1980

Software Engineering-Education

Harlan D. Mills

Software Engineering Education

HARLAN D. MILLS

Abstract—In a field as rapidly growing as software engineering, the education problem splits into two major parts—university education and industrial education. (Some of which is given at university locations, as short courses, but considered industrial education here.) Both parts draw on the same underlying disciplines and methodologies. But the people involved—both teachers and students—have different objectives and characteristics. At the university level students are young, inexperienced, and relatively homogeneous in background and abilities. At the industrial level, students are older, more experienced, and vary considerably in background and abilities.

In this paper, we discuss the underlying commonalities and the overlaid differences of university and industrial education in software engineering. The commonalities in discipline and methodologies involve the study and understanding of the Software Process, as discussed in Section II of this special issue, and of the "Tools" and "Know How" discussed in Section III. The differences are due to the characteristics and objectives of students, and show up on curricula content and structure and in course definition.

I. Software Engineering Education in Flux

A. University Education and Industrial Education

IN A FIELD as rapidly growing as software engineering, the education problem splits into two major parts—university education and industrial education. (Short courses given at university locations without degree credits are considered industrial education here.) Both parts draw on the same underlying disciplines and methodologies. But the people involved—both teachers and students—have different objectives and characteristics.

University students are young, inexperienced, and relatively homogeneous in background and abilities. Industrial students are older, more experienced, and vary considerably in background and abilities. University teachers are oriented toward a transient student population (in 2–4 years they are gone) and to their own publications. Industrial teachers are oriented to a more stable student population and to improved industrial performance of students due to their education. In brief, university students are "supposed to be learning" while industrial students are "supposed to be working."

In a field more stable than software engineering, university education plays a dominant role in shaping the principles and values of the field, while industrial education consists of refresher and updating courses in fringe and frontier areas. But university education in software engineering was not available to the majority of people who practice and manage it today. Therefore, the principles and values of software engineering are being shaped jointly by university and industrial influences.

B. A Serious Problem

The U.S. finds itself far ahead in computer hardware but also heading for a serious problem in software. In a recent object lesson, our electronics industry was strengthened significantly by the shortfall of our missile boosters compared to those of the Soviet Union 20 years ago. As a partial result of the severe discipline of power, space, and weight limitations in our boosters, our electronics was miniaturized and improved in dramatic ways. And we lead in electronics today because of this history.

In reverse, we have seen an astonishing growth in computer power and availability. And our software industry has suffered from the lack of enforced discipline thereby, even while developing the largest software systems known today. Simply put, we are used to squandering computer power. This bad habit pervades industry, government, and the very sociology and psychology of the bulk of the computer programming today. Since information processing has become an essential part of the way society manages its industries and thereby a key to industrial power, the inertia of several hundred thousand undisciplined programmers in the U.S. is real reason for future concern.

We can also be sure that this causality will work in reverse. The lack of computing scarcity provides temptations every day in every way to excuse and condone poor performance in the software sector. Indeed, the software industry has already bungled its way into a predominate share of the costs of data processing.

Unless we address this problem with exceptional measures, we are on the way to a "software gap" much more serious and persistent than the famous "missile gap" which helped fuel the very growth of our electronics industry.

C. The Problem Perpetuated

As a result of this history, the educational background and discipline of the vast majority of computer programmers is seriously low. But, as a natural human trait, most of these programmers would rather be comforted than educated. "After all, if I'm as good as the next person, I'm good enough."

Fortunately for these programmers, there are any number of industrial short courses which will comfort, rather than educate. They are "practical," "easy to understand," "the latest techniques." On attendance, programmers discover various new names for common sense, superficial ideas, and thereby conclude, with much comfort and relief, that they have been up to date all the time. But unfortunately for the country, these programmers have not only learned very little, but have been reinforced in the very attitude that they have little to learn!

To make matters worse, many of these comfortable and comforting short courses make liberal use of the term "software engineering" as a buzz word. Such a typical "education" in software engineering consists of three days of listening, no exams, but a considerable feeling of euphoria.

This accident of history poses critical problems for univer-

Manuscript received January 22, 1980; revised May 28, 1980.
The author is with IBM Corporation, 10215 Fernwood Road, Bethesda, MD 20034.

0018-9219/80/0900-1158$00.75

sities, as well. The great demand for software engineering provides many temptations for lowered academic standards. The solid mathematical bases for software analysis and design are just emerging and are not easy to package for classroom use at this stage. But since software touches so many broad issues, there is no problem in filling a semester course, or even a curriculum, with all the latest buzz words and proposals of the field.

II. What Is Software Engineering?

A. Computer Science, Computer Programming, and Software Engineering

It is fashionable to relabel all computer programming as software engineering today, but we will not do that here. Our definition of software engineering requires both software and engineering as essential components. By software we mean not only computer programs, but all other related documentation including user procedures, requirements, specifications, and software design. And by engineering, we mean a body of knowledge and discipline comparable to other engineering curricula at universities today, for example, electrical engineering or chemical engineering.

We distinguish software engineering from computer science by the different goals of engineering and science in any field—practical construction and discovery. We distinguish software engineering from computer programming by a presence or not of engineering-level discipline. Software engineering is based on computer science and computer programming, but is different from either of them.

The full discipline of software engineering is not economically viable in every situation. Writing high-level programs in large well structured application systems is such an example. Such programming may well benefit from software engineering principles, but its challenges are more administrative than technical, more in the subject matter than in the software.

However, when a software package can be written for fifty thousand dollars, but costs five million to fix a single error because of a necessary recall of a dangerous consumer product, the product may well require a serious software engineering job, rather than a simple programming job of unpredictable quality.

B. Mathematical Foundations of Software Engineering

It is characteristic of an engineering discipline to have explicit technical foundations, and software engineering is no exception. Since the content of software is essentially logical, the foundations of software engineering are primarily mathematical—not the continuum mathematics underlying physics or chemistry, of course, but finite mathematics more discrete and algebraic than analytic in character. It has been remarked[1] that "algebra is the natural tool to study things made by man, and analysis the tool to study things made by God." Software is made by man and algebra is indeed the natural mathematical tool for its study, although algebra appears in many forms and disguises in computer science topics. For example, automata theory, theories of syntax and semantics of formal languages, data structuring and abstractions, and program correctness are all algebraic in character, in spite of widely differing notations due to their historical origins.

In contrast, electrical engineering combines physical and logical design, and therefore draws on both continuum and discrete mathematics. Software engineering uses continuum mathematics only for convenient approximation, e.g., in probability or optimization theory. The difference between the logical design of electrical engineering and the logical design of software engineering is one of scale. The logical complexity of a large software system is orders of magnitude above the logical complexity of a physically realizable processor. In fact, this ability to realize and implement logical complexity of high order is the reason for software.

Note that discrete mathematics does not necessarily imply finite mathematics. The analysis of algorithms, for example, leads to deep logical questions as to whether a computational process is finite or not, even though all operations are discrete. The theory of Turing machines provides another such example [8].

C. Structure and Organization in Software Engineering

The primary difficulty in software engineering is logical complexity [4]. And the primary technique for dealing with complexity is structure. Because of the sheer volume of work to be done, software development requires two kinds of structuring, algebraic and organizational. Algebraic structuring, applied in different ways, allows mental techniques of divide and conquer, with the same underlying principles, in the various phases of specification, design, implementation, operation, and evolution of software. The result of proper structuring is intellectual control, namely the ability to maintain perspective while dealing with detail, and to zoom in and out in software analysis and design.

The principal organizational technique is work structuring—between workers and machines, and further, between workers. Software tools, in the form of language compilers, operating systems, data entry and library facilities, etc., represent techniques of structuring work between workers and machines. One major dimension of work structuring among people is along the conceptual–clerical axis, which permits effective isolation and delegation of clerical work. Other dimensions are based on subject matter in software and applications. A surgical team represents a good example of work structuring, with different roles predefined by the profession and previous education. Surgery, anesthesiology, radiology, nursing, etc., are dimensions of work structuring in a surgical team. The communication between these roles is crisp and clean—with a low bandwidth at their interface, e.g., at the "sponge and scalpel" level, not the whole bandwidth of medical knowledge. A grammar school soccer team represents a poor example of work structuring—the first kid who reaches the ball gets to kick it. But the first person reaching the patient doesn't get to operate, and hospital orderlies do not become surgeons through on-the-job training.

D. Career Structures in Software Engineering

In addition to degree-level engineering skills in software, we identify the need for various grades of technician skills, and for degree-level science and administration skills as well. Within the engineering skills, we can differentiate by subject matter and further by skill level through graduate degree levels.

Just as in any other profession such as law, medicine, etc., many skill categories and skill levels go into a well-formed soft-

[1] By Professor W. Huggins, The Johns Hopkins University.

ware engineering team. In software development, the sheer weight of precise logic dominates, and the need for precision procedures for design and control is critical. For example, in law, three judges may subdivide an opinion for a joint writing project and meet the requirements for legal precision with small variations in their individual vocabularies. But a joint software development by three programmers will not tolerate the slightest variation in vocabulary because of the literal treatment of the design text by a computer.

The software engineer is at the center of software development and computer operations in which basic algorithms and data processing may require other advanced skills for their definition, analysis, and validation. Because of this, graduate science and administrative skills are frequent partners in software development, and the software engineer needs to be at home with an interdisciplinary approach.

Within software engineering, we can identify several areas of concentration which have the depth and substance that can occupy a person through a life-long career. Those areas include such topics as compilers, operating systems, data-base systems, real-time control systems, and distributed processing systems. These specialties in software engineering usually require graduate-level education for effective team leadership and advanced technical contributions.

III. Software Engineering Practices

A. Elements of Software Engineering

The effective practice of software engineering must be based on its technical foundations just as any other engineering activity, in combining real world needs and technical possibilities into practical designs and systems. For our purposes it is convenient to classify the disciplines and procedures of software engineering into three categories.

1) Design—(after Plato, Phaedrus). "First, the taking in of scattered particulars under one Idea, so that everyone understands what is being talked about . . . Second, the separation of the Idea into parts, by dividing it at the joints, as nature directs, not breaking any limb in half as a bad carver might."

2) Development—The organization of design activities into sustained software development, including the selection and use of tools and operational procedures for work structuring among different categories of personnel.

3) Management—Requirements analysis, project definition, identifying the right personnel, and the estimation, scheduling, measurement, and control of software design and development.

B. Software Engineering Design

The availability of useful, tested, and well-documented principles of software specification and design has exploded in the past decade, in three distinct areas, namely,

1) sequential process control—characterized by structured programming and program correctness ideas of Dijkstra [7], Hoare [14], Linger, Mills, and Witt [17], and Wirth [26], [27];
2) system and data structuring—characterized by modular decomposition ideas of Dijkstra [9], Dahl [7], Ferrentino and Mills [11], [19], and Parnas [22]:
3) real-time and multidistributed processing control—characterized by concurrent processing and process synchronization ideas of Brinch Hansen [5], Dijkstra [10], Hoare [15], and Wirth [28].

The value of these design principles is in the increased discipline and repeatability they provide for the design process. Designers can understand, evaluate, and criticize each other's work in a common objective framework. In a phrase of Weinberg [25], people can better practice "egoless software design" by focusing criticisms on the design and not the author. Such design principles also provide direct criteria for more formal design inspection procedures so that designers, inspectors, and management can better prepare for, conduct, and interpret the results of periodic orderly design inspections.

C. Software Engineering Development

Even though the primary conceptual work of software engineering is embodied in design, the organization and support of design activities into sustained software development is a significant activity in itself, as discussed in [3], [20]. The selection and definition of design and programming support languages and tools, the use of library support systems to maintain the state of a design under development, the test and integration strategy, all impact the design process in major ways. So the disciplines, tools, and procedures used to sustain software development need to be scrutinized, structured, and chosen as carefully as the design principles themselves.

The principal need for development discipline is in the intellectual control and management of design abstractions and details on a large scale. Brooks [6] states that "conceptual integrity is the most important consideration in systems design." Design and programming languages are required which deal with procedure abstractions and data abstractions, with system structure, and with the harmonious cooperation of multidistributed processes. Design library support systems are needed for the convenient creation, storage, retrieval, and modification of design units, and for the overall assessment of design status and progress against objectives.

The isolation and delegation of work between conceptual and clerical activities, and between various subactivities in both categories is of critical importance to a sustained and manageable development effort. Chief programmer teams [3] embody such work structuring for small and medium size projects. In larger projects, an organization of Chief Programmer Teams and other functional units is required.

D. Software Engineering Management

The management of software engineering is primarily the management of a design process, and represents a most difficult intellectual activity. Even though the process is highly creative, it must be estimated and scheduled so that various parts of the design activity can be coordinated and integrated into a harmonious result, and so that users can plan on results as well. The intellectual control that comes from well-conceived design and development disciplines and procedures is invaluable in achieving this result. Without that intellectual control, even the best managers face hopeless odds in trying to see the work through.

In order to meet cost/schedule committments in the face of imperfect estimation techniques, a software engineering manager must practice a manage-and-design-to-cost/schedule process. That process calls for a continuous and relentless rectification of design objectives with the cost/schedule required for achieving those objectives. Occasionally, this rectification can be simplified by a brilliant new approach or technique, which increases productivity and shortens time in the development process. But usually, just because the best possible approaches and techniques known are already planned, a shortfall, or even a windfall in achievable software, requires

consultation with the user in order to make the best choices among function, performance, cost, and schedule. It is especially important to take advantage of windfalls, to counter other shortfalls; too often windfalls are unrecognized and squandered. The intellectual control of good software design not only allows better choice in a current development, but also permits subsequent improvements of function and performance in a well-designed baseline system.

In software engineering, there are two parts to an estimate—making a good estimate and making the estimate good. It is up to the software engineering manager to see that both parts are right, along with the right function and performance.

IV. Principles of Education in Software Engineering

A. Degrees in Software Engineering

A degree in software engineering should first of all be an engineering degree, dealing with engineering design and construction. It should not simply be a computer programming degree or a computer science degree. As already noted, there is much programming to be done in society, and other curricula in arts and science or business administration should be called upon to provide properly focused education for more general programming in business and science applications. The UCLA masters program in Computer Science [16] is a good model of such other curricula, which has high-technology content, yet does not pretend to be software engineering.

The usual principles of university education should apply to a curriculum in software engineering, namely that it be a preparation for a career based on topics of reasonable half life, while producing entry-level job skills, and the ability to learn later. These objectives are not incompatible because the very topics required for dealing with technically challenging software problems are generally basic topics of long life, and do indeed prepare people for more advanced education and continued learning. It is well known that mathematics and science are more easily learned when young and so, as a rule, soft topics should be deferred for postgraduate experience and continued learning. There is real danger in over using soft topics and survey courses loaded with buzz words to provide near-term job entry salability. But without adequate technical foundations people will become dead ended in mid-career, just when they are expected to solve harder problems as individuals, as members or as managers, of teams.

In the three categories of software engineering practices listed above, studies in design practices are prime candidates for early university education; development practices should be phased in later, and management practices deferred for continued postdegree learning, after considerable experience in individual and team practice in software engineering.

B. Foundations and Problem Solving

This is a difficult dilemma in university curricula in balancing the needs for solid technical foundations and to learn problem solving. Of course, this dilemma is not unique to software engineering. Limiting topics to techniques allows a more efficient education process in terms of quantity, volume, and quality of techniques that are teachable. But it is frequently difficult for students to apply such techniques in problem-solving contexts. Problem solving is a great motivator and confidence builder. But too much emphasis on problem solving cuts into the amount of technique preparation possible, and produces students able to make a good first showing in their career but who are likely to drop out early because of the lack of deeper technical abilities.

It is characteristic in software engineering that the problems to be solved by advanced practitioners require sustained efforts over months or years from many people, often in the tens or hundreds. This kind of mass problem-solving effort requires a radically different kind of precision and scope in techniques than is required for individual problem solvers. If that precision and scope is not gained in university education, it is difficult to acquire it later, no matter how well motivated or adept a person might be at individual, intuitive approaches to problem solving.

We all know of experiences in elementary mathematics courses in getting little or no credit for guessing correct answers without showing the process for finding them. There was a good reason, because guessing answers to small problems cannot be scaled up to larger problems, whereas processes need to solve smaller problems can be scaled up. That scale up problem is the principal difference between computer programming and software engineering.

C. Curriculum Topics

ACM Curriculum '78 [2] is a well-accepted prescription for an undergraduate degree in computer science/programming. But there are those who believe that Curriculum '78 does not present enough, and the right kind of mathematics. In any case, this author believes that degrees in software engineering should be considerably stronger in discrete mathematics than suggested by Curriculum '78. In particular, a curriculum in software engineering should require a good working knowledge of the first-order predicate calculus, the algebras of sets, functions and relations, and a deep enough understanding of mathematical reasoning to use it in a flexible way in large and complex problems. We are beginning to see evidence of the practical power of mathematical reasoning in mastering software complexity, for example in program verification [12], and in the development of entire software systems, such as the UCLA Unix Security Kernel [24]. With such a foundation, the curriculum can provide an understanding of algorithms [1], computer programs [17], [26], [27] data structures [13], data abstractions [18], and data bases [23] as mathematical objects.

D. Adult University Education

The rapid growth of software engineering means that there will be a considerable amount of adult education in university work (in contrast to short courses which may be given in universities on a nondegree basis.) Typically these will be advanced degrees for people with an already good foundation in mathematics or engineering science. It is to be expected that adult education will go on in parallel in arts and sciences, and in business administration schools for much the same reason because the whole industry is growing rapidly. But as noted before, we distinguish between programming and software engineering and we mean to discuss here adult university education in software engineering only.

Adult students in university curricula have advantages and disadvantages over younger students coming directly out of previous education. Their advantages are in their motivation and in the fact that they have a larger experience base in which to embed the ideas, techniques, etc., they receive in the education process. Their disadvantages are in being rusty in the learning process and possibly in having their education somewhat outmoded through the passage of time. On balance,

people who are motivated enough to return for adult education at the university level, are usually superior students and get more out of their education than their younger peers, but they should be expected to live up to the academic standards of the institution.

E. Laboratory Courses in Software Engineering

We know from other science and engineering disciplines that laboratory courses are usually more difficult to develop than lecture courses. In software, simply letting people learn by themselves in developing programs and systems as projects can lead to two weeks of experience repeated seven times rather than a fourteen-week laboratory course of cumulative experience. The problem with such open-loop student projects is that much of the time is spent on recovering from unwise decisions or poor executions made earlier, with little real learning going on.

A degree program in software engineering should contain a minimum sequence of laboratory courses, which is based on understanding and modifying existing programs and solving hardware/software integration problems before proceeding to program design and development and later into system specification and design. This laboratory sequence should proceed from 1) a highly structured environment in which carefully conceived programs (with carefully conceived problems) are presented to students for testing and modification to 2) less structured situations where students design and develop small, then large, software products from well-defined specifications, finally to 3) even less structured situations where they deal with informal requirements from which specifications and designs are to be developed. In this sequence there is an opportunity to identify problems, which all students encounter simultaneously, for which instructors can help develop approaches and solutions. A hardware/software integration problem early in the laboratory sequence seems especially important for software engineering students, because there are usually important interfaces between hardware and software in the high-performance systems dealt with by software engineering.

REFERENCES

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*. Reading, MA: Addison-Wesley, 1974.

[2] R. Austing *et al.*, Eds., "Curriculum 78: Recommendations for the undergraduate program in computer science—A report of the ACM curriculum committee on computer science," *Commun. Ass. Comput. Mach.*, vol. 22, no. 3, Mar. 1979.

[3] F. T. Baker, "Chief programmer team management of production programming," *IBM Syst. J.*, vol. II, no. 1, 1972.

[4] L. A. Belady and M. M. Lehman, "The evolution dynamics of large programs," IBM, Yorktown Heights, NY, RC 5615 (#24294), Sept., 1975.

[5] P. Brinch Hansen, *The Architecture of Concurrent Programs*. Englewood Cliffs, NJ: Prentice-Hall, 1977.

[6] F. P. Brooks, *The Mythical Man-Month: Essays on Software Engineering*. Reading, MA: Addison-Wesley, 1975.

[7] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, *Structured Programming*. New York: Academic Press, 1972.

[8] P. Denning and J. Dennis, *Machines, Languages, and Computation*. Englewood Cliffs, NJ: Prentice-Hall, 1978.

[9] E. W. Dijkstra, "The structure of 'THE'-multiprogramming system," *Commun. Ass. Comput. Mach.*, vol. 11, no. 5, pp. 341-346, May 1968.

[10] ——, "Co-operating sequential processes," in *Programming Languages*. London, England: Academic Press, 1968, pp. 43-112.

[11] A. B. Ferrentino and H. D. Mills, "State machines and their semantics in software engineering," *Proc. IEEE Comsac '77*, pp. 242-251, 1977. (IEEE Catalog no. 77Ch1291-4C.)

[12] S. L. Gerhart, "Program verification in the 1980's: Problems, perspectives, and opportunities," ISI Rep. ISI/RR-78-71, Aug. 1978.

[13] C. C. Gottlieb and L. R. Gotlieb, *Data Types and Data Structures*. Englewood Cliffs, NJ: Prentice-Hall, 1978.

[14] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. Ass. Comput. Mach.*, vol. 12, pp. 576-583, 1969.

[15] ——, "Monitors" an operating system structuring concept," *Commun. Ass. Comput. Mach.*, vol. 18, p. 95, 1975.

[16] W. J. Karplus, "The coming crisis in graduate computer science education," *UCLA Comput. Sci. Dep. Quarterly*, Jan. 1977, pp. 1-5.

[17] R. C. Linger, H. D. Mills, and B. I. Witt, *Structured Programming: Theory and Practice*. Reading, MA: Addison-Wesley, 1979.

[18] B. Liskov and S. Zilles, "An introduction to formal specifications of data abstractions," in *Current Trends in Programming Methodology*, vol. 1, R. Yeh, Ed. Englewood Cliffs, NJ: Prentice-Hall, 1977, pp. 1-32.

[19] H. D. Mills, "On the development of systems of people and machines," in *Lecture Notes in Computer Science 23*. New York: Springer-Verlag, 1975.

[20] ——, "Software development," *IEEE Trans. Software Eng.*, vol. SE-2, pp. 265-273, 1976.

[21] M. Moriconi, *A system for incrementally designing and verifying programs*, Ph.D. dissertation, Univ. of Texas, Austin, TX, Nov. 1977.

[22] D. L. Parnas, "The use of precise specifications in the development of software," in *Information Processing*, B. Gilchrist, Ed. Amsterdam, The Netherlands: North Holland, 1977, pp. 861-867.

[23] J. Ullman, *Principles of Data Base Systems*. Washington, DC: Computer Science Press, 1980.

[24] B. J. Walker, R. A. Kemmerer, and G. J. Popek, "Specification and verification of the UCLA unix security kernel," *Commun. Ass. Comput. Mach.*, vol. 23, no. 2, pp. 118-131, Feb. 1980.

[25] G. M. Weinberg, *The Psychology of Computer Programming*. New York: Van Nostrand Reinhold, 1971.

[26] N. Wirth, *Systematic Programming*. Englewood Cliffs, NJ: Prentice-Hall, 1973.

[27] ——, *Algorithms + Data Structures = Programs*. Englewood Cliffs, NJ: Prentice-Hall, 1976.

[28] ——, "Toward a discipline of real-time programming," *Commun. Ass. Comput. Mach.*, vol. 20, pp. 577-583, 1977.