FINDING A HISTORY FOR SOFTWARE ENGINEERING

Michael S. Mahoney
Princeton University

Annals of the History of Computing 26,1(2004), 8-19 (Preprint version, with illustrations)

Introduction

Dating from the first international conference on the topic in October 1968, software engineering just turned thirty-five. It has all the hallmarks of an established discipline: societies (or sub-societies), journals, textbooks and curricula, even research institutes. It would seem ready to have a history. Yet, a closer look at the field raises the question of just what the subject of the history would be. It is not hard to find definitions. A leading practitioner spoke of it in 1989 as "the disciplined application of engineering, scientific, and mathematical principles and methods to the economical production of quality software."(1) But it is also not hard to find doubts about whether its current practice meets those criteria and, indeed, whether it is an engineering discipline at all. A colleague of the practitioner just quoted declared at about the same time (1990): "Software engineering is not yet a true engineering discipline, but it has the potential to become one."(2) From the outset, software engineering conferences have routinely begun with a keynote address that asks "Are we there yet?" and proposes yet another specification of just where "where" might be.(3)

(1) W. Humphrey, "The Software Engineering Process: Definition and Scope", *Representing and Enacting the Software Process: Proc. 4th Int'l Software Process Workshop*, ACM Press, 1989, p. 82.

(2) M. Shaw, "Prospects for an Engineering Discipline of Software", *IEEE Software* vol. 7, no. 6, Nov. 1990, p. 15.

(3) Indeed, this article stems from just such an address, delivered to ACM SIGSOFT's 9th Foundations of Software Engineering Conference (FSEC 9) in 1998.

Since the field has been a moving target for its own practitioners, historians may understandably have trouble knowing just where to aim their attention. What is a history of software engineering about? Is it about the engineering of software? If so, by what criteria or model of engineering? Is it engineering as applied science? If so, what science is being applied and what is its history? Is it about engineering as project management? Is it engineering by analogy to one of the established fields of engineering? If so, which fields, and what are the terms of the analogy? Of what history would the history of software engineering be a part, that is, in what larger historical context does it most appropriately fit? Is it part of the history of engineering? The history of business and management? The history of the professions and of professionalization? The history of the disciplines and their formation? If several or all of these are appropriate, then what aspects of the history of software engineering fit where?

Alternatively, to put the question in another light, is the historical subject more accurately described as "software engineering" with the inverted commas as an essential part of the title? As noted above, what seems clear from the literature of the field from its very inception, reinforced by addresses, panels, articles and letters to the editor that continue to appear regularly, is that its practitioners do not agree on what software engineering is, although most of them freely confess that, whatever it is, it is not (yet) an engineering discipline. Historians have no stake in the outcome of that question. They can just as readily write a history of "software engineering" viewed as the continuing effort of various groups of people engaged in the production of software to establish their practice as an engineering discipline. The question of interest to historians would then be how "software engineers" have gone about that task of self-definition. In large part, addressing that question comes down to observing and analyzing the answers practitioners have offered to the questions just posed above. That is, rather than positing a consensus among practitioners concerning the nature of software engineering, historians can follow the efforts to achieve a consensus. Taking that approach would place the subject firmly in the comparative context of the history of professionalization and the formation of new disciplines.(4)

(4) See, for example, A.D. Abbott, *The System of Professions: An Essay on the Division of Expert Labor*, University of Chicago Press, 1988.

For this reason, it may help to think of historians and practitioners as engaged in a common pursuit. They are both looking for a history for software engineering, though not for the same purpose and not from the same standpoint. Hence, the title above is meant to be ambiguous. In one sense, it describes historians trying to determine just what the subject of their inquiry might be and then deciding how to write its history.(5) In another sense, it describes efforts by practitioners to define or to characterize software engineering. Quite often those efforts amount to finding a history, that is, to seeking to identify the current development of software engineering with the historical development of one of the established engineering disciplines or indeed of engineering itself.(6) Using history in this way has its real dangers; the initial conditions cannot by their nature be exactly repeated. Nonetheless, it is at the very least essential both that one have the right history and that one have the history right, not least because what passes for history often amounts to common wisdom, folklore or local myth.(7) Here historians may offer some assistance to the software engineers. While we may not be able to tell them whether they have the right history, we can in many cases tell them what history they have chosen and whether they have got it right.

(5) For a recent discussions of the question, see *The History of Software Engineering* (Report of the Dagstuhl Seminar No. 9635, ed. W. Aspray, R. Keil-Slawik, and D. Parnas, Dagstuhl, 1996; available online at http://www.dagstuhl.de/9635/Report/), and J.E. Tomayko, "Software as Engineering" with commentaries by A. Endres and B.E. Seely, *History of Computing: Software Issues,* U. Hashagen, R. Keil-Slawik, and A. Norberg, eds., Springer Verlag, 2002.

(6) Mary Shaw of Carnegie Mellon University and the Software Engineering Institute took this approach explicitly in "Prospects for an Engineering Discipline of Software," *IEEE Software,* vol. 7, no. 6, Nov. 1990, pp. 15-24, where she proposed a historical model of the professionalization of engineering based primarily on the development of chemical engineering. Her diagram of the process reappeared in enhanced from in W.W. Gibbs, "Software's Chronic Crisis",

Ultimately, every definition of software engineering presupposes some historical model. For example, take the oft-quoted passage from the introduction to the proceedings of the first Software Engineering Conference, convened by the NATO Science Committee in 1968:

> The phrase 'software engineering' was deliberately chosen as being provocative, in implying the need for software manufacture to be based on the types of theoretical foundations and practical disciplines that are traditional in the established branches of engineering.(8)

The phrase indeed turned out to be provocative, if only because it left all the crucial terms undefined. What does it mean to "manufacture" software? Is that a goal or a current practice? What, precisely, are the "theoretical foundations and practical disciplines" that underpin the "established branches of engineering"? What roles did they play in the formation of the engineering disciplines? Is the story the same in each case? The reference to "traditional" makes the answer to that question a matter of history. It is a question of how the fields of engineering took their present form. It is a search for historical precedents, or what we have come to refer to as "roots".

Or rather, it is a matter of what I call "agendas" The agenda of a field consists of what its practitioners agree ought to be done, a consensus concerning the problems of the field, their order of importance or priority, the means of solving them (the tools of the trade), and perhaps most importantly, what constitutes a solution. Becoming a recognized practitioner means learning the agenda and then helping to carry it out. Knowing what questions to ask is the mark of a full-fledged practitioner, as is the capacity to distinguish between trivial and profound problems; "profound" means moving the agenda forward. One acquires standing in the field by solving the problems with high priority, and especially by doing so in a way that extends or reshapes the agenda, or by posing profound problems. The standing of the field may be measured by its capacity to set its own agenda. New disciplines emerge by acquiring that autonomy.(9) Conflicts within a discipline often come down to disagreements over the agenda: what are the really important problems?

A new science means a new agenda, and tracing the emergence of a new science means showing how a group of practitioners coalesces around a common agenda different from other agendas in which they have been engaged. Each of those other agendas reflects a history, and so the members of the group bring to their new agenda a variety of histories. Some, or perhaps even much, of the disagreement among the participants in the first two NATO conferences, especially the second, rested on the different histories they brought to the gatherings. None of them was a software engineer, for the field did not exist. Rather, people came from quite varied professional and disciplinary traditions, each of which had its own history, in many cases a mythic history.(10) Three of these in particular have directed the practitioners' search for historical guidance: applied science, mechanical engineering, and industrial engineering and management. What follows is a brief look at how the histories have been invoked and how they have been understood.

Scientific American, vol. 271, no 3, Sep. 1994, pp. 86-95; at p. 92.

(7) For example, at the first NATO conference (see below), Ronald Graham of Bell Labs remarked that "we build systems like the Wright brothers built airplanes -- build the whole thing, push it off the cliff, let it crash, and start over again" (*Software Engineering: Concepts and Techniques. Proceedings of the NATO Conferences,* P. Naur, B. Randell, and J.N. Buxton, eds., Petrocelli, 1976, p. 7). Historians of technology know that the Wright Brothers' successful flight was in fact the culmination of a carefully planned, theoretically and empirically informed, program of research and development.. In particular, they had a relatively clear idea of what problems they had to solve and of how they might go about solving them. Whether or not their approach might have served as a useful example for fledgling software engineers, it does not seem *prima facie* to constitute a negative example.

(8) *Software Engineering: Report on a Conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968*, P. Naur and B. Randell, eds., Scientific Affairs Division, NATO, 1969, p. 13. The report was republished, together with the report on the second conference in Rome the following year, in P. Naur, B. Randell, and J.N. Buxton, eds., *Software Engineering: Concepts and Techniques. Proceedings of the NATO Conferences*, Petrocelli, 1976. Randell has made both reports available for download in pdf format at http://homepages.cs.ncl.ac.uk/brian.randell/NATO/. The site includes photographs of the participants and some of the sessions at Garmisch.

(9) On the formation of the agendas of theoretical computer science, see M.S. Mahoney, "Software as Science - Science as Software", *History of Computing: Software Issues*, U. Hashagen, R. Keil-Slawik, and A. Norberg, eds. (Berlin: Springer Verlag, 2002), pp. 25-48

(10) "Myth" here should be taken in the sense of a story told by a community to account for why it does things they way it does. The story may be more or less factually accurate, but its function does not depend on it.

Models of engineering: Historical Precedents

Applied Science

To some, in particular many of the European participants, engineering was essentially applied science, and the science in question here was mathematics.(11) What was needed, then, was firm grounding in theoretical, i.e. mathematical, computer science. The historical model seemed clear. Indeed, it had been set forth explicitly almost ten years earlier, albeit in another context, by John McCarthy, the creator of LISP and co-founder of artificial intelligence. Looking "Towards a mathematical theory of computation" at IFIP 1962, he had reached for a familiar touchstone:

> In a mathematical science, it is possible to deduce from the basic assumptions, the important properties of the entities treated by the science. Thus, from Newton's law of gravitation and his laws of motion, one can deduce that the planetary orbits obey Kepler's laws.(12)

As McCarthy and his audience well knew, one can also deduce the laws of the motion of terrestrial bodies and all the mechanics that derives from them. He extended the model at the conclusion of his 1963 article, "A Basis for a Mathematical Theory of Computation", by reference to later successes in mathematical physics:

> It is reasonable to hope that the relationship between computation and mathematical logic will be as fruitful in the next century as that between analysis and physics in the last. The development of this relationship demands a concern for both applications and mathematical elegance.(13)

(11) Edsgar W. Dijksta was the foremost proponent of this view.

(12) "Towards a mathematical science of computation", *Proc. IFIP Congress, Munich, 1962* (IFIP 62), North-Holland, 1963, p. 21.

(13) "A basis for a mathematical theory of computation", *Proc. Western Joint Computer Conf.* , vol.19, May, 1961, Spartan Books, pp. 225-238; reprinted, with corrections and an added tenth section, in *Computer Programming and Formal Systems*, P. Braffort and D. Hirschberg, eds., North-Holland, 1963, pp. 33-70; at p. 69.

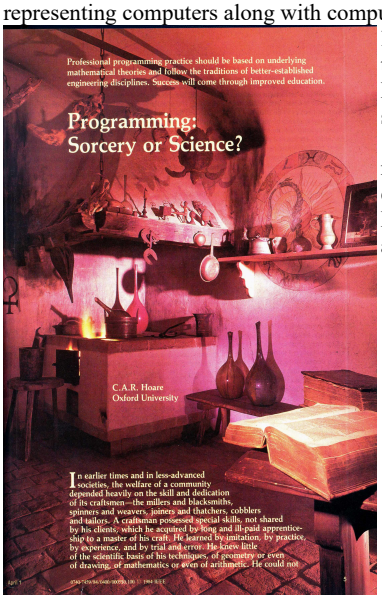
The applications of mathematics to physics had produced more than new theories. The mathematical theories of thermodynamics and electricity and magnetism had informed the development of heat engines, of dynamos and motors, of telegraphy and radio. Those theories formed the scientific basis of engineering in those fields.

The twentieth century had a new science, McCarthy believed, and it too had implications beyond just theory. "Computation is sure to become one of the most important of the sciences," he began,

> This is because it is the science of how machines can be made to carry out intellectual processes. We know that any intellectual process that can be carried out mechanically can be performed by a general purpose digital computer. Moreover, the limitations on what we have been able to make computers do so far clearly come far more from our weakness as programmers than from the intrinsic limitations of the machines. We hope that these limitations can be greatly reduced by developing a mathematical science of computation.(14)

(14) McCarthy, "Basis", 33.

The ultimate object of computer science was working programs, argued McCarthy, and a suitable theory of computation would provide: first, a universal programming language along the lines of Algol but with richer data descriptions; second, a theory of the equivalence of computational processes, by which equivalence-preserving transformations would allow a choice of among various forms of an algorithm, adapted to particular circumstances; third, a form of symbolic representation of algorithms that could accommodate significant changes in behavior by simple changes in the symbolic expressions; fourth, a formal way of representing computers along with computation; and finally a quantitative theory of computation along the lines of Claude Shannon's measure of information.(15) Note that as this list progresses, it sounds more and more like engineering, and McCarthy's agenda (and its history) continued to echo in the software-engineering literature. In arguing in 1984 that "[p]rofessional programming practice should be based on underlying mathematical theories and follow the traditions of better-established engineering disciplines," C.A.R. Hoare highlighted in a sidebar McCarthy's comparison of physics and mathematical logic quoted above. (16)



(15) *Ibid.*, 34. McCarthy argued that none of the three current (1961) directions of research into the mathematics of computing held much promise of such a science. Numerical analysis was too narrowly focused. The theory of computability set a framework into which any mathematics of computation would have to fit, but it focused on what was unsolvable rather than seeking positive results, and its level of description was too general to capture actual algorithms. Finally, the theory of finite automata, though it operated at the right level of generality, exploded in complexity with the size of current computers. As he explained in another article, "... [T]he fact of finiteness is used to show that the automaton will eventually repeat a state. However, anyone who waits for an IBM 7090 to repeat a state, solely because it is a finite automaton, is in for a very long wait." ("Towards a mathematical science of computation", *Proc. IFIP Congress 62,* North-Holland, 1963, p. 22).

(16) C.A.R. Hoare, "Programming: Sorcery or Science?", *IEEE Software,* vol. 1, no. 2, Mar. 1984, pp. 5-16; at p. 10. Perhaps only coincidentally the article included a photograph of the room in which Kepler died (p.14).

Over the decade of the '60s theoretical computer science achieved standing as a discipline recognized by both the mathematical and the computing communities, and it could point to both applications and mathematical elegance.(17) Yet, it took the form more of a family of loosely related research agendas than of a coherent general theory validated by empirical results. No one mathematical model had proved adequate to the diversity of computing, and the different models were not related in any effective way. What mathematics one used depended on what questions one was asking, and for some questions no mathematics could account in theory for what computing was accomplishing in practice. It was a far cry from Newton's mechanics, much less the mathematical physics of the nineteenth century, and it remains so.

(17) For an overview, see M.S. Mahoney, "The Structures of Computation", *The First Computers - Histories and Architectures*, R. Rojas and U. Hashagen, eds., MIT Press, 2000.

In a discussion on the last day of the second NATO Conference on Software Engineering held in Rome in October 1969, Christopher Strachey, Director of the Programming Research Group at Oxford University and a leading figure in the development of formal semantics, lamented that "one of the difficulties about computing science at the moment is that it can't demonstrate any of the things that it has in mind; it can't demonstrate to the software engineering people on a sufficiently large scale that what it is doing is of interest or importance to them."(18) About a decade later, a committee in the United States reviewing the state of art in theoretical computer science echoed his diagnosis, noting the still limited application of theory to practice. (19) By the mid-'70s, moreover, it seemed clear to some that, even if existing theory had practical application, it would not quite meet the needs of software engineering. In a 1976 article, Barry Boehm of TRW proposed that software engineering be defined 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." Boehm identified the salient terms as "design", "software maintenance", and "scientific knowledge" and took stock of what was known in each area.(20)

(18) Naur, *et al., Software Engineering,* p. 147.

(19) *What Can Be Automated? (COSERS)*, Bruce W. Arden, ed., MIT Press, 1980, p. 139. The committee consisted of Richard M. Karp (Chair; Berkeley), Zohar Manna (Stanford), Albert R. Meyer (MIT), John C. Reynolds (Syracuse), Robert W. Ritchie (Washington), Jeffrey D. Ullman (Stanford), and Shmuel Winograd (IBM Research).

(20) B. Boehm, "Software Engineering", *IEEE Transactions on Computers*, vol. C-25, no. 12, Dec. 1976, pp. 1226-41 (repr. in *Milestones of Software Engineering*, P. W. Oman and Ted G. Lewis, eds.,IEEE Computer Society Press, 1990, pp. 54-69), at p. 1226 (p. 54). An early leader in the field of software metrics, Boehm later developed COCOMO, a system for estimating the cost of software projects and wrote the leading text in the subject, *Software Engineering Economics.*

The first two terms he addressed by reference to what by then was becoming the standard model of the "software life cycle", a sequence that took a project from the requirements to an operating program by way of

(21) B. Boehm, "Software and its Impact: A Quantitative Assessment", *Datamation*, vol. 19, 1973, pp. 48-59.

specification, design, coding, and testing. What he saw as current practice reinforced the concerns of the crisis. In particular, requirements analysis was informal at best, and software design was "still almost completely a manual process ... [with] relatively little effort devoted to design validation and risk analysis". Yet, as he had shown in a now classic article in 1973, the bulk of the errors in software were made during the design phase.(21)

Most significantly for present purposes, he also concluded that little of current computer science was relevant to the problems of software engineering:

> Those scientific principles available to support software engineering address problems in an area we shall call *Area 1: detailed design and coding* of *systems software* by *experts* in a relatively *economics-independent* context. Unfortunately, the most pressing software development problems are in an area we shall call *Area 2: requirements analysis design, text, and maintenance* of *applications software* by *technicians* in an *economics-driven* context.(22)

(22) Boehm, "Software Engineering", p. 67. Boehm's footnote to "technicians" is worth repeating here. "For example, a recent survey of 14 installations in one large organization produced the following profile of its 'average coder': 2 years college-level education, 2 years software experience, familiarity with 2 programming languages and 2 applications, and generally introverted, sloppy, inflexible, 'in over his head', and undermanaged. Given the continuing increase in demand for software personnel, one should not assume that this typical profile will improve much. This has strong implications for effective software engineering technology which, like effective software, must be well-matched to the people who use it."

However successful the experimental systems and theoretical advances produced in the laboratory, especially the academic laboratory, they did not take account of the challenges and constraints of "industrial-strength" software in a competitive market. As Fritz Bauer, the organizer of the first NATO conference, had put it at IFIP '71, those problems made software engineering "the part of computer science that is too difficult for the computer scientists".(23)

(23) F. L. Bauer, "Software Engineering", *Information Processing 71*, North-Holland Publishing Co., 1972, pp. 530-538; at p. 530. Repr. in *Advanced Course in Software Engineering,* F.L. Bauer, ed., Springer-Verlag, 1973, pp. 522-545; the reprint did not include Bauer's playful parody of a computer scientist's design of a three-prong hay fork.

Mechanical Engineering

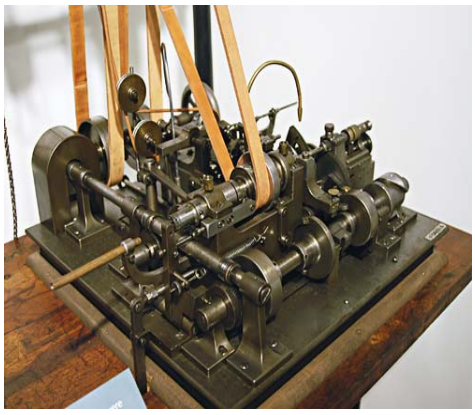
If not applied science, then what? Others at the NATO conference had proposed models of engineering that emphasized analogies of practice rather than theory. Perhaps the most famous of these was M.D. McIlroy's evocation of the machine-building origins of mechanical engineering and the system of mass production by interchangeable parts that grew out of them. Seeing software sitting somewhere on the other side of the Industrial Revolution, he proposed to vault it into the modern era.

> We undoubtedly produce software by backward techniques. We undoubtedly get the short end of the stick in confrontations with hardware people because they are the industrialists and we are the crofters. Software production today appears in the scale of industrialization somewhere below the more backward construction industries. I think its proper place is considerably higher, and would like to investigate the prospects for mass-production techniques in software.

He left no doubt of whose lead to follow. He continued:

> In the phrase 'mass production techniques', my emphasis is on 'techniques' and not on mass production plain. Of course mass production, in the sense of limitless replication of prototype, is trivial for software. But certain ideas from industrial technique I claim are relevant. The idea of subassemblies carries over directly and is well exploited. The idea of interchangeable parts corresponds roughly to our term 'modularity', and is fitfully respected. The idea of machine tools has an analogue in assembly programs and compilers. Yet this fragile analogy is belied when we seek for analogues of other tangible symbols of mass production. There do not exist manufacturers of standard parts, much less catalogues of standard parts. One may not order parts to individual specifications or size, ruggedness, speed, capacity, precision or character set.(24)

(24) M.D. McIlroy, "Mass Produced Software Components," in Naur and Randell, pp. 138-150.; at p. 138-39. At the time, McIlroy was one of the representatives of Bell Labs to the Multics project at MIT, where he worked on the semantics of PL/I. He subsequently oversaw the development of Unix, to which he contributed the notion of "pipes", which allows the chaining of programs, each taking as its input the output of its predecessor.



As studies of the American machine-tool industry during the 19th and early 20th century have shown, McIlroy could hardly have chosen a more potent model (he has a longstanding interest in the history of technology). Between roughly 1820 and 1880, developments in machine-tool technology had increased routine shop precision from .01" to .0001". More importantly, in a process characterized by the economist Nathan Rosenberg as "convergence", machine-tool manufacturers learned how to translate new techniques developed for specific customers into generic tools of their own.(25) So, for example, the need to machine bits for drilling small holes in percussion locks led to the development of the vertical turret lathe, which in turn lent itself to the production of screws and small precision parts, which in turn led to the automatic turret lathe. Indeed, it was

(25) Nathan Rosenberg, "Technological Change in the Machine Tool Industry, 1840-1910", *Journal of Economic History* vol. 23, 1963, pp. 414-443; repr. in Rosenberg, *Perspectives on Technology*, Cambridge Univ. Press., 1976), Chap.1.

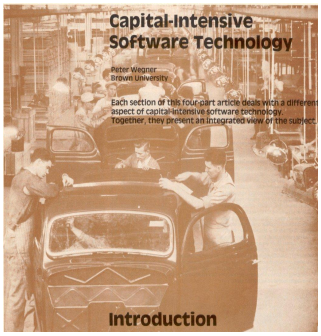
(26) In conversation at Bell Labs, Fall, 1989.

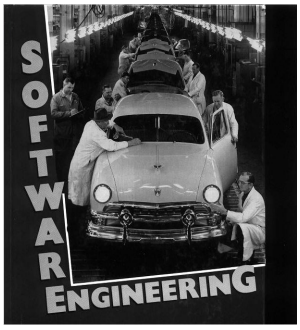
precisely the automatic screw-cutting machine that McIlroy had in mind.(26)

As McIlroy noted, he was giving sharper, historically grounded form to an idea that had already begun to take shape. In an Advanced Course in Software Engineering that took place at Munich's Technical University in 1972, Jack B. Dennis of MIT's Project MAC lectured on "Modularity", pointing as example to standardized floor tiles (19" square "modules") which fill any size or shape of floor area "with just a bit of trimming at the boundary", while allowing great variety through different colors and textures of modules.

> In modular software, clearly the "standardized units or dimensions" should be standards such that software modules meeting the standards may be conveniently fitted together (without "trimming") to realize large software systems. The reference to "variety of use" should mean that the range of module types available should be sufficient for the construction of a usefully large class of programs.(27)

(27) Jack B. Dennis, "Modularity", in Bauer, *Advanced Course on Software Engineering* , Chap. 3.A; at p. 128.





Especially as expressed by McIlroy, the idea has had a long career in software engineering. During the '70s it directed attention beyond the development of libraries of subroutines to the notion of "reusable" programs across systems, and in the '80s it underlay the growing emphasis on object-oriented programming as the means of achieving such reusability on a broad scale. It is essentially what Brad Cox was looking for around 1990 as the basis for software's "industrial revolution".(28) More generally, the analogy with machine-building and the metaphorical language of machine-based production became a continuing theme of software engineering, often illustrated by pictures of automobile assembly lines, as in the case of Peter Wegner's four-part article in *IEEE Software* in 1984 on "Capital-Intensive Software Technology".(29) The cover of that issue bore a photograph of a Ford assembly line in the '30s, and a picture of the same line in the early '50s adorned Gregory W. Jones's *Software Engineering* (Wiley, 1990).(30)

(28) Brad J. Cox, "Planning the Software Industrial Revolution", *IEEE Software* vol. 7, no. 6, Nov. 1990.

(29) *IEEE Software* vol. 1, no. 3, July 1984, pp. 7-45.

(30) Both Wegner and Jones have told me that their editors, not they, chose the pictures in question. Thus, the analogy was widely shared in the larger community.

Industrial Engineering

As the move from machine tools to the assembly line suggests, McIlroy's model of mechanical engineering was closely akin to F.L. Bauer's proposal at IFIP 71 that "software design and production [be viewed] as an industrial engineering field".

> For the time being, we have to work under the existing conditions, and the work has to be done with programmers who are not likely to be re-educated. It is therefore all the more important to use organizational and managerial tools that are appropriate to the task.(31)

(31) F. L. Bauer, "Software Engineering", p. 532.

On that model the problems of large software projects came down to the "division of the task into manageable parts", its "division into distinct stages of development", "computerized surveillance", and "management". Each of these tasks posed significant problems, and Bauer had specific suggestions to make only with regard to the third: computerized surveillance consisted of:

- Automatic updating and quality control of documentation,
- Selective dissemination of information to all project staff,
- Surveillance of deadline plans,
- Collection of data for simulation studies,
- Collection of data for quality control,
- Automatic production of manuals and maintenance material.

"It is clear," he noted, "that a house well equipped with programs and an underlying philosophy for doing these things, can be regarded as a modern software plant."(32)

(32) *Ibid*., 533.

Bauer's idea was not new. In a "Position Paper for [the] Panel Discussion [on] the Economics of Program Production" at IFIP 68, also presented in substance at the NATO conference, R.W. Bemer of GE had suggested that what software managers lacked was a proper environment:

> It appears that we have few specific environments (factory facilities) for the economical production of programs. I contend that the production costs are affected far more adversely by the absence of such an environment than by the absence of any tools in the environment (e.g. writing a program in PL/1 is using a tool.)

A factory supplies power, work space, shipping and receiving, labor distribution, and financial controls, etc. Thus a software factory should be a programming environment residing upon and controlled by a computer.

(33) R.W. Bemer, "Position Paper for Panel Discussion [on] the Economics of Program Production", *Information Processing 68*, North-

Program construction, checkout and usage should be done entirely within this environment. Ideally it should be impossible to produce programs exterior to this environment.(33)

Holland Publishing Company, 1969, vol. II, p. 1626.

Bemer's proposal was aimed at the problem of workers' near-total control over production, which the computer itself held promise of overcoming. "Economical products of high quality," he continued,

> are not possible (in most instances) when one instructs the programmer in good practice and merely hopes that he will make his invisible product according to those rules and standards. This just does not happen under human supervision.
>
> A factory, however, has more than human supervision. It has measures and controls for productivity and quality. Financial records are kept for costing and scheduling. Thus management is able to estimate from previous data: not so with programming management in general. Computer supervision and aid are vital, with the accent upon human engineering factors so that working in the environment is both attractive and effective for the programmer.

In reading these words, it is hard not to hear an echo of Frederick W. Taylor and his methods of "scientific management", which informed management thinking, both here and in Europe in ways that are only now becoming clear.(34) Indeed, the basic principles of Taylor's system sound much like the agenda that early software engineer- managers were laying out for themselves. The primary obligation of management according to Taylor was to determine the scientific basis of the task to be accomplished. That came down to four main duties:

(34) In the now classic *Taylorism at Watertown Arsenal: Scientific Management in Action, 1908-1915* (Cambridge, MA: Harvard U.P., 1960; repr. as *Scientific Management in Action: Tarlorism at Watertown Arsenal, 1908-1915*, Princeton, Princeton U.P., 1985), H.G.J. Aitken listed Taylor's "solutions of enduring significance" (p. 29): (1) the planned routing and scheduling of work in progress, leading to the assembly line and continuous flow production; (2) systematic inspection procedures between operations; (3) printed job and instruction cards; (4) refined cost-accounting techniques; (5) systematization of store procedures, purchasing, and inventory control; (6) and "functional foremanship", which was the only element not to gain general acceptance. Taylor got little credit from historians for these things, yet "these inconspicuous innovations have probably exercised a more far-reaching influence on industrial practice than has the conspicuous innovation of stop-watch time study". Taylor and Taylorism have attracted renewed attention from historians in recent decades; see in particular D. Nelson, ed., *A Mental Revolution: Scientific Management Since Taylor* (Ohio State Univ. Press, 1992) and S.P. Waring, *Taylorism Transformed: Scientific Management since 1945* (Univ. North Carolina Press, 1991). R. Kanigel's *The One Best Way: Frederick Winslow Taylor and the Enigma of Efficiency* (Viking Press, 1997) is a full and informative biography.

> *First.* They develop a science for each element of a man's work, which replaces the old rule-of-thumb method.(35)
>
> *Second*. They scientifically select and then train, teach, and develop the workman, whereas in the past he chose his own work and trained himself as best he could.
>
> *Third*. They heartily cooperate with the men so as to insure all of the work [is] being done in accordance with the principle of the science which has been developed.
>
> *Fourth*. There is an almost equal division of the work and the responsibility between the management and the workmen. The management take over all work for which they are better fitted than the workmen, while in the past almost all of the work and the greater part of the responsibility were thrown upon the men.(36)

(35) That science constituted the famous "one best way" on which Taylor's system rested.

(36) F.W. Taylor, *The Principles of Scientific Management,* 1911; repr. Norton, 1967, pp. 36-37.

In the emphasis on supervision and support of the programmer, Bemer's factory sounds like Taylor's machine shop, with management seeking to impose the "one best way" over a worker still in control of the shop floor.

A decade later, William W. Agresti of the University of Michigan-Dearborn made the tie to Taylor explicit. In a follow-up to his talk, "Applying Industrial Engineering to the Software Development Process, presented at the IEEE Computer Society's 23rd International Conference in the fall of 1981, he published a short piece on "Software Engineering as Industrial Engineering" in *Software Engineering Notes*:

> While working on this project, I returned for inspiration to the "old masters" of industrial engineering: Frederick Taylor, Henry Gantt, and Frank and Lillian Gilbreth. The accounts of their work in the early 1900s provide remarkable reading as a glimpse of society at that time. I was also impressed that much of what they were saying then about I.E. (or "scientific management" as it was known then) could be said today about software engineering.(37)

(37) W.W. Agresti, "Software Engineering as Industrial Engineering", *Software Engineering Notes* vol. 6, no. 5, 1981, 11-12; at 11. I thank Michael Cusumano for drawing my attention to this article. Agresti later moved to Computer Sciences Corporation and then to MITRE Corporation.

As examples, Agresti offered a page of excerpts from the works of the masters as they might apply to such matter as "Finding Program 'Bugs'", "Introducing Structured Programming Methods", and "Software Tools". Concerning the "Analysis of Algorithms," he went to the heart of Taylor's system: "Now, among the various methods used..., there is always one method which is quicker and better than any of the rest. And this one best method can only be discovered through a scientific study and analysis of all the methods in use...."

Whether implicitly or explicitly, Taylorism continued to inform the industrial approach to software engineering. Leon J. Osterweil's keynote address at the 9th International Conference on Software Engineering in 1987 offers a striking example.(38) Even more recently, Watts S. Humphrey, principal designer of the widely used (and DoD-sanctioned) Capability Maturity Model and Personal Software Process, provides more

(38) L.J. Osterweil', "Software Processes are Software Too", *Proceedings of the 9th Int'l Conf. Software Engineering* (ICSE 9), IEEE Computer Society Press, 1987, pp. 2-13. At ICSE 19,

explicit testimony to Taylor's presence in thinking about software management. In an article on the current status and trends in the Personal Software Process, Humphrey references Peter Drucker in asserting that "Even though manual and intellectual tasks are significantly different, we can measure, analyze, and optimize both and thus apply Taylor's principles equally well." He then explains his point in language quite close to Bemer's:

> The principal difference between manual and intellectual work is that the knowledge worker is essentially autonomous. That is, in addition to deciding how to do tasks, he or she must also decide what tasks to do and the order in which to do them. The manual worker commonly follows a relatively fixed task order, essentially prescribed by the production line. So studying and improving the performance of intellectual work must not only address the most efficient way to do each task but also consider how to select and order these tasks. The is essentially the role of a defined process and a detailed plan. The process defines the tasks, task order, and task measures, while the plan sizes the tasks and defines the task schedule for the job being done.(39)

Osterweil's paper was recognized as the most influential paper of ICSE 9.

(39) W.S. Humphrey, "The Personal Software Process: Status and Trends", *IEEE Software* vol. 17, no. 6, Nov./Dec. 2000, p.72.

If, as Osterweil maintains, "software processes are software, too", then some of those processes are Taylor-inspired software for managing workers.

Yet, in the late 1960s, when the notions of software engineering and the software factory were first proposed, practitioners could fulfill none of Taylor's requirements. To what extent computer science could replace rule of thumb in the production of software was precisely the point at issue at the NATO conferences (and, as noted above, it remains a question). Even the optimists agreed that progress had been slow. Unable, then, to fulfil the first duty, programming managers were hardly in a position to carry out the third. Everyone bemoaned the lack of standards for the quality of software. As far as the fourth was concerned, few ventured to say who was best suited to do what in large-scale programming project. Frederick P. Brooks offered an example in his now classic *The Mythical Man-Month*:

> It is a very humbling experience to make a multimillion-dollar mistake, but it is also very memorable. I vividly recall the night we decided how to organize the actual writing of external specifications for OS/360. The manager of architecture, the manager of control program implementation, and I were threshing out the plan, schedule, and division of responsibilities.
>
> The architecture manager had 10 good men. He asserted that they could write the specifications and do it right. It would take ten months, three more than the schedule allowed.
>
> The control program manager had 150 men. He asserted that they could prepare the specifications, with the architecture team coordinating; it would be well-done and practical, and he could do it on schedule. Furthermore, if the architecture team did it, his 150 men would sit twiddling their thumbs for ten months.
>
> To this the architecture manager responded that if I gave the control program team the responsibility, the result would *not* in fact be on time, but would also be three months late, and of much lower quality. I did, and it was. He was right on both counts. Moreover, the lack of conceptual integrity made the system far more costly to build and change, and I would estimate that it added a year to debugging time.(40)

(40) F.P. Brooks, Jr., *The Mythical Man-Month: Essays on Software Engineering*, Addison-Wesley, 1975, pp. 47-48.

Only with that experience behind him was Brooks in a position to think about what precisely was wrong with his decision.

As for Taylor's second principle, by 1969 the failure of management to establish standards for the selection and training of programmers was legend. As Dick H. Brandon, the head of one of the more successful software houses, pointed out, the industry at large scarcely agreed on the most general specifications of the programmer's task. Managers seeking to hire people without programming experience (as the pressing need for programmers required) had only one quite dubious aptitude test at their disposal, and no one knew for certain how to train those people once they were hired.(41) So one was back where one started: to implement the model required solving the problems to which the model was supposed to provide the solution, quite apart from how effective that solution had in fact turned out to be.

(41) D. H. Brandon, "The Economics of Computer Programming", *On the Management of Computer Programming*, G.F. Weinwurm, ed., Auerbach, 1970, Chap.1. Brandon evidently viewed management through Taylorist eyes, but he was clear-sighted enough to see that computer programming failed to meet the prerequisites for scientific management. For an analysis of why testing was so unreliable, see R.N. Reinstedt, "Results of a Programmer Performance Prediction Study", *IEEE Trans. Engineering Management*, Dec. 1967, 183-87, and G.M. Weinberg, *The Psychology of Computer Programming* (NY, 1971), Chap.9.

Much of the articulation of software engineering during the 1970s and '80s aimed at laying the groundwork for effective management: structured analysis and design as a means of hierarchical division of projects and allocation of tasks, structured programming as a means both of quality control and of disciplining programmers, methods of cost accounting and estimation, methods of verification and validation, techniques of quality assurance. Except for structured programming, which could be enforced by increasingly effective diagnostic compilers, most of these methods were paper exercises for which the computer served largely clerical purposes. One could very well program "outside the environment".

A year after Bemer laid out his scheme, GE left the computer business, but the concept of the software factory survived. Indeed, the Systems Development Corporation trademarked the term, and proposed to set

(42) M. Cusumano, *Japan's Software Factories*, pp. 147-8; the quotation is from the description of

up what Michael Cusumano describes as a "conveyor and control system that brought work and materials (documents, code modules) through different phases, with workers using standardized tools and methods to build finished software products", or, in the words of its designers,

> In the Factory, the Development Data Base serves as the assembly line --carrying the evolving system through the production phases in which factory tools and techniques are used to steadily add more and more detail to the system framework.(42)

the SDC software factory by H. Bratman and T. Court, "Elements of the Software Factory: Standards, Procedures, and Tools," in *Software Engineering Techniques*, Infotech International, 1977, p. 137. For a historical overview of the concept, see Cusumano, "Shifting Economies: From Craft Production to Flexible Systems and Software Factories," *Research Policy*, vol. 21, 1992, pp. 453-80.

The evocation of the assembly line linked the software factory to a model of industrial production different from Taylor's --how different is a complex historical and technical question-- namely Ford's system of mass production through automation. Ford did not have to concern himself about how to constrain workers to do things in "the one best way". His machines of production embodied that way of doing things; the worker had little to do with it. The same was true of the assembly line itself.

> In the chassis assembling are forty-five separate operations or stations. The first men fasten four mud-guard brackets to the chassis frame; the motor arrives on the tenth operation and so on in detail. Some men do only one or two small operations, others do more. The man who places a part does not fasten it --the part may not be fully in place until after several operations later. The man who puts in a bolt does not put on the nut; the man who puts on the nut does not tighten it. (43)

(43) H. Ford, *My Life and Work*, Doubleday, 1922, pp. 82-83.

As parts moved through the production process, they took on the shape of the Model T because that shape was, so to speak, built into the machines of production. Ford's methods worked because he was producing a machine, the essential components of which could be completely and precisely specified and hence could be produced by machines, themselves in turn fully specifiable. Indeed, Ford designed the Model T to be produced by machines, and therefore the available means of production were part of the target specifications. Underpinning that achievement was the development of the machine-tool industry alluded to above.

The assembly line has held continuing allure for software engineers, who generally find it ironic that "programmers have done a good job of automating everyone's work but their own." Indeed, that is referred to as the "software paradox".(44) That one would find it paradoxical lies in the nature of the computer combined with a particular notion of engineering. "We know," said John McCarthy, "that any intellectual process that can be carried out mechanically can be performed by a general purpose digital computer." By "mechanically", he meant according to clear, unambiguous procedures. Engineering, especially science-based engineering, aims at providing solutions of just that sort to its problems. Hence, one ought to be able to do for software what one has done for other engineering problems, namely to transfer solutions to the computer for execution. The grail of "automatic programming", as pursued in particular by Robert Balzer of ISI, with the support of DARPA, throughout the '70s and '80s was a software development system which could take the specification of a problem and transform it automatically into a working system as solution, in essence cutting out the programmer altogether. Much of the CASE software developed during this period purported to achieve portions of that goal; on close inspection, little of it in fact lived up to its claims.

(44) See B. Blum, "Understanding the Software Paradox", *ACM SIGSOFT Software Engineering Notes*, vol. 10, no. 1, 1985, pp. 43-47, who attributes the notion to L.G. Stucki.

In *Japan's Software Factories* and related articles, Michael Cusumano presents evidence suggesting that the effective management of software production lies at a level in between craft production and mass production, namely at the level of flexible design and production systems. The "factories" Cusumano examined during the mid-to-late '80s involved the following measures:

- identification of a target market and of a range of "semi-custom" products for it
- a longterm commitment to production for that market
- intensive review of currently available tools and practices
- intensive and continuing training of personnel and imposition of a programming discipline on them
- commitment of productive effort to the building of tools
- emphasis on reusability, encouraging designers and programmers to devote project effort to non-project goals
- emphasis on design and testing phases of development
- intensive quality control through inspection and testing.

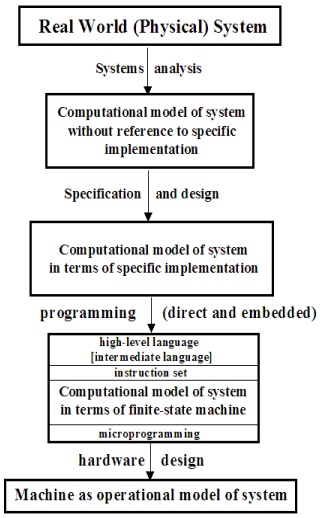
Basically, the list comes down to a corporate investment in training and maintaining a skilled work force with cumulative experience in the areas for which they are building systems. There are examples of such environments in the U.S., Bell Labs perhaps foremost among them. Unix is a leading model for the notion of software tools and of a "programmer's workbench".

To what extent such environments are "factories" in the sense in which they were originally conceived is debatable. Indeed, Cusumano notes the resistance of the programmers themselves to the term, because it connotes a devaluation of their skills. The phrase "workbench", which also appears in the Japanese context, lies closer to the shop than to the assembly line. Although these environments suggest that the production of software is far from inherently unmanageable, they also make clear that productivity depends on a highly skilled, and commensurately expensive workforce.

What is being automated?

Those "factories" are also not likely to hold a solution for the problems of software production that motivated the drive for software engineering, but, then, neither is automatic programming. Consider another version of

development phases of the software life cycle (Fig. 1). We are on firmest theoretical ground at the bottom of the diagram. That is where computer science has achieved its most profound results, and that is where theory has most effectively translated into practical software tools. But the problems of air traffic control systems, of national weather systems, of airline booking systems all lie at the top of the diagram, where a real-world system must be transformed into a computational model. That is where software engineering is not about software, indeed where it may not be about engineering at all.



Software engineering began as a search for an engineering discipline on which to model the design and production of software. That the search continues after thirty-five years suggests that software may be fundamentally different from any of the artifacts or processes that have been the object of traditional branches of engineering: it is not like machines, it is not like masonry structures, it is not like chemical processes, it is not like electric circuits or semiconductors. It thereby raises the question of how much guidance one may expect from trying to emulate the patterns of development of those engineering disciplines. During general discussion concerning theory and practice held on the last day of the Rome conference in 1969, I.P. Sharp came at the issue from an entirely different angle, arguing that one ought to think in terms of "software architecture" (= design), which would be the meeting ground for theory (computer science) and practice (software engineering). "Architecture is different from engineering," he maintained and then added, "I don't believe for instance that the majority of what [Edsger] Dijkstra does is theory -- I believe that in time we will probably refer to the 'Dijkstra School of Architecture'."(45) That is no small distinction. Architecture has a different history from engineering, and we train architects differently from engineers.(46) It is striking to a historian looking for a history of software engineering that the 9th Foundations of Software Engineering Conference in 1998, which concluded with a plenary session on whether software engineering is ready to become a "profession", that is, whether its practitioners should be subject to licensing as professional engineers, was preceded by the 3rd International Workshop on Software Architecture.

(45) *Software Engineering Techniques: Report on a conference sponsored by the NATO Science Committee, Rome, Italy, 27th to 31st October 1969,* J.N. Buxton and B. Randell, eds., NATO Science Committe, [1969], p. 12.

(46) For a review of the architectural model, see J.O. Coplien, "Reevaluating the Architectural Metaphor: Toward Piecemeal Growth", *IEEE Software* vol. 16, no. 5, Sep./Oct. 1999, pp. 40-44.