"Software Engineering" programs have become a source of contention in many universities. Computer Science departments, many of which have used that phrase to describe individual courses for decades, claim SE as part of their discipline. Yet some engineering faculties claim it as a new specialty among the engineering disciplines. This article discusses the differences between traditional CS programs and most engineering programs, and argues that we need SE programs that follow the traditional engineering approach to professional education.

Software Engineering Programs Are Not Computer Science Programs

David Lorge Parnas, McMaster University

Since 1967, when a group of people from a variety of disciplines (most of whom would now be identified as computer scientists) met to discuss "Software Engineering" in southern Germany, computer scientists have discussed SE as if it were a subfield of computer science. Within CS departments we find people who specialize in automata theory, language design, operating systems, theorem proving, software engineering, and many other areas. Students take courses in a variety of subjects such as compilers, database systems, and, also, software engineering. Usually there is just one course entitled "Software Engineering," although sometimes we find faddish extras such as "Object-Oriented Software Engineering" or "Component-Based Software Engineering."

0740-7459/99/$10.00 © 1999 IEEE

In this article, I take a different view. Rather than treat software engineering as a subfield of computer science, I treat it as an element of the set {Civil Engineering, Mechanical Engineering, Chemical Engineering, Electrical Engineering, …}. This is not simply a game of academic taxonomy, in which we argue about the parentage or ownership of the field; the important issue is the content and style of the education. University programs in engineering are very different from programs in the sciences, mathematics, or liberal arts. These disparities derive from the differences in the career goals and interests of the students. I believe that many of our students, the ones who are destined for careers in software development, would be better served by an engineering style of education than they are by computer science education.

Attempts to distinguish two separate bodies of knowledge will lead to confusion.

It is important to stress that I am not comparing two areas of science. Just as the scientific basis of electrical engineering is primarily physics, the scientific basis of software engineering is primarily computer science. Attempts to distinguish two separate bodies of knowledge will lead to confusion. This article contrasts an education in a science with an education in an engineering discipline based on the same science. Recognizing that the two programs would share much of their core material will help us to understand the real differences.

Why Do We Need a New Type of Engineer?

Engineers are professionals whose education prepares them to use mathematics, science, and the technology of the day to build products that are important to the safety and well-being of the public. Because today's products are so varied that no individual can know everything necessary to design them all, engineering has split into many distinct specialties focusing on specific types of products. Civil engineers specialize in physical structures such as roads, bridges, and buildings. Chemical engineers are concerned with the design of plants and manufacturing processes for the chemical industry. Electrical engineers specialize in power delivery systems, electronics, communications devices, and so forth. Over the past three decades, it has become increasingly common to find that software is a major component of a wide variety of products—including many traditional engineering products. Further, software is used by engineers when designing other (noncomputerized) products; the correctness of their designs depends, in part, on the correctness of the software that they use. Over the same period, computer scientists have devoted a lot of effort to studying computers and programming. Today, we know far more about computing and programming than we did in 1967. Much of what we can teach to software developers today was not known to those who met at the original software engineering conferences sponsored by the NATO Science Committee in 1968 and 1969.

The increasing importance of software, combined with our increased knowledge about how to build it, has resulted in a need for graduates who, like other engineers, have received an education that focuses on how to design and manufacture reliable products, but who specialize in designing, building, testing, and "maintaining" software products. We can no longer squeeze what we know into a few courses in traditional engineering or computer science programs.

Our present approach to education for software professionals is not satisfactory to anyone. While many consumers decry the poor quality of software products, software companies complain about a shortage of highly qualified personnel. Employers also complain that they do not know what a graduate of either a CS program or a traditional engineering program can be expected to know about software development. Because of a rigid accreditation process (more on this later), there is a well-documented "core body of knowledge" for each of the established engineering disciplines. There is no corresponding body of knowledge for computer science. I have not been able to identify a single piece of knowledge or technique that is always taught in all CS programs. This article argues that the introduction of accredited professional programs in SE, programs that are modeled on programs in traditional engineering disciplines, will help to increase both the quality and quantity of graduates who are well prepared, by their education, to develop trustworthy software products. I also argue that, just as the introduction of electrical engineering programs did not eliminate the need for physicists, we will continue to need CS programs. In fact, I believe

that the introduction of SE programs will allow the development of even better CS programs than we have now.

Software Engineers Are Not Just Good Programmers

In many places, for example in job advertisements, "Software Engineer" is used as a euphemism for "programmer." Many writers seem to assume that the sole responsibility of a software engineer is to write well-structured code. These assumptions reflect ignorance about the historical and legal meaning of "Engineer." An Engineer is a professional who is held responsible for producing products that are fit for use. To be sure that a product is fit for use requires an understanding of the environment in which it is used. Consequently, those who are called "Software Engineers" need to know many things that are not part of computer science. Software is never used in isolation from other engineering products; it is a component in a system containing physical components and it is used to compute information about physical systems. While SE programs must reflect the fact that their graduates will specialize in software design, they must also equip their students with enough knowledge about other areas of engineering that they will know when to call for help from other engineers and can work well in a team with other types of engineers.

A Historical Analog

It has happened before; as an area of science becomes more mature, educational institutions develop an engineering program that is based on that science. For example, as our understanding of the physics of magnetic and electric fields improved, a new specialty within engineering, electrical engineering, was identified and the corresponding educational programs were developed. Although some physicists were heard grumbling that this was all physics and asserting that they could teach it all in an applied physics program, most universities developed EE programs that now flourish alongside physics programs and previously existing engineering programs.

Questions like the ones that we are now asking were asked then, for example, "If electrical engineering is based on physics, why do we need two programs?" "Why don't EE students just study physics?" It is clear that two programs are needed, not because there are two areas of science involved, but because there are two very different career paths. One career path is followed by graduates whose role will be designing products for others to use. The other career path is that of graduates who will be studying the phenomena that interest both groups and extending our knowledge in this area.

I do not wish to suggest that computer science will become exclusively theoretical or that Software Engineers will never do mathematical work. Physicists still build things, and many EEs publish highly mathematical papers. Rather, it is a question of goals. Physicists are primarily expected, and trained, to extend our knowledge, while EEs are expected to develop products or techniques for product production.

Each career path attracts a distinct type of student, requiring a distinct education.

Each career path attracts a distinct type of student and requires a distinct educational program. Most students choose to study EE rather than physics because they like building things. Those who study physics are often more excited by learning than by building. We can all name exceptions to these rules, but we do not design educational programs for the exceptions. There are many opportunities for people to deviate from their original plans later in life.

When the dust settles down and there are two distinct educational programs (both stable and both functioning well), we will find that CS and SE complement each other and cooperate in much the same way that science and engineering departments do. We will also find that students might want to switch between CS and SE just as the occasional student now switches between science and engineering.

How Does Science Education Differ from Engineering Education?

Although few people ever bother to compare and contrast them, a science education is very different from that received by engineering students. These differences are not accidental; they are based on the different needs of two very different types of careers.

Future scientists, who will add to our knowledge

base, must learn

- what is true (an organized body of knowledge about the phenomena of interest),
- how to confirm or refute models of the world, and
- how to extend our knowledge of what is true in their field.

In other words, scientists learn science plus the scientific methods needed to extend it.

Future engineers, who will design trustworthy products, must learn

- what is true and useful in their chosen specialty (that organized body of knowledge again),
- how to apply that body of knowledge,
- how to apply a broader area of knowledge necessary to build complete products that must function in a real environment, and
- the design and analysis discipline that must be followed to fulfill the responsibilities incumbent upon those who build products for others.

In other words, engineers learn science plus the methods needed to apply it.

For science students, keeping up-to-date on the most recent research in their specialty is essential, but a research scientist's knowledge can be very focused. In contrast, for practicing engineers, it is important to have relatively broad knowledge, but in most cases it suffices for them to be aware of only the scientific knowledge and technology that has already been proven to be reliable and effective in applications.

Engineers learn science plus the methods needed to apply science.

An illustration of the difference in emphasis is provided by a dispute between EE and physics departments in the late '60s. Some physics departments wanted to revise the shared physics course, reducing the coverage of electric and magnetic fields so that they could add a section presenting the results of the latest research on atomic particles. EE departments objected, pointing out that its students did not need to know about the newest particles, but must be competent in designing electric and electronic devices. Both departments were correct about the needs of their own students, but the needs of engineering students and physics students were not the same.

These differences between science and engineering programs make sense because of the following:

- If you are going to do specialized research, extending science, you can afford to be narrow but you cannot afford to be out-of-date. If you are going to apply science to build reliable products, you rarely need the very latest scientific research results, but you must have a broad understanding that makes you aware of the many factors that should be taken into account when designing a product.
- If you are carrying out scientific research, you can expect your results to be checked by those who read and referee your reports and papers. However, as Richard DeMillo, Richard Lipton, and Alan Perlis[1] pointed out many years ago, if you are designing a product, your work is not likely to undergo that kind of scrutiny. Engineering educators stress that engineers must accept responsibility for the correctness of their own designs.
- Scientists frequently work on narrow problems or in teams; licensed professional engineers often take responsibility for some complete product, which means that they require extensive knowledge outside of their engineering specialty. A mechanical engineer might have to do some electrical power design, or an electrical engineer might have to look at the mechanical aspects of a motor or servomechanism. Traditional engineering programs are designed with this in mind. A Professional Engineer is required to know when to talk to other engineers and to know enough to communicate easily with engineers who have other specialties.

Of course, the availability of well-educated Software Engineers will not eliminate the need for computer scientists. While there is a very strong need for engineers specializing in software design, the field is young and there is also a need for people who will experiment with tools and methods that will be used by the software engineers and extend our knowledge of how to design computer systems. Many open issues remain and studying these will require well-educated scientists, not engineers.

The Role of Accreditation in Engineering

The work of scientists will usually be judged by other scientists, but engineers often deal directly with customers who are neither engineers nor scientists. Thus, while nobody has ever felt it necessary to hold science programs to rigid standards, accreditation has always been an important consideration for engineering programs. In Canada and most US states, legislation limits those who may practice engineering to those licensed by designated profes-

sional engineering societies. These organizations have joined to create accreditation boards for university programs in engineering. Licenses are issued primarily to those who have graduated from accredited programs, but individuals who have obtained the requisite knowledge in other ways may apply for individual examination and, if they pass the required examinations, be licensed.

Science programs are subject to review by the universities that offer them because good institutions are always concerned about the quality of their offerings, but for science programs there is nothing corresponding to accreditation by the Canadian Engineering Accreditation Board or the Accrediation Board for Engineering and Technology. The Canadian Information Processing Society does offer a voluntary accreditation program, but the documentation describing that program states explicitly that there are "no rigid standards." Many of the stronger CS departments do not bother with accreditation, and some of those that do, don't take it seriously. In contrast, CEAB and ABET requirements are quite rigid, and accreditation visits are major events for all engineering departments. Because use of the title "Engineer" is restricted by law, the standards are demanding and are taken seriously by all who offer engineering programs.

Why do we need new laws and accreditation mechanisms when the existing ones can be exploited to meet the need?

In my experience, the accreditation process for engineering programs is very effective in raising the quality of educational programs and in assuring that all graduates have been exposed to the most important ideas and how to use them. Software engineering will not achieve the status of a true profession until it has a similar accreditation system. The easiest and best path to establishing such a system is to treat software engineering as just another specialty within engineering.

What will Software Engineers Do?

The first step in developing Software Engineering will be to do what has been done for other engineering disciplines—identify a core body of knowledge. This process must begin with a description of the tasks that we expect them to be able to perform. The following are the key steps in the development of computer systems. A software engineer should be prepared to participate in each of these steps.

- Analyze the intended application to determine the requirements that must be satisfied and record those requirements in a precise, well-organized, and easily used document.
- Participate in the design of the computer system configuration, determining which functions will be implemented in hardware, which functions will be implemented in software, and selecting the basic hardware and software components.
- Analyze the performance of a proposed design (either analytically or by simulation) to make sure that the proposed system can meet the application's requirements.
- Design the basic structure of the software, that is, its division into modules, the interfaces between those modules, and the structure of individual programs while precisely documenting all software design decisions.
- Analyze the software structure for completeness, consistency, and suitability for the intended application.
- Implement the software as a set of well-structured and well-documented programs.
- Integrate new software with existing or off-the-shelf software.
- Perform systematic and statistical testing of the software and integrated computer system.
- Revise and enhance software systems, maintaining their conceptual integrity and keeping all documents complete and accurate.

Like all engineers, software engineers are responsible for the usability, safety, and reliability of their products. They are expected to be able to apply basic mathematics and science (including the relevant computer science), to assure that the system they design will perform its tasks properly when delivered to a customer.

Many computer scientists whose specialty is "Software Engineering" would add other things to this list. For example, they would point out that software engineers must know how to work in teams, must know how to make schedules, set deadlines, estimate costs, and other project management functions. At some institutions it is project management

that dominates the "Software Engineering" courses. However, while these activities might distinguish the work of software developers from that of academics, they do not distinguish the work of Software Engineers from that of other engineers. I consider some courses on these aspects of project work to belong in the core of all engineering programs; extra courses may be taken by those who are especially interested in engineering management.

Why Does the Distinction between Computer Science and Software Engineering Appear Fuzzy?

Many faculty in Computer Science departments believe that they are already teaching software engineering. Some departments claim that there is no need for a new program and may offer a "Software Engineering" track or specialization within computer science.

As Spinoza wrote, "Nature abhors a vacuum."[2] Faculties of Engineering have ignored software for far too long. Society in general, and industry in particular, need software developers and have turned to several other sources:

- Engineers, and others, have learned about software in ad hoc ways after graduation.
- Various educational programs have included some software courses in their programs, and software has also been taught as part of other courses.
- Computer Science departments have tried to fill the gap by including so-called "Systems" or "Applied Computer Science" courses in their offerings.

It would be much easier to identify Software Engineering as a new branch of engineering than to set up an entirely new accreditation and licensing system.

Those who are now doing software development work have followed one of these paths; many now see a degree in CS as the best preparation for such careers. There was no other choice! Today's CS programs are rarely pure science programs, but neither are they programs that could win accreditation as engineering programs.

Following the traditions of education in the sciences, CS programs offer students much more freedom than traditional engineering programs. This in itself would make accreditation very difficult because accreditation committees look for the "weakest path" through a program and will not accept a program if even one possible path does not meet their criteria. A deeper problem is that, in the tradition of science programs, where experimental and theoretical science are often viewed as competing subfields, there is often little connection between the theoretical and practical sides of CS programs. Certain courses (for example, Denotational Semantics) are identified as theoretical, others (for example, Compiler Construction) as practical, and there are few places where material from one type of course is used or illustrated in another. This is in sharp contrast to engineering programs, where it is considered important to teach how to apply theory when solving practical problems—that is, to integrate, rather than separate, mathematics and design. Finally, the "practical" computer scientists have sometimes confused technology with engineering. Many courses fail to stress fundamentals and are organized around current fads and buzzwords. It is also amazing how many of these courses teach about very specific systems or languages and stress material that will be obsolete before the student graduates.

The need for something like accreditation in the software area has not escaped the attention of industry or academics with a practical inclination. For several years, a US committee of computer scientists has been asking how a similar mechanism could be created for software developers.[3] However, there is little visible progress. One must ask why we would need new laws and accreditation mechanisms when the existing ones could be exploited to meet the need. It would be much easier to identify Software Engineering as a new branch of engineering than to set up an entirely new accreditation and licensing system.

It is the attempt by many CS programs to fill both roles that makes it difficult for some to see why we need a separate SE program. With the benefit of hindsight, we can see that separating EE and Physics was a good idea. It allows both programs to do their job better. Neither has to be a compromise. We will reach the same conclusion about SE and CS within the next decade.

It is the isolation of CS and engineering departments from each other that makes it difficult for some engineers to understand the need for an SE program. Few engineers have kept up with computer science, and consequently few realize how much useful material has been developed in the last

30 years. For many engineers, the shared scientific base of SE and CS programs includes much material that they do not personally know and, consequently, they cannot appreciate the relevance of that material to engineering.

Software Engineering Is Not Just about Software Used by Engineers

Some people from both computer science and engineering have the impression that SE programs emphasize software that is used in traditional engineering applications. Many traditional engineers and computer scientists seem to believe that the work of engineers is limited to the construction of physical products. This has not been true for several decades. Today many of our most important products are intangible. Thirty years ago, designing a directional antenna meant "cutting tin"; today, it means writing programs.

Engineers are taught how to apply science and mathematics to design products that will be used by others. This is especially important in those situations where the safety and well-being of the public depends on the correct design of those products. However, many engineers work on other products as well. Those products include financial systems and other systems outside of the traditional engineering areas. Computers are general-purpose tools and our graduates should be equally flexible. The basic software design principles and techniques apply to all kinds of software, and our graduates should be as prepared to work for banks as for steel companies.

How Should Software Engineering and Computer Science Programs Differ?

With this background, I can outline how SE and CS programs should differ.

Differences in curriculum philosophy

♦ An SE program should be designed for accreditation as an engineering program. The CS programs need not be subject to those restrictions.

♦ An SE program will be relatively rigid with few technical options. CS programs should continue to offer the traditional possibilities for specialization.

♦ An SE program, like many other engineering programs, might not require students to pick their specialty within engineering until second year. It can share a common first year with the other engineering programs. CS programs can start presenting specialized material earlier.

♦ An SE program should stress breadth, that is, be designed to make sure that its graduates have some familiarity with the most important engineering topics. The CS program should continue to offer students a chance to be curiosity-driven when choosing courses and should allow specialization.

Thirty years ago, designing a directional antenna meant "cutting tin"; today, it means writing programs.

♦ An SE program should include a lot of the basic material taken by most other engineering students (control theory, for example). This material would not be required for CS students.

♦ An SE program should stress the application of computer science in a variety of areas. The CS program should stress understanding the inherent properties of computer systems and focus on support software and program development tools.

♦ CS programs can spend more time discussing research areas that are not yet routine or even ready to use (for example, genetic algorithms). SE programs should place more emphasis on the vast amount of material that has already been proven practical.

♦ The SE program, while strong in theory, should stress the application of that theory. The CS program should allow students to study theory for its own sake and prepare them for work that extends or refines the theory.

♦ The SE program should have a strong stress on end-user applications; CS programs should prepare their graduates to develop new tools for software developers to replace the rather primitive and ad hoc tools that are now available.

♦ In SE courses, theory and practical considerations must be integrated. In CS, I would expect the traditional specialized courses to continue.

Differences in topic coverage

Many topics in computer science are interesting and challenging but have not yet found practical application. The denotational semantics of programming languages is one such area. At the risk of offending some of my favorite people, I venture to

say that one can build sound software systems without any knowledge of this field. In contrast, one could hardly call oneself a computer scientist without some familiarity with this topic. Again risking the ire of my colleagues, I could make similar remarks about neural computation, many parts of artificial intelligence, and some aspects of computability and automata theory. Some discussion of those topics must be included in CS programs. In contrast, a graduate of an SE program should understand aspects of communication, control theory, and interface design that are rarely seen in CS programs.

Most of my engineering students become impatient if they are not shown how to apply what they are learning.

There are many advanced courses in CS where the most important thing learned by the students is how to invent new algorithms or design new tools. Many of these will not be as important in the SE program, where the stress will be on selecting from known algorithms and applying tools and technology that were developed by others.

Every educational program is a compromise. Any topic that is essential in one of these programs would be of potential interest to the students in the other program. However, the limited length of university programs will force us to make choices based on priorities. In the SE program, the priority will be usefulness and applicability; for the CS program it is important to give priority to intellectual interest, to future developments in the field, and to teaching the scientific methods that are used in studying computers and software development.

Differences in course style and content

Having taught both engineering and computer science students at several institutions, I see important differences between them. I have found most CS students relatively patient and willing to explore topics just because they are interesting. In contrast, most of my engineering students become impatient if they are not shown how to apply what they are learning. For many engineering students the remark "That course is mostly theory" is strong criticism; for many CS students it is praise. Similarly, when the EE department head at Carnegie Mellon told a visitor that my work was "intellectual and very abstract," he had chosen a polite way to say "useless." Had my CS department head used that same phrase, there would have been no such negative connotation. These differences must be reflected in curricula and course outlines.

Many topics should be covered in both programs, but we might have to give quite different courses. For example, I consider mathematical logic to be a fascinating topic that is obviously important for both sets of students. In teaching logic to SE students, I find it essential to emphasize the use of logic to describe properties of systems and properties of states. In the SE program, I would also emphasize the role of logic in checking specifications and programs for completeness and consistency. Deduction or proof would be discussed, and students would be given the opportunity to use theorem-proving software, but the differences between types of logic would not get much time. In contrast, in a CS course on logic, students would learn the differences between various kinds of logics, and to discuss issues such as generalized decision procedures and the meaning of non-denoting terms for which the SE course would have little time. Remember that in one program we are teaching students to apply well-established techniques, in the other we should teach them how to add new elements to that set of techniques.

Another example might be courses on operating systems. I have found it interesting (and useful) to teach a taxonomy of operating systems, classifying them by features and properties in much the way that a biologist might classify insects. One can even get insight from evolutionary studies, looking at how ideas moved (often with people) from one operating system to another. Such a course would be very important to someone who is going to investigate how to build new operating systems or develop new models and theories. However, that course would not be the best way to teach SE students what they need to know about operating systems. They would be less interested in the history or comparative anatomy of the systems; they want to know how to select a system to use and how to use it. No sound engineering program can afford to stress the details of one particular system because that system might be out-of-date before the students graduate, but we can teach basic principles that help the student to make good choices and to use any system effectively. It is also true that few engineering graduates will end up designing new operating systems; they are much more likely to use existing ones. However, much of what is often taught as part of a course on the design of operating systems is rel-

evant to the design of many other interactive and real-time systems. That material should be included in an advanced software design course.

Finally, many CS programs, like other science programs, prepare students to continue their studies in a graduate program. In contrast, traditional engineering programs focus on preparing students to enter the workforce immediately after completing their undergraduate program.

A Sketch of a Software Engineering Curriculum

If I were to design a program that would add Software Engineering to the family of engineering programs, it would comprise

- basic courses taken by all other engineering disciplines,
- courses for software engineers that provide an overview of basic engineering issues,
- courses on the mathematical foundations of software engineering (stressing applications in software development), and
- software design courses.

In addition, all engineers would take the usual set of "complementary studies" courses. Below is a possible curriculum. Each item would be a one-semester course.

Courses shared with most other engineering disciplines

Many SE graduates will work in teams with other engineers and should share a common knowledge base with them. Most of the following courses are part of the core engineering program and are taken in the first year.

G1. General Chemistry for Engineering
G2. Engineering Mathematics Ia (linear systems, matrices, complex numbers)
G3. Engineering Mathematics Ib (continuation of above)
G4. Calculus for Engineering I
G5. Introductory Mechanics
G6. Engineering Design and Communication
G7. Safety Training (1 unit)
G8. Calculus for Engineering II
G9. Waves, Electricity, and Magnetic Fields
G10. Engineering Mathematics IIa (differential equations, transforms)
G11. Engineering Mathematics IIb (vector calculus, coordinate systems)
G12. Introductory Programming for Engineers
G13. Engineering Economics

Courses introducing other engineering areas to software engineers

The software engineer cannot be a universal engineer; the program cannot deal with certain engineering topics in the same depth as other programs. However, it should provide a good overview of engineering materials, control, heat transfer, and computer engineering.

E1. Introduction to the Structure and Properties of Engineering Materials
E2. Introduction to Dynamics and Control of Physical Systems
E3. Digital System Principles and Logic Design for Software Engineers
E4. Architecture of Computers and Multiprocessors
E5. Introduction to Thermodynamics and Heat Transfer

Applied mathematics

Each of these courses introduces an area of mathematics that is important for software engineering and not always taught to other engineers. In each of these courses, examples and assignments will show how mathematics can be used when designing software. Moreover, where possible, mathematical packages will be used to give students practical experience in using the concepts.

Products containing software and products designed by software are important to the public's safety and well-being.

M1. Applications of Mathematical Logic in Software Engineering
M2. Applications of Discrete Mathematics in Software Engineering
M3. Statistical Methods for Software Engineers

Software courses

These are the "core" of the program, presenting computer science material and showing how it can be used to design successful software products.

S1. Software Design I: Programming to Meet Precise Specifications
S2. Software Design II: Structure and Documentation of Software

S3. Design and Selection of Computer Algorithms and Data Structures
S4. Machine-Level Computer Programming
S5. Design and Selection of Programming Languages
S6. Communication Skills: Explaining Software
S7. Software Design III: Designing Concurrent and Real-Time Software
S8. Computational Methods for Science and Engineering
S9. Optimization Methods, Graph Models, Search, and Pruning Techniques

We must have some assurance that those practicing software engineering have graduated from a program in which the most important basic material has been covered.

S10. Data Management Techniques
S11. Software and Social Responsibility
S12. Design of Real-Time Systems and Computerized Control Systems
S13. Fundamentals of Computation
S14. Performance Analysis of Computer Systems
S15. Design of Human–Computer Interfaces
S16. Design of Parallel and Distributed Computer Systems and Computations
S17. Software in Communications Systems
S18. Computer Networks and Computer Security
S19. Senior Thesis I
S20. Senior Thesis II
S21. Technical Elective I
S22. Technical Elective II

The ideas in this article can be summarized by the following observations.

Software Engineering is different from Computer Science. An examination of the program I just outlined shows that an educational program that treats SE as a branch of engineering is quite different from the specialized computer science programs entitled "Software Engineering." Classical CS courses such as compilers and operating systems are missing from this program (because the graduates are unlikely to design those products), but the material that can be used in other applications (for example, scanning algorithms, machine representation of data structures, synchronization of concurrent activities) has been distributed among other courses. CS programs tend to focus on "core" software areas, but there is a growing need for people to develop software for new applications and applications where software is replacing or supplementing traditional engineering technologies. Approximately half of the required courses present material that is not taught to computer science students, but is important for a growing number of software applications. The program presented earlier focuses on fundamental design principles that are applicable in both the classic CS areas and the broad class of applications where well-educated developers are badly needed.

Software Engineering programs should be accredited. Products containing software and products designed by software are so important to the safety and well-being of the public that we must have some assurance that those practicing software engineering have graduated from a program in which the most important basic material has been covered. Licensing of Software Engineers who are in private practice is just as important as the licensing of Civil Engineers.

Software Engineering education can, and must, focus on fundamentals. When I began my EE education, I was surprised to find that my well-worn copy of the *RCA Tube Manual* was of no use. None of my lecturers extolled the virtues of a particular tube or type of tube. When I asked why, I was told that the devices and technologies that were popular then would be of no interest in a decade. Instead, I learned fundamental physics, mathematics, and a way of thinking that I still find useful today. Clearly, practical experience is essential in every engineering education; it helps the students learn how to apply what they have been taught. I did learn a lot about the technology of the day in laboratory assignments, in my hobby (amateur radio), as well as in summer jobs, but the lectures taught concepts of more lasting value that, even today, help me to understand and use new technologies.

Readers familiar with the software field will note that today's "important" topics are not mentioned. "Java," "Web technology," "component orientation," and "frameworks" do not appear. The many good ideas that underlie these approaches and tools must be taught. Laboratory exercises and other projects should provide students with the opportunity to use the most popular tools and to experiment with some new ones. However, we must remember that these topics are today's replacements for earlier fads and panaceas and will themselves be replaced. It is

the responsibility of educators to remember that today's students' careers could last four decades. We must identify the fundamentals that will be valid and useful over that period and emphasize those principles in the lectures. Many programs lose sight of the fact that learning a particular system or language is a means of learning something else, not a goal in itself.

We will need new courses, not a new combination of existing courses. At some institutions, the debate about "Software Engineering" was treated as a jurisdictional dispute between the CS and ECE departments. The dispute is sometimes "resolved" by including some courses from each department in a new program. This type of compromise will produce graduates that are neither engineers nor computer scientists. It is essential that the CS material be taught in the engineering style.

Teaching style and course organization must change. It is also important that the software courses be taught differently from conventional CS courses. In science courses, it is quite reasonable to teach about things, but in engineering courses, students are taught how to do things. With engineering students one cannot simply fill the board with derivations and proofs. Each course must integrate theory and practice and stress how to apply the theory when designing.

Most of us teach what we were taught and teach it in the way that we were taught it. Since nobody has yet graduated from an SE program of this type, we will have to teach unfamiliar material or teach familiar material in unfamiliar ways. Until programs like this one are well established, there will have to be very careful and detailed course content specifications as well as careful coordination and supervision.

Most of us teach what we were taught and teach it in the way that we were taught it.

Staffing will be the most critical problem. Finding appropriate teachers for this type of program will be critical and difficult. Students who choose engineering as a career path are people who want to learn how to design and analyze real systems. It is important that their teachers be people who know how to do those things and who are interested in building products. Too many of today's computer scientists, even those who identify their area of interest as software engineering, are interested in abstractions, and are reluctant to get involved in product design or even to look closely at what is being done in practice today. On the other hand, few of today's engineers know enough computer science to teach these courses properly.

Since experience producing software products is essential, and we all have a limited amount of time, we are going to have to use different standards when recruiting. It takes much longer to produce a software product than to write a paper. We can't compare paper counts for people with practical experience with those for people who have been pure researchers. In some cases, experience and insight might be worth more than an advanced degree. In engineering, practical experience is valued more highly than it is in some of the sciences. This will lead to serious conflicts in "mixed" appointment committees.

Computer Science maturity allows us to offer SE programs. It is only because of the maturity of CS, because of the many results that have been obtained in the last 30 years of CS research, that we can now start SE programs. Without the research carried out since the first NATO-sponsored SE conferences, we would not have enough teachable knowledge to justify starting a new branch of engineering. Because computer science must continue to develop, it is important to have both an engineering program and a science program. By developing two complementary programs—one for scientists, the other for engineers—we have a unique opportunity to do both jobs well. Neither program will need to make uncomfortable compromises to do the work of the other.

Only real cooperation will serve the students properly. In speaking on this subject elsewhere, and in my own institution, I have encountered deep, sincere, and determined opposition to the idea that SE be treated as a new branch of engineering. On the engineering side, I see lack of recognition of the large body of knowledge that has been accumulated about how to write software. Many engineers seem to believe that programming is simply learning language and operating-system conventions. Some argue that any engineer who writes programs is a software engineer, and believe that you need no special expertise or experience to establish and run a SE program. Others believe strongly that you cannot have an engineering discipline whose scientific base is outside the physical sciences. Programs such as that proposed here are seen as "narrow"—that is,

too heavy on programming. Most of the engineers who have reviewed the program have told me I should replace many of the S (software) courses with additional E courses (introducing other engineering areas to software engineers). In other words, they do not view software engineering as an engineering discipline.

On the computer science side, the reaction is no less negative but the reasons are more complex. A major factor seems to be computer scientists' (mistaken) fear that something is being stolen from them; they feel that they, not the Faculty of Engineering, should design and control any SE programs. Equally serious is a lack of understanding of the ways that engineering education differs from the education that they received. Several people who have successfully made the transition from teaching in other disciplines (such as physics or mathematics) to teaching in an engineering program have told me how much their teaching style and course content had to change. Few computer scientists have made this transition or recognize the need to make it. Finally, few CS faculty seem to recognize that to be a good Software Engineer, you must be much more than a good programmer. Consequently, most of the CS graduates who have reviewed the program have told me that much of the material in the E courses (designed to introduce other engineering areas to software engineers) is irrelevant and should be replaced by additional S (software) courses.

Educating software engineers who are prepared to work as Professional Engineers cannot be done by either computer scientists or engineers working alone. It is our responsibility to our students to work together, and each group must be prepared to learn from the other. ❖

Acknowledgments

So many people have provided both interesting and helpful comments on earlier versions of this article that I cannot list them all. The members of the Ad Hoc Curriculum Committee for Software Engineering at McMaster University (SanZheng Qiao, Paul Taylor, Ryszard Janicki, and ZhiQuan Luo) helped in developing the curriculum. I am most thankful to all who have argued directly with me, making it possible for me to understand their positions and improve my own.

This article was originally published as "Software Engineering Programmes Are Not Computer Science Programmes" on pp. 19–37 of the Software Engineering Education special issue of *Annals of Software Engineering*, Vol. 6, Apr. 1999, edited by N.S. Coulter and N.E. Gibbs. More information can be found at http://manta.cs.vt.edu/ase.

References

1. R.A. DeMillo, R.J. Lipton, and A.J. Perlis, "Social Processes and Proofs of Theorems and Programs," *Fourth ACM Symp. Principles of Programming Languages*, 1977, pp. 206–214. Also published in *Comm. ACM*, Vol. 22, No. 5, May 1979, pp. 271–280.
2. B. Spinoza, *The Ethics of Spinoza*, Part I, Carol Publishing Group, New York, 1982.
3. B.K. Boehm, "The IEEE–ACM Initiative on Software Engineering as a Profession," *IEEE Computer Soc. Software Eng. Tech. Council Newsletter*, Vol. 13, No. 1, Sept. 1994, p. 1.

About the Author

David Lorge Parnas is the NSERC/Bell Industrial Research Chair in Software Engineering in the Department of Computing and Software, Faculty of Engineering, at McMaster University. He is a fellow of the ACM and the Royal Society of Canada and a senior member of the IEEE.

Parnas has a BS, MS, and PhD in electrical engineering from Carnegie Institute of Technology. He also has honorary doctorates from the ETH in Zurich and the University of Louvain in Belgium.

Contact Parnas at the Dept. of Computing and Software, Faculty of Engineering, McMaster Univ., Hamilton, ON L8S4K1, Canada.