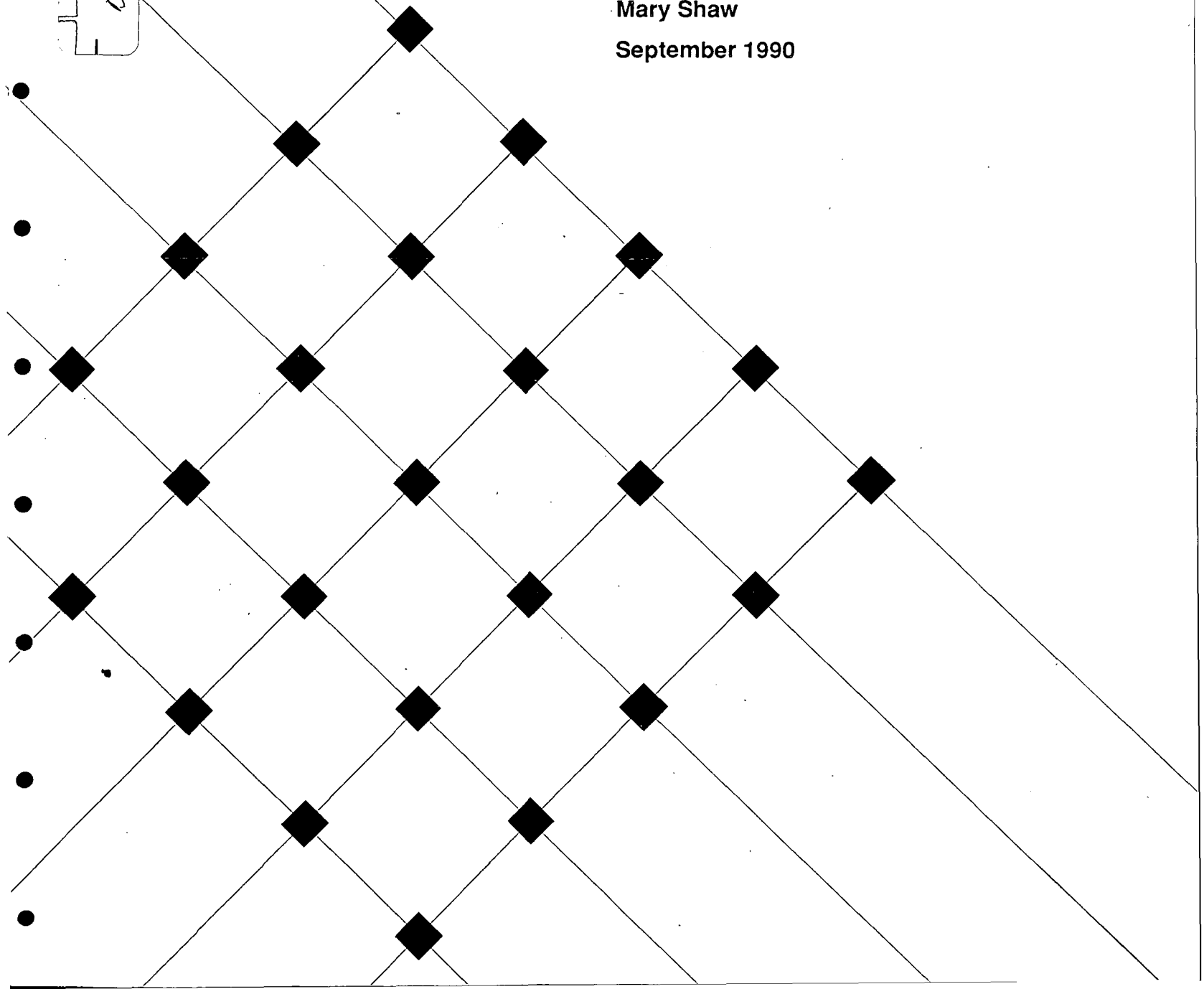Carnegie-Mellon University
Software Engineering Institute

Prospects for
an Engineering Discipline
of Software

Mary Shaw

September 1990

Prospects for
an Engineering Discipline
of Software

Mary Shaw

Software Architecture Design Principles

This technical report was prepared for the
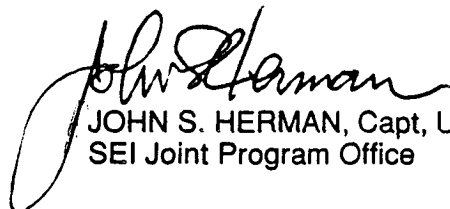
SEI Joint Program Office
ESD/AVS
Hanscom AFB, MA 01731

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

JOHN S. HERMAN, Capt, USAF
SEI Joint Program Office

This report appears as an invited paper in the November 1990 issue of *IEEE Software*. It also appears as Carnegie Mellon University School of Computer Science Technical Report No. CMU-CS-90-165.

Table of Contents

Table of Figures

Prospects for an Engineering Discipline of Software

Abstract: Software engineering is not yet a true engineering discipline, but it has the potential to become one. Older engineering fields offer glimpses of the character software engineering might have. From these hints and an assessment of the current state of software practice, we can project some characteristics software engineering will have and suggest some steps toward an engineering discipline of software .

The term *software engineering* was coined in 1968 as a statement of aspiration—a sort of rallying cry. That year NATO convened a workshop by that name to assess the state and prospects of software production [NATO 69]. Capturing the imagination of software developers, the phrase 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.

This paper begins by examining the usual practice of engineering and the way it has evolved in other disciplines. This discussion provides a historical context for assessing the current practice of software production and setting out an agenda for attaining an engineering practice.

1. What is Engineering?

Software engineering is a label applied to a set of current practices for software development. Using the word *engineering* to describe this activity takes considerable liberty with the common use of that term. In contrast, 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* have been written for well over a hundred years. Here is a sampling of typical definitions:

> In 1828 Thomas Tredgold wrote, "Engineering is the art of directing the great sources of power in nature for the use and convenience of man; being that practical application of the most important principles of natural philosophy which has, in a considerable degree, realized the anticipations of Bacon, and changed the aspect and state of affairs in the whole world" [Armytage 61, p.123].

> "Engineering is 'the art or science of making practical applications of the knowledge of pure sciences.' In other words, although engineers are not scientists, they study the sciences and use them to solve problems of practical interest, most typically by the process that we call creative design" [Florman 76, p.x].

> "Engineering is often defined as the use of scientific knowledge and principles for practical purposes ..., usually by designing useful products" [Parnas 90, p.17].

> Arthur Wellington, the late 19th century scholar of railroad economics, defined an engineer as "a man who can do that well with one dollar which any bungler can do with two after a fashion" [Finch 51, p.102].

Although these and other definitions differ in detail, they share a common set of 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.

...building things... — Engineering emphasizes the solutions, which are usually tangible artifacts.

...in the service of mankind — Engineering not only serves the immediate customer, but 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 to create sophisticated systems that work—unspectacularly, perhaps, but reliably. The history of software 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 software 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.*

1.1. 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 in order to make routine design simpler. Handbooks and manuals are often the carriers of this organized information [Marks 87, Perry 84]. 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. Further, library components require more care in design, implementation, and documentation than similar components that are simply embedded in systems.

Practitioners recognize the need for mechanisms to share experience with good designs. This cry from the wilderness appeared on a Software Engineering news groups:

> "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 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 handbook is the standard design handbook for chemical engineering; it is about 4 inches thick x 8-1/2" x 11", printed in tiny type on tissue paper [Perry 84].

1.2. A 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. This model is depicted in Figure 1. 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.

Figure 1: Evolution of an Engineering Discipline

Science

Production

Professional Engineering

Commercial

Craft

- Virtuosos and talented amateurs
- Intuition and brute force
- Haphazard progress
- Casual transmission
- Extravagant use of available materials
- Manufacture for use rather than sale

- Skilled craftsmen
- Established procedure
- Pragmatic refinement
- Training in mechanics
- Economic concern for cost and supply of materials
- Manufacture for sale

- Educated professionals
- Analysis and theory
- Progress relies on science
- Educated professional class
- Enabling new applications through analysis
- Market segmentation by product variety

Figure 1: Evolution of an Engineering Discipline

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 particular kind. Intuition and brute force are the primary movers in design and construction. Progress is haphazard, particularly before the advent of good communication; hence 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 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 is successful. 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 software 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 to apply the theory to analysis of problems and synthesis of solutions. In the 18th and early 19th centuries,

> "what was happening was a gradual drawing together of the common interests in basic physical understandings of natural science and engineering. On the one hand, the reduction of many empirical engineering techniques to a more scientific basis was essential to further engineering progress. On the other, this liaison was helpful and stimulating to further advances in natural science. An important and mutually stimulating tie-up between natural and engineering science, a development which had been discouraged for centuries by the long dominant influence of early Greek thought, was at long last consummated" [Finch 51, p.6].

The emergence of an engineering discipline allows technological development to 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 [Dijkstra 89, Parnas 90]. 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.

We turn now to two examples that serve to make this model concrete, then assess the current state of software practice and its prospects for following this evolutionary path.

2. Examples from Traditional Engineering

The evolution of engineering disciplines is demonstrated by civil and chemical engineering. The comparison of the two is also illuminating, because they have very different basic organizations.

2.1. 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. 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, such as buildings, bridges, dams, tunnels, canals, highways, railroads, public water supplies, and sanitation. As a rule, civil engineering efforts involve well-defined task groups who use appropriate tools and technologies to execute well-laid plans.

2.1.1. Emergence of Civil Engineering

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 Middle Ages nor the Ancient World showed any signs of the deliberate quantitative application of mathematics to determination of dimensions and shapes that characterizes modern civil engineering. But even without formal understanding, pragmatic rules for recurring elements were documented. 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, Brunelleschi, and others to explain structures and why they worked. Over a period of about two hundred 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 such as force, in particular 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) which we now teach to freshmen, nor was it possible to deal with strengths of materials.

About 1700 Varignon and Newton developed the *theory of statics* to explain the composition of forces and Coulomb and 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 (though 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 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 be of direct utility at the scale of a bridge design.

2.1.2. Structure of Civil Engineering

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 can be regarded as 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 which could not be solved by calculation" [Straub 64 pp.236-241].

The transition from craft to commercial practice can be dated to the Romans' extensive transportation system of the 1st 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 the significant events on our model.

1700: statics
1700: strength of materials
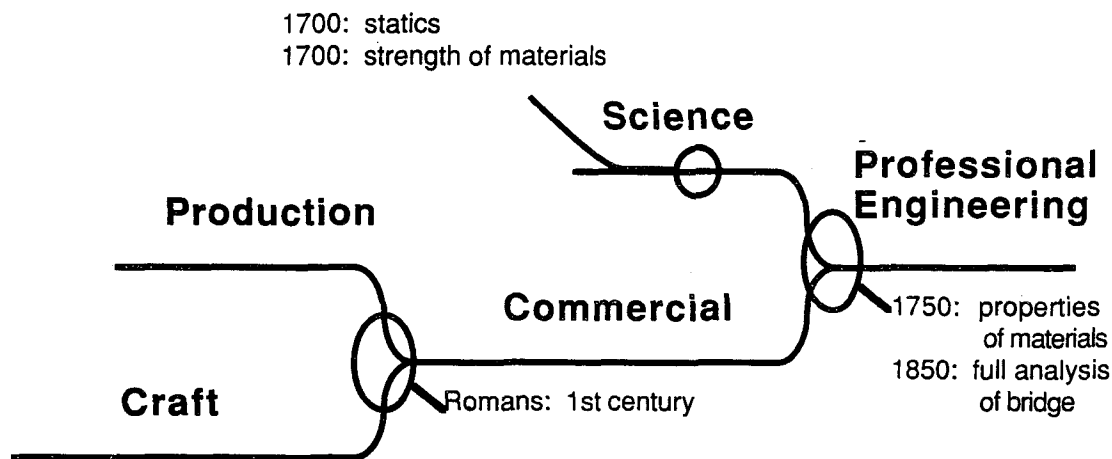
Figure 2: Evolution of Civil Engineering

2.2. Chemical Engineering: A Basis in Practice

The second example deals with a very different kind of engineering, chemical 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, etc. Though 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 heavily on mechanical engineering as well as chemistry.

2.2.1. Emergence of Chemical Engineering

Until the late 18th century, chemical production was largely a cottage industry. The first chemical produced at industrial scale was *alkali*, 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 procedures involved. He identified a collection of functional operations that took place within 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 United States by MIT 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 actually coined in 1915 by Arthur D. Little.

2.2.2. Structure of Chemical Engineering

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 small number of 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.

This is a very different kind of structure from that of civil engineering. It is a pragmatic, empirical structure, rather than a theoretical one.

> "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 which in their proper sequence and coordination constitute a chemical process as conducted on the industrial scale. These operations, as grinding, extracting, roasting, crystallizing, distilling, air-drying, separating, and so on, 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 is the province of chemical engineering. It is this selective emphasis on the unit operations themselves in their quantitative aspects that differentiates the chemical engineer from industrial chemistry, which is concerned primarily with general processes and products" [AIChE Committee on Education, 1922, quoted in van Antwerpen pp.111-12].

The transition from craft to commercial practice can be dated to the introduction of the LeBlanc process for alkali in 1789. The science emerged with 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 the significant events on our model.
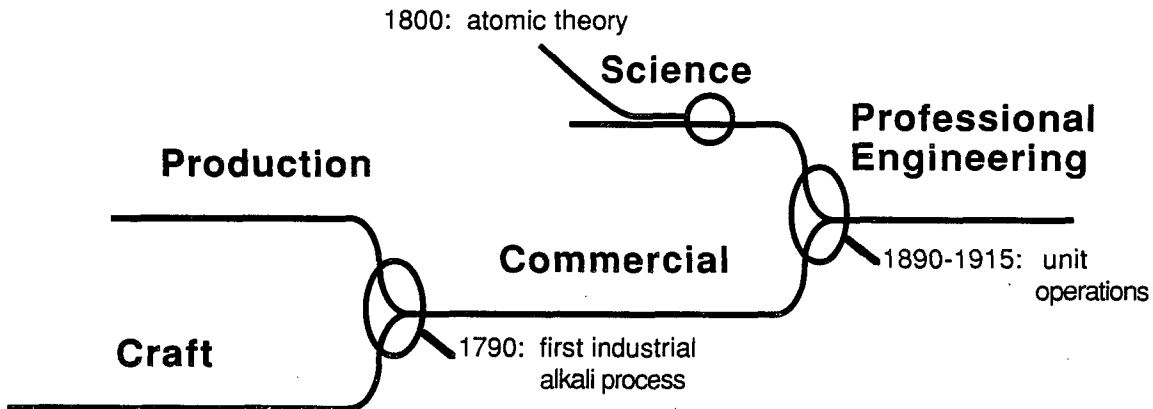
Figure 3: Evolution of Chemical Engineering

3. Current State of Software Technology

We turn now to software. We begin by establishing that the problem is appropriately viewed as an engineering problem: *creating cost-effective solutions to practical problems ... building things in the service of mankind.* We then address the question of whether software developers do or ever can do this by applying scientific knowledge. In the process we position *software engineering* in the evolutionary model described earlier.

3.1. Information Processing as an Economic Force

The US computer business, including computers, peripherals, packaged software, and communications, was about $150B in 1989 and is projected to be over $230B by 1992. The packaged software component is projected to grow from $23.7B to $37.5B in this period [DAG 89]. Services, including systems integration and in-house software development, are not included in these figures.

Worldwide, software sales amounted to about $65B in 1989. This does not include the value of in-house software development, which is a much larger activity. World figures are hard to estimate, but the cost of in-house software in the U.S. alone may be in the range $150B-$200B [CSTB 90]. 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 attempted: software development backlogs so large as to discourage new requests, gigabytes of unprocessed raw data from satellites and space probes, and so on. Despite very real (and quite substantial) successes, the litany of mismatches of cost, schedule, and expectations is a familiar one.

3.2. Growing Role of Software in Critical Applications

The (US) National Academy of Engineering recently selected the ten greatest engineering achievements of the last 25 years. Of the ten, 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 [NAE 89].

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 quite 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" [Levin 89]—but the supercomputing community is increasingly recognizing software development 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 distinction between the computational component of a system and the application it serves is often very soft; the development of effective software now often requires substantial application expertise.

3.3. Maturity of Software Development Techniques

Our software 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 of 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. Figure 4 summarizes these shifts, both of which describe the focus of attention of the software research community.

Prior to about 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 quite 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 Knuth showed the utility of thinking of 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 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 emphasis on algorithms and data structures, but the programs in question were still simple programs that execute once and then terminate.

The shift that took place in the mid-1970s from programming-in-the-small to programming-in-the-large can be viewed 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 the management of the programming process. In addition, the data of complex systems often outlives the programs and may be more valuable, so we 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: the sequence of system states is often much more important that 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 only matured 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 providing management visibility into production processes. This directly supports the commercial practice of software development.

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 data bases, 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

Figure 4: Significant Shifts in Research Attention

Practical software development proceeded to large complex systems much faster than the research community. 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 (e.g., electrical) engineering. Modern software development methodologies can be viewed as management procedures intended to guide large numbers of developers through similar disciplines.

The term *software engineering* was introduced in 1968 [NATO69]. In 1976 Boehm proposed the definition, "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" [Boehm76]. This definition is consistent with older definitions of engineering, though 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.

3.4. 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 packaging of these results for operational use is lacking.

3.4.1. 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 as well as correctness. Concurrently, the programming implications of these abstractions were explored; abstract data type research dealt with such issues as:

Specifications	abstract models, algebraic axioms
Software structure	bundling representation with algorithms
Language issues	modules, scope, user-defined types
Information hiding	protecting integrity of information-not in specification
Integrity constraints	invariants of data structures
Rules for composition	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 twenty 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 such as structured programming, Smalltalk, and unix [Redwine 84]. But the whole field of computing is only about 40 years old, and many theories are emerging in the research pipeline.

3.4.2. Interaction Between Science and Engineering

The development of good models within the software domain follows the pattern of Figure 5. We begin by solving problems any way we can manage. 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 to improve practice, and experience from that practice can sharpen the theories. Further, the improvement in practice enables us to think about harder problems—which we first solve ad hoc, then find heuristics, eventually develop new models and theories, and so on. 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.

Ad hoc solutions	→	Folklore
Systematic folklore	→	Codification, analysis
Codification	→	Models, theories
Models, theories	→	Improved practice
Experience from practice	→	New theories, principles
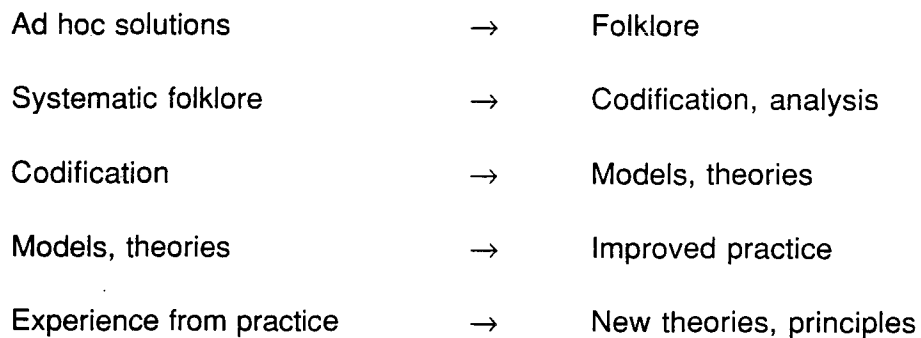
Figure 5: Codification Cycle for Science and Engineering

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 the semantics of these statements and of 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, then, does current software practice lie on the path to engineering? As Figure 6 suggests, it is still in some cases craft and in some cases commercial practice. A science is beginning to contribute results, and for isolated examples it is possible to argue that professional engineering is taking place. That is not, however, the common case.
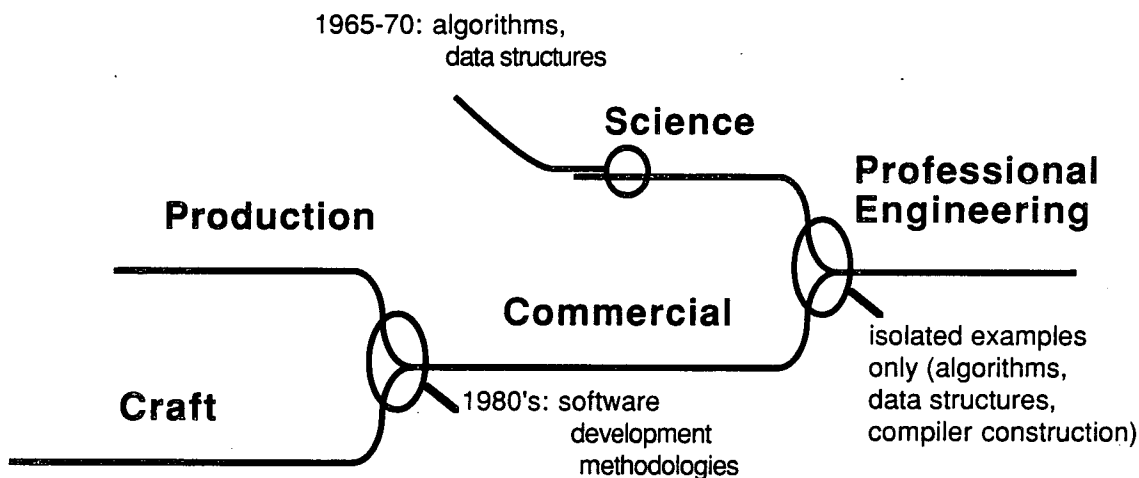
Figure 6: Evolution of Software Engineering

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. Though 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 to stabilize commercial practice and

- invest in long term efforts to develop and make available basic scientific contributions.

4. Toward an Engineering Discipline of Software

4.1. 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 demonstrate this across a wide range of problem domains, including medical diagnosis, physics, chess, financial analysis, architecture, scientific research, policy decision making, and others [Reddy 88, pp. 13-14; Simon 89, pp.1].

An expert in a field must know around 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 ten years for a world-class expert to achieve that level of proficiency [Simon 89, pp.2-4].

Thus, fluency in a domain requires content and context as well as skills. In the case of natural language fluency, Hirsch argues 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"; intellectual and social skills are supposed to develop naturally without regard to the specific content [Hirsch 88]. However, says Hirsch, 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 provides a list of some five thousand 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. He or she must 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.

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 that useful subsets are learned first, followed by progressively more sophisticated subsets. We could also develop standard reference materials as carriers of the content.

4.2. Recognize Different Ways to Obtain Information

Given that a large body of knowledge is important to a working professional, we must ask how software engineers should acquire the knowledge, either as students or as working professionals. Generally speaking, there are three ways to obtain 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:

	Infrastructure Cost	Initial Learning Cost	Cost of Use In Practice
Memory	low	high	low
Reference	high	low	medium
Derivation	medium-high	medium	high

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 individual 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; since each new application requires the model to be applied anew, the cost in use may be quite high [SGR 89].

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

Software engineering requires investment in the infrastructure cost—that is, in creating the materials required to organize information, especially reference material for practitioners.

4.3. 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 software 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.

Brooks argues that one of the great hopes for software engineering is the cultivation of great designers [Brooks 86]. 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' terms) instead of always building from scratch.

It is unreasonable to expect a software designer or developer to take advantage of scientific theories or prior experience if the necessary information is not readily available. Scientific results need to be recast in operational form; the important information from prior experience must be extracted from particular examples. The content should include design elements, components, interfaces, interchange representations, and algorithms. A conceptual structure must be developed so that 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 [CSTB 89].

A few examples of well-organized reference materials already exist. For example, the summary flowchart of Martin's sorting survey [Martin 71] captured in one page the information a designer needed to choose among the then-current sorting techniques. Cody & Waite's manual for implementing elementary mathematical functions [CW 80] gives for each 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.

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.

4.4. 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 within 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. Since these

specialties rely critically on mastery of a substantial body of computer science, they may be most appropriately organized as post-baccalaureate 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 [NSF 89].

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

4.5. Improve 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.; it 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.

An engineering basis for software will evolve faster if constructive interaction between research and production communities can be nurtured.

Acknowledgements

References

[Armytage 61]   W.H.G. Armytage. *A Social History of Engineering*. Faber and Faber 1961.

[Boehm 76]   Barry Boehm. Software Engineering. *IEEE TR on Computers*, C-25, 12 (December 1976), pp. 1226-1241.

[Brooks 86]   Frederick P. Brooks, Jr. No Silver Bullet: Essence and Accidents of Software Engineering. *Information Processing 86*, pp. 1069-1076. [Reprinted in *IEEE Computer*, 20, 4 (April 1987), pp. 10-19.]

[CSTB 89]   Computer Science and Technology Board, National Research Council. *Scaling Up: A Research Agenda for Software Engineering*. National Academy Press 1989.

[CSTB 90]   Computer Science and Technology Board, National Research Council. *Keeping the US Computer Industry Competitive*. National Academy Press, 1990.

[CW 80]   William James Cody Jr. and William McCastline Waite. *Software Manual for the Elementary Functions*. Prentice-Hall 1980.

[DAG 89]   Data Analysis Group. Computer Industry Forecasts, Fourth Quarter 1989.

[Dijkstra 89]   Edsger W. Dijkstra. On the Cruelty of Really Teaching Computing Science. *Comm ACM*, 32,12 (Dec 1989), pp.1398-1404.

[Finch 51]   James Kip Finch. *Engineering and Western Civilization*. McGraw-Hill, 1951.

[Florman 76]   Samuel C. Florman. *The Existential Pleasures of Engineering*. St. Martin's Press, New York, 1976.

[Furter 80]   William F. Furter (ed). *History of Chemical Engineering*. American Chemical Society 1980.

[Hirsch 88]   E. D. Hirsch, Jr. *Cultural Literacy: What Every American Needs to Know*. Vintage Books 1988.

[Levin 89]   Eugene Levin. Grand Challenges to Computational Science. *Comm ACM*, 32, 12 (December 1989), pp. 1456-1457.

[Marks 87]   Lionel S. Marks. *Marks' Standard Handbook for Mechanical Engineers*. McGraw-Hill 1987.

[Martin 71]   William A. Martin. Sorting. *ACM Computing Surveys*, 3,4 (Dec 1971), pp.147-174.

[NATO 69]   Peter Naur and Brian Randell (eds). *Software Engineering: report on a conference sponsored by the NATO Science Committee, Garmisch Germany 1968*. NATO 1969.

[NAE 89]   National Academy of Engineering. *Engineering and the Advancement of Human Welfare: 10 Outstanding Achievements 1964-1989*. National Academy Press 1989.

[NSF 89]   Report on the National Science Foundation Disciplinary Workshops on Undergraduate Education. National Science Foundation, 1989.

[Parnas 90]   David Lorge Parnas. Education for Computing Professionals. *IEEE Computer* 23,1 (Jan 1990) pp. 17-22.

[Perry 84]   Robert H. Perry. *Perry's Chemical Engineers' Handbook*. McGraw-Hill 1984.

[Reddy 88]   Raj Reddy. Foundations and Grand Challenges of Artificial Intelligence. *AI Magazine*, vol 9, no 4 (Winter 1988), pp. 9-21 (1988 presidential address, American Association for Artificial Intelligence).

[Redwine 84]   Samuel T. Redwine et al. DoD Related Software Technology Requirements, Practices, and Prospects for the Future. IDA Paper P-1788, Institute for Defense Analysis, 1984.

[SGR 89]   Mary Shaw, Dario Giuse, Raj Reddy. What A Software Engineer Needs to Know I: Vocabulary. CMU-CS and CMU/SEI-89-TR-30 Tech reports (DTIC No. ADA219064), August 1989.

[Simon 89]   Herbert A. Simon. Human Experts and Knowledge-Based Systems. Talk given at IFIP WG 10.1 Workshop on Concepts and Characteristics of Knowledge-Based Systems, Mt Fuji Japan, November 9-12, 1987.

[Straub 64]   Hans Straub. *A History of Civil Engineering: An Outline from Ancient to Modern Times*. MIT Press, 1964.

[van Antwerpen 80] F. J. van Antwerpen. The Origins of Chemical Engineering. In [Furter 80], pp.1-14.

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED		1b. RESTRICTIVE MARKINGS NONE			
2a. SECURITY CLASSIFICATION AUTHORITY N/A		3. DISTRIBUTION/AVAILABILITY OF REPORT APPROVED FOR PUBLIC RELEASE DISTRIBUTION UNLIMITED			
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A					
4. PERFORMING ORGANIZATION REPORT NUMBER(S) CMU/SEI-90-TR-20		5. MONITORING ORGANIZATION REPORT NUMBER(S) ESD-TR-90-221			
6a. NAME OF PERFORMING ORGANIZATION SOFTWARE ENGINEERING INST.	6b. OFFICE SYMBOL (If applicable) SEI	7a. NAME OF MONITORING ORGANIZATION SEI JOINT PROGRAM OFFICE			
6c. ADDRESS (City, State and ZIP Code) CARNEGIE MELLON UNIVERSITY PITTSBURGH, PA 15213		7b. ADDRESS (City, State and ZIP Code) ESD/AVS HANSCOM AIR FORCE BASE HANSCOM, MA 01731			
8a. NAME OF FUNDING/SPONSORING ORGANIZATION SEI JOINT PROGRAM OFFICE	8b. OFFICE SYMBOL (If applicable) ESD/AVS	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F1962890C0003			
8c. ADDRESS (City, State and ZIP Code) CARNEGIE MELLON UNIVERSITY PITTSBURGH, PA 15213		10. SOURCE OF FUNDING NOS.			
		PROGRAM ELEMENT NO. 63752F	PROJECT NO. N/A	TASK NO. N/A	WORK UNIT NO. N/A
11. TITLE (Include Security Classification) PROSPECTS FOR AN ENGINEERING DISCIPLINE OF SOFTWARE					

12. PERSONAL AUTHOR(S) Mary Shaw				
13a. TYPE OF REPORT FINAL	13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (Yr., Mo., Day) September 1990	15. PAGE COUNT 24	
16. SUPPLEMENTARY NOTATION				

17.	COSATI CODES		18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number) software engineering history of engineering history of software engineering
FIELD	GROUP	SUB. GR.	

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Software engineering is not yet a true engineering discipline, but it has the potential to become one. Older engineering fields offer glimpses of the character software engineering might have. From these hints and an assessment of the current state of software practice, we can project some characteristics software engineering will have and suggest some steps toward an engineering discipline of software.

The term software engineering was coined in 1968 as a statement of aspiration, a sort of rallying cry. That year NATO convened a workshop by that name to assess the state and prospects of software production (NATO 69). Capturing the imagination of software developers, the phrase 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.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS ☒	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED, UNLIMITED DISTRIBUTION	
22a. NAME OF RESPONSIBLE INDIVIDUAL JOHN S. HERMAN, Capt, USAF	22b. TELEPHONE NUMBER (Include Area Code) 412 268-7630	22c. OFFICE SYMBOL SEI JPO

DD FORM 1473, 83 APR   EDITION OF 1 JAN 73 IS OBSOLETE.