensive. Even when documentation is available, however, it may be underused because it is poorly indexed or because developers have learned to prefer fresh derivation to use of existing solutions. The same is true of subroutine libraries.

Simply put, software engineering requires investment in the infrastructure cost — in creating the materials required to organize information, especially reference material for practitioners.

Encourage routine practice. Good engineering practice for routine design depends on the engineer's command of factual knowledge and design skills and on the quality of reference materials available. It also depends on the incentives and values associated with innovation.

Unfortunately, computer-science education has prepared developers with a background that emphasizes fresh creation almost exclusively. Students learn to work alone and to develop programs from scratch. They are rarely asked to understand software systems they have not written. However, just as natural-language fluency requires instant recognition of a core vocabulary, programming fluency should require an extensive vocabulary of definitions that the programmer can use familiarly, without repeated recourse to documentation.

Fred Brooks has argued that one of the great hopes for software engineering is the cultivation of great designers.[14] Indeed, innovative designs require great designers. But great designers are rare, and most designs need not be innovative. Systematic presentation of design fragments and techniques that are known to work can enable designers of ordinary talent to produce effective results for a wide range of more routine problems by using prior results (buying or growing, in Brooks's terms) instead of always building from scratch.

It is unreasonable to expect a designer or developer to take advantage of scientific theories or experience if the necessary information is not readily available. Scientific results need to be recast in operational form; the important information from experience must be extracted from examples. The content should include design elements, components, interfaces, interchange representations, and algorithms. A conceptual structure must be developed so the information can be found when it is needed. These facts must be augmented with analysis techniques or guidelines to support selection of alternatives that best match the problem at hand.

A few examples of well-organized reference materials already exist. For example, the summary flowchart of William Martin's sorting survey[15] captured in one page the information a designer needed to choose among the then-current sorting techniques. William Cody and William Waite's manual for implementing elementary mathematical functions[16] gives for each function the basic strategy and special considerations needed to adapt that strategy to various hardware architectures.

Although engineering has traditionally relied on handbooks published in book form, a software engineers' handbook must be on line and interactive. No other alternative allows for rapid distribution of updates at the rate this field changes, and no other alternative has the potential for smooth integration with on-line design tools. The on-line incarnation will require solutions to a variety of electronic-publishing problems, including distribution, validation, organization and search, and collection and distribution of royalties.

Simply put, software engineering would benefit from a shift of emphasis in which both reference materials and case studies of exemplary software designs are incorporated in the curriculum. The discipline must find ways to reward preparation of material for reference use and the development of good case studies.

Expect professional specializations. As software practice matures toward engineering, the body of substantive technical knowledge required of a designer or developer continues to grow. In some areas, it has long since grown large enough to

require specialization — for example, database administration was long ago separated from the corresponding programming. But systems programming has been resistant to explicit recognition of professional specialties.

In the coming decade, we can expect to see specialization of two kinds:

- internal specialization as the technical content in the core of software grows deeper and
- external specialization with an increased range of applications that require both substantive application knowledge and substantive computing knowledge.

Internal specialties are already starting to be recognizable for communications, reliability, real-time programming, scientific computing, and graphics, among others. Because these specialties rely critically on mastery of a substantial body of computer science, they may be most appropriately organized as postbaccalaureate education.

External specialization is becoming common, but the required dual expertise is usually acquired informally (and often incompletely). Computational specializations in various disciplines can be supported via joint programs involving both computer science and the application department; this is being done at some universities.

Simply put, software engineering will require explicit recognition of specialties. Educational opportunities should be provided to support them. However, this should not be done at the cost of a solid foundation in computer science and, in the case of external specialization, in the application discipline.

Improve the coupling between science and commercial practice. Good science is often based on problems underlying the problems of production. This should be as true for computer science as for any other discipline. Good science depends on strong interactions between researchers and practitioners. However, cultural differences, lack of access to large, complex systems, and the sheer difficulty of understanding those systems have interfered with the communication that supports these interactions.

Similarly, the adoption of results from the research community has been impeded by poor understanding of how to turn a research result into a useful element of a production environment. Some companies and universities are already developing cooperative programs to bridge this gap, but the logistics are often daunting.

Simply put, an engineering basis for software will evolve faster if constructive interaction between research and production communities can be nurtured. ❖

Acknowledgments

This article benefited from comments by Allen Newell, Norm Gibbs, Frank Friedman, Tom Lane, and the other authors of articles in this special issue. Most important, Eldon Shaw fostered my appreciation for engineering. Without his support, this work would not have been possible, so I dedicate this article to his memory.

This work was supported by the US Defense Dept. and a grant from Mobay Corp.

References

1. J.K. Finch, *Engineering and Western Civilization*, McGraw-Hill, New York, 1951.
2. E.W. Dijkstra, "On the Cruelty of Really Teaching Computing Science," *Comm. ACM*, Dec. 1989, pp. 1,398-1,404.
3. D.L. Parnas, "Education for Computing Professionals," *Computer*, Jan. 1990, pp. 17-22.
4. H. Straub, *A History of Civil Engineering: An Outline from Ancient to Modern Times*, MIT Press, Cambridge, Mass., 1964.
5. F.J. van Antwerpen, "The Origins of Chemical Engineering," in *History of Chemical Engineering*, W.F. Furter, ed., American Chemical Society, Washington, D.C., 1980, pp. 1-14.
6. Computer Science and Technology Board, National Research Council, *Keeping the US Computer Industry Competitive*, National Academy Press, Washington, D.C., 1990.
7. National Academy of Engineering, *Engineering and the Advancement of Human Welfare: 10 Outstanding Achievements 1964-1989*, National Academy Press, Washington, D.C., 1989.
8. E. Levin, "Grand Challenges to Computational Science," *Comm. ACM*, Dec. 1989, pp. 1,456-1,457.
9. *Software Engineering: Report on a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 1968*, P. Naur and B. Randell, eds., Scientific Affairs Div., NATO, Brussels, 1969.
10. B.W. Boehm, "Software Engineering," *IEEE Trans. Computers*, Dec. 1976, pp. 1,226-1,241.
11. S.T. Redwine et al., "DoD-Related Software Technology Requirements, Practices, and Prospects for the Future," Tech. Report P-1788, Inst. Defense Analyses, Alexandria, Va., 1984.
12. E.D. Hirsch, Jr., *Cultural Literacy: What Every American Needs to Know*, Houghton Mifflin, Boston, 1989.
13. M. Shaw, D. Giuse, and R. Reddy, "What a Software Engineer Needs to Know I: Vocabulary," tech. report CMU/SEI-89-TR-30, Carnegie Mellon Univ., Pittsburgh, Aug. 1989.
14. F.P. Brooks, Jr., "No Silver Bullet: Essence and Accidents of Software Engineering," *Information Processing 86*, pp. 1,069-1,076.
15. W.A. Martin, "Sorting," *ACM Computing Surveys*, Dec. 1971, pp. 147-174.
16. W.J. Cody, Jr., and W.M. Waite, *Software Manual for the Elementary Functions*, Prentice-Hall, Englewood Cliffs, N.J., 1980.

Mary Shaw is a professor of computer science at Carnegie Mellon University, where she has been on the faculty since 1971. From 1984 to 1987, she was chief scientist at the Software Engineering Institute, with which she still has a joint appointment. Her primary research interests are programming systems and software engineering, particularly abstraction techniques and language tools for developing and evaluating software.

Shaw received a BA in mathematics from Rice University and a PhD in computer science from Carnegie Mellon University. She is a member of the IEEE Computer Society, ACM, New York Academy of Sciences, and Sigma Xi. She also serves on the National Research Council's Computer Science and Technology Board, IEEE Technical Committee on Software Engineering, and IFIP Working Group 2.4 (System Implementation Languages).

Address questions about this article to the author at Computer Science Dept., Carnegie Mellon University, Pittsburgh, PA 15213-3890; Internet shaw@cs.cmu.edu.