Software Engineering Programs: Dispelling the Myths and Misconceptions

Hossein Saiedian, *University of Kansas*
Donald J. Bagert, *Rose-Hulman Institute of Technology*
Nancy R. Mead, *Software Engineering Institute*

Some people think new software engineering degree programs address industrial software development problems; others argue that they are merely an opportunity to provide industrial training in programming. The authors address these and similar issues, discussing commonly held myths about such programs.

In a now classic 1994 *Scientific American* article, W. Wayt Gibbs described software crises in both the private and government sectors.[1] The problems he discussed ranged from overrunning budgets and schedules to terminating projects despite multimillion-dollar investments. Similar concerns were reported as recently as March 2001 in the *Communications of the ACM*, where several authors made grim predictions about the future of software engineering (SE) if the industry continues with "business as usual."[2]

In recent years, software researchers and developers have explored numerous principles for improving software practices, some of which have proven effective in practical projects. These include software development methodologies and environments, structured and object-oriented programming, software process improvement models (such as the CMM), Computer-Aided Software Engineering (CASE) tools, and fourth-generation languages. Nevertheless, we have not completely resolved our software problems, and many organizations continue to suffer from bad practices.

One way to improve practice is to focus on properly educating the next generation of SE professionals. However, the debate about the most effective approach to educating this next generation continues unabated. Some argue for an SE track under existing computer science (or computer engineering) programs. Others promote specialized and independent SE degrees at the graduate as well as undergraduate levels. Some universities have established such specialized programs, hoping that they will address all industrial SE problems. Unfortunately, universities often don't know what new programs should offer or when or at what level they are most appropriate. Furthermore, traditional computer scientists have criticized these programs as merely providing an opportunity to offer industrial training in programming. In many ways, the current situation mirrors that of the computer science field in the 1960s and 70s: electrical engineering and mathematics faculty initially resisted the growth of computer science degree programs just as current computer faculty are treating SE today.

Contributing to the hype about and harsh criticism of SE programs are some widely held

0740-7459/02/$17.00 © 2002 IEEE

Adding a new degree program will lead to a better image, but we must calculate the associated costs to determine whether the program will be cost effective in the long run.

myths. This article examines these myths, dispelling the misconceptions to defuse unnecessary concerns, conflicts, and distractions and help provide an appropriate context and direction for new SE degree programs.

Myth 1

A new software engineering degree program is an academic necessity.

There seems to be a rush to develop new SE degree programs (especially at the graduate level) simply because many SE faculty members believe that adding such programs will improve the reputation of their departments and institutions. Adding a new degree program will lead to a better image, but we must calculate the associated costs to determine whether the program will be cost effective in the long run. In certain market areas, such programs are absolutely necessary and a welcome addition; in other areas, they won't have such an impact and thus might become a burden.

Institutions should not develop new programs based on image-enhancing effects, popular trends, or peer pressure. Rather, they should introduce a program only if it is necessary and has a valid and viable market. Those interested in introducing a degree program must objectively assess existing industrial needs, the potential pool of students, expertise among the existing faculty, and administrative support. Furthermore, developing a new SE program is not always the only means of addressing the needs of local industrial organizations or student requests. Many times, adding core SE courses in a computer science program or adding an SE focus area (or specialization) will address both real and perceived demands.

Myth 2

Software engineering programs will unnecessarily expend computer science resources.

The general consensus in both industry and academia is that computer science degree programs are worthwhile and should continue for the foreseeable future. Therefore, to meet the demand for computer scientists as well as the emerging need for those educated in SE, there must be sufficient resources (especially faculty) for both programs.

There is a severe shortage of faculty in all computing fields, including SE. Master's programs in SE often overcome this obstacle by using part-time adjunct faculty, but this is a less viable option for many undergraduate programs. With a small pool of potential new faculty available, some institutions must retrain computer science faculty to teach SE. However, finding people willing to undergo such retraining is more difficult than in industry, due to the rights that tenured faculty exercise at many academic institutions.

The discussion here assumes that a computer science department will house the SE program, perhaps becoming the Department of Computer Science and Software Engineering (CSSE). Because the disciplines are closely related, housing them in the same academic unit is the best option, letting the two faculties work together for their mutual good. However, at many institutions, there are difficult political issues involved in forming a CSSE department, especially if the Computer Science Department is not in a College of Engineering. Furthermore, people believe that an SE program in such a CSSE department would considerably drain computer science resources—but this is not necessarily the case.

Consider the following scenario: The curriculum for a computer science department in a College of Engineering has nearly half of its credit hours in the computer science department. In addition, there is a 90/10 percent split between the computer science and SE hours, which is consistent with the core material recommended for a computer science degree under *Computing Curricula 2001* (CC2001).[3] Thus, 90 percent of the computer science faculty must teach computer science courses, and the other 10 percent must teach software engineering classes.

Now suppose the department adds an SE degree program. The total number of credit hours computer science and SE majors take through the newly named CSSE Department are the same, but the split in hours is now 50/50 between the two disciplines. (This is consistent with the recommendations made in the *Guidelines for Software Engineering Education*,[4] which provides the SE undergraduate curriculum model most cited in recent literature.) Subsequently, one-third of the department's majors are in SE and the other two-thirds are in computer science. How does this affect the allocation of resources?

When academic programs discuss resources, they're generally referring to faculty. Suppose the Computer Science Department has 30 faculty members. Before the SE degree existed, the department would need 27 computer science instructors and three SE instructors, due to the 90/10 percent split in hours. Under this scenario, implementing the SE program would require a shift of four faculty members. The computer science majors (two-thirds of the total) would require 18 computer science and two SE instructors; the SE majors would need five computer science and five SE instructors, meaning that the computer science and SE split of the faculty would now be 23/7.

So, only 13 percent of the faculty (four out of 30) would need to shift to SE to implement the change. Considering the number of electrical engineering and mathematics faculty that changed to computer science as it was emerging, this seems reasonable, despite the retraining issues involved. Furthermore, it is likely that students who would have originally been computer engineering majors in an Electrical and Computer Engineering (ECE) Department will now be SE majors. Those students will take about the same number of computer science hours as in computer engineering, so there will be no additional drain on computer science faculty; any additional faculty needed would be in SE. So, this means that the department would gain more majors (at the expense of ECE) and (if available) more faculty to teach them.

We can vary the scenario, but it still indicates that creating an SE program in an CSSE Department will have minimal negative impact on the faculty. The positive benefits—more majors, faculty, and choices available to the student—far outweigh the disadvantages, and we would get better-educated software professionals without depleting the supply of computer science graduates.

Myth 3

Software engineering undergraduate programs do not have enough depth.

Many companies believe that a master's degree program in SE provides a sufficient overall background for future software professionals. Such programs typically require the student to have a minimal background of a set of undergraduate computer science courses such as data structures, discrete structures, design and analysis of algorithms, and operating systems, for a total of 24 to 30 hours. The question is, can an undergraduate SE program provide the computer science background needed while providing the additional SE topics necessary to educate a software professional?

Several sources indicate the type of core computer science background that should be required. Tim Lethbridge surveyed software professionals and reported that the 25 most important topics required of such individuals include computer science areas such as specific programming languages, data structures, object-oriented concepts, design of algorithms, operating systems, systems programming, databases, file management, and networks.[5] Providing undergraduate education in these topics would once again require 24 to 30 semester hours. This range of hours would also be sufficient for CC2001's core requirements and many of the requirements for computer science courses (outside of SE) specified in the criteria of the Computing Accreditation Commission of the Accreditation Board for Engineering and Technology.[6] (ABET is the accreditation body for engineering degree programs in the US.) Therefore, it is reasonable to assume that those same 24 to 30 computer science hours would be sufficient in an undergraduate SE curriculum.

As far as determining how many SE credit hours are required, a typical master's degree requires at least 24 semester hours of graduate course work in the major (SE, in this case). The question is, how does this translate to undergraduate hours? One option is to use a 3:2 ratio between undergraduate and graduate hours, which is commonly done in comparable courses in computer science and other disciplines. So 24 graduate hours would then translate to 36 undergraduate hours. It is possible (although difficult) to squeeze 24 to 30 hours of computer science and 36 hours of SE into an undergraduate SE curriculum. However, such comparisons are usually made using graduate courses that build on undergraduate classes in the same discipline, whereas a master's degree in SE requires little or no SE background when entering the program. Therefore, 36 undergraduate SE hours might be too many.

Many times, adding core SE courses in a computer science program or adding an SE focus area will address both real and perceived demands.

There must be a paradigm shift in attitude at the workplace. Existing software development professionals and managers must value and respect SE education.

The *Guidelines for Software Engineering Education* suggests that an undergraduate SE curriculum have 21 required hours of computer science, 24 required hours of SE, and nine hours of electives in either computer science or SE. That model was intended to satisfy CC2001 core computer science requirements and ABET criteria for SE undergraduate degree programs in the US. It also aims to cover the same material typically found in an SE degree program, all in a 120-semester-hour, four-year curriculum, which is generally the minimum requirement for a baccalaureate program in the US. (It is also interesting to note that virtually all of the 25 most important topics for a software professional cited in the Lethbridge survey—including those in computer science and SE—are also covered in the model from the *Guidelines* report, even though the latter was published first.) Such an SE curriculum would provide minimally sufficient depth for an SE major; allowing more than 120 hours (which is often the case for US engineering programs) would provide even more depth in computer science or SE.

Myth 4

A new SE degree will address industrial software development crises.

A new SE degree program will not be a panacea or a silver bullet. It will be one of the first steps, albeit the most important step, toward addressing industrial software development crises, but we also must consider complementary factors. For example, we need to clearly define the "engineering education" (the curricula and the style of presentation). A starting point is understanding the objective of an engineering education as David Parnas defines it.[7] He argues that to provide the most effective SE education, a new SE program must follow the traditional engineering approach to professional education while maintaining the scientific basis of SE (computer science). He emphasizes that an engineering education should teach engineers

- What is true and useful in their chosen specialty
- How to apply the body of knowledge
- How to apply a broader area of knowledge to build complete products that must function in a real environment

(In a related article, Mary Shaw compares the evolution of chemical engineering and civil engineering with today's SE. She uses her evolution model of engineering disciplines to identify the steps for enhancing the SE discipline.[8])

Another important issue is that of defining an acceptable body of knowledge for software engineers. The Software Engineering Body of Knowledge (SWEBOK) has been an excellent starting point, but it has certain deficiencies. One is its perceived North American bias; such a guide must obtain international acceptance. Yet another criticism surrounds its certification and licensing implications. We must clearly address such issues and invite other computing associations to join SWEBOK's development.

Furthermore, software engineers must be able to apply the methods in different contexts and tune their knowledge to more effectively use the new technologies. As Michael McCracken has observed,[9] academia cannot predict the next popular language or methodology industry will use, so the education provided to software engineers must focus on the fundamentals to prepare the new graduates to assimilate and apply new technology quickly and efficiently. A long-term aspiration would be to identify distinct roles in SE and provide appropriate education and specialized training for each.[10]

SE students (and education) must include an element of training, not only during their academic careers (for example, through internships) but once they enter the workforce and before important design and implementation responsibilities are delegated to them. This is not only true in other engineering disciplines, but also in nonengineering disciplines. (For example, in the medical field, new graduates go through at least two years of training as part of their residency program before they are allowed to engage in real practice.) Steve McConnell and Leonard Tripp suggest at least four years of apprenticeship for software engineers.[11]

In addition, there must be a paradigm shift in attitude at the workplace. Existing software development professionals and managers must value and respect SE (and computer science) education, acquaint themselves with the fundamentals and the body of knowledge, and update their personnel skills to avoid "cultural clashes" with newcomers. Otherwise, new SE graduates will be

unsuccessful in transitioning their new body of knowledge and will end up following unproven skills and the "code and fix" culture that dominates the workplace.[11]

Certification and licensing are equally important. Although formal education is crucial, a software engineer should also regularly (for example, every five years) prepare for and pass certification exams. This would assure that he or she has maintained a minimum understanding of the SE body of knowledge. Certification exams can then evolve into a kind of licensing exam, similar to licensing in other fields, to facilitate and assure professional competency and responsibility.

Myth 5

Computer science is to software engineering what chemistry is to chemical engineering.

Another widely held misconception pertains to the relationship between computer science and SE (which has been compared with that of chemistry to chemical engineering or physics to mechanical engineering). This is essential in understanding how educating software professionals differs from educating computer scientists.

This myth is tempting for SE faculty because it supports their contention that, over time, computer science will become more theoretical. In the long term, it will thus be difficult to find (more) room for SE in computer science curricula. The alternative then is to also develop SE curricula, which can focus more on the practical aspects of software development while also including topics such as software management, process, and project organization throughout the curriculum.

It is true that computer science—itself a relatively young field—has gradually expanded both its theoretical and scientific bases, and that this has caused an increase in theory content in many computer science curricula. However, physics and chemistry are examples of physical sciences, whereas software is a nonphysical entity. As such, software (on a small scale, which is useful in an educational setting) can be easily created and duplicated. So, the development and manipulation of software should continue to be a central theme in computer science curricula as well as in practice by computer professionals.

Myth 6

Software engineering graduates will not need further training to perform like experienced software engineers.

Typical new SE graduates find themselves working on teams where they are expected to perform (with little or no additional training) alongside experienced software engineers. Organizations often assume that new hires can internalize corporate culture and standards on their own and acquire domain expertise on the job.

The new hires often find themselves on a software project's critical path. One reason for this is that most corporate managers think their new hires know the latest and greatest methods and can handle more challenging assignments than some of the folks who have been around for a while. Another reason is that the new hire is often expected to put in many extra hours and not to have outside family obligations. A third reason is that many software projects start out with difficult schedules, and it is just not feasible to give new staff members the time to gradually ramp up the learning curve. In *Death March*, Ed Yourdon says, "To many grizzled veterans... every project is a death march project."[12] Tom DeMarco, in his book *Slack*, says: "…a dangerous corporate delusion: the idea that organizations are effective only to the extent that all their workers are totally and eternally busy."[13]

Consider the following example: A new hire joins a project in progress, replacing another employee who has been transferred to a different project. After a brief orientation, he or she inherits the other employee's work and is expected to perform to the existing schedule. After all, if the schedule for this particular software isn't met, the whole project will fall behind. Furthermore, the other team members are busy with their own work. Although they will answer an occasional question, they are quick to refer the new hire to documentation or Web resources.

New employees on new projects don't fare much better. Faced with a death-march-type plan, the new employee is given the same workload as experienced employees. Furthermore, new-hire salaries are sufficiently high that experienced employees are not that sympathetic to the new hire's plight, thinking to themselves, "When I started out,

Although formal education is crucial, a software engineer should also regularly (for example, every five years) prepare for and pass certification exams.

The education received on best engineering practices and techniques to support various software life-cycle activities will benefit a job candidate for a lifetime.

I got a fifth of what these new hires are getting, so they should pull their own weight."

Whatever the reason, new graduates are expected to perform on the same level as their experienced counterparts. Everyone up the line is under schedule pressure, and the idea of an apprenticeship period is a foreign concept in software development. The best that the new employee can hope for is a sympathetic, experienced mentor who will coach him or her along.

Prospective employees must look for those enlightened companies that can provide appropriate education and mentoring for their staff, and companies must recognize the need for apprenticeship and continuing education. The fact that books such as *Death March* and *Slack* exist and have a large audience suggests that this will not be an easy task.

Myth 7

Software engineering programs will correspond to specific corporate requirements.

All you have to do is to look at current job postings to see a list of specific languages and tools, such as C, C++, Ada, UML, Visual Studio, XML, and ASP, along with the disclaimer that no experience is required. To quote from some recent newspaper classified ads: "Must know C/C++," "Must be familiar with Accelerated SAP," and "Must have experience with object-oriented programming in Java or C++." Searching the job site www.monster.com, we found only one job description that talked about developing software requirements, designing, coding, and testing, along with using best engineering practices. We seem to be stuck in a time warp that emphasizes form over substance. There is no point in requiring experience in specific languages and tools—a software engineer can learn new languages and tools fairly readily and most will change in five years anyway. On the other hand, the education received on best engineering practices and techniques to support various software life-cycle activities will benefit a job candidate for a lifetime.

It would seem that many employers are still looking for programmers who can produce code in specific languages using specific tools in the short term. They're not looking for software engineers who can develop software using best engineering practices with a long-term view. This is because people in industry typically are looking for someone to start development work immediately—they do not have the time or interest to train their employees. Furthermore, companies expect their employees to leave in a year or two, so they assume they won't benefit from the longer term SE knowledge that the employee might have. The events of recent years have supported this attitude; SE staff expect to change jobs regularly, in some cases for a healthy salary increase. So, we have a vicious cycle: Software staff change jobs regularly because companies make little investment in employee retention, and companies make little investment in employee retention because software staff change their jobs regularly. In fact, the company that invests in employee training might find that the employee adds the newly acquired skill to his or her resume to find a job. Another contributing factor is that industry managers typically started as programmers, with no SE background, so that is their frame of reference. They want someone like themselves when they were starting out.

Companies also expect universities to use particular programming languages and tools in their curriculum, regardless of whether these languages and tools are the best vehicle to support the universities' education goals. If you query a software executive on his or her needs and how universities can help to meet them, that executive's first reaction is to list languages and tools. Only after some discussion does he or she move beyond this low-level litany to focus on the real education software engineers need.

How should we, as SE practitioners and educators, respond to these myths? First, remember that the field is still young, so we can expect to see diverse opinions on various issues. Moreover, education is not a panacea. We are not going to cause SE to become a mature field overnight by fielding relatively small numbers of SE degree programs.

We must foster stronger communication between diverse groups, such as various faculty groups, and between universities and industry. Myths tend to develop when there is little communication or when the communication that exists reflects our preconceived notions rather than objective assessment.

Universities and degree programs that have industry advisory boards report valuable exchanges of information through this mechanism. Faculty groups in different departments can also benefit from both informal and formal communications opportunities.

We tend to lose sight of the fact that there might not be a right or wrong approach to SE education. There is ample opportunity to experiment. We do not need to pigeonhole SE education into one model or another just yet. If we experiment and track our results, we will learn what works over time. It is actually good to have many different kinds of degree programs because they will allow the proper environment for experimentation and discovery.

So, maybe we just need to lighten up a little when we consider SE education degree programs, have fun developing and delivering these programs, and try to identify good educational models that work. At the same time, we need to figure out how to elicit industry feedback and incorporate it into degree programs in appropriate ways. Only then will SE reach the professional status it so richly deserves.

References

1. W. Gibbs, "Software's Chronic Crisis," *Scientific Am.*, vol. 271, no. 3, Sept. 1994, pp. 86–95.
2. H. Lieberman and C. Fry, "Will Software Ever Work?" *Comm. ACM*, vol. 44, no. 3, Mar 2001, pp. 122–124.
3. *Computing Curricula 2001*, ACM Special Interest Group on Computer Science Education, 2001, www.acm.org/sigs/sigcse/cc2001.
4. D. Bagert et al., *Guidelines for Software Engineering Education*, Version 1.0., tech. report CMU/SEI-99-TR-032, Software Eng. Inst., Carnegie Mellon Univ., Pittsburgh, Pa., 1999.
5. T. Lethbridge, "What Knowledge Is Important to a Software Professional?" *Computer*, vol. 33, no. 5, May 2000, pp. 44–50.
6. ABET, *ABET Criteria for Accrediting Computing Programs*, 2002, www.abet.org/criteria.html.
7. D. Parnas, "Software Engineering Programs Are Not Computer Science Programs," *IEEE Software*, vol. 16, no. 6, Nov./Dec. 1999, pp. 19–30.
8. M. Shaw, "Prospect for an Engineering Discipline of Software," *IEEE Software*, vol. 7, no. 1, Jan./Feb. 1990, pp. 15–24.
9. M. McCracken, "Software Engineering Education: What Academia Can Do," *IEEE Software*, vol. 14, no. 6, Nov./Dec. 1997, pp. 26–29.
10. M. Shaw, "Software Engineering Education: A Roadmap," *The Future of Software Engineering*, A. Finkelstein, ed., ACM Press, New York, 2000, pp. 371–380.
11. S. McConnell and L. Tripp, "Professional Software Engineering: Fact or Fiction?" *IEEE Software*, vol. 16, no. 6, Nov./Dec. 1999, pp. 13–17.
12. E. Yourdon, *Death March*, Prentice Hall, Upper Saddle River, N.J., 1997, p. 218.
13. T. DeMarco, *Slack*, Broadway Books, New York, 2001, p. 226.

About the Authors

Hossein Saiedian is a professor of software engineering and an associate chair in the Department of Electrical Engineering and Computer Science at the University of Kansas. His primary research area is software engineering—in particular, models for quality software development. He is also interested in SE education and training and cochaired the ICSE's Software Engineering Education track for 2000 and 2001 (and will cochair it again for 2003). He received his PhD in computer science from Kansas State University. He is a senior member of the IEEE and a member of the IEEE Computer Society and ACM. He is chair of the IEEE-CS TCSE's Committee on Software Engineering Education. Contact him at the Dept. of EECS, Univ. of Kansas, Lawrence, KS, 66045; saiedian@eecs.ku.edu.

Donald J. Bagert is a professor of computer science and software engineering at the Rose-Hulman Institute of Technology, where he is also the director of software engineering. His research interests include software process improvement, software tools for student advising, and software methodologies. He received a PhD in computer science from Texas A&M University. He is the steering committee chair for the IEEE Computer Society Conference on Software Engineering Education and Training, and the Professional Issues Editor for *FASE*, an electronic newsletter devoted to software engineering education, training, and professional issues. He is also a member of both the Educational Activities Board and the Professional Practices Committee for the IEEE Computer Society, and is a senior member of the IEEE. Contact him at the Dept. of Computer Science and Software Engineering, Campus Mail Box 97, Rose-Hulman Inst. Of Technology, 5500 Wabash Ave., Terre Haute, IN 47803; don.bagert@rose-hulman.edu.

Nancy R. Mead is the team leader for the Survivable Systems Engineering team as well as a senior member of the technical staff in the Networked Systems Survivability Program at the Software Engineering Institute. She is also a faculty member in the Master of Software Engineering and Master of Information Systems Management programs at Carnegie Mellon University. She received her PhD in mathematics from the Polytechnic Institute of New York, and a BA and an MS in mathematics from New York University. She is a senior member of the IEEE and IEEE Computer Society and is a member of the ACM. Contact her at the Software Engineering Inst., Carnegie Mellon Univ., Pittsburgh, PA 15213; nrm@sei.cmu.edu.