IEEE Software / Sounding Board
Jan-Feb 2013, 30(1):96

Putting the Engineering into Software Engineering Education

Jeff Offutt

I recently read a paper about software engineering research,[1] and once again discovered that its author, Lionel Brand, had published "my" ideas before I wrote them. Thankfully, his writing often stimulates further thinking, and this was no exception. His visionary thoughts on software engineering research started me thinking, but in terms of software engineering education rather than research.

Briand wrote about the "paradox of being both highly relevant and increasingly underfunded and discredited." Personally, I've found that software engineering research gets more respect every year, although the funding is abysmally low, at least in the US. This article argues that software engineering is not given enough relevance or support in higher education.

Research and Education

I've been a researcher in software engineering for more than 25 years, but I've also been an educator. I taught my first software engineering course as a graduate student in 1985, a standard undergraduate survey course. I joined my current university, George Mason, in 1992, partly because it had a full MS program in software engineering that was distinct from computer science. I've led this large successful program since 2003 and helped create software engineering concentrations in our PhD program (2000) and in our undergraduate applied computer science program (2010). Along the way, I've created over a dozen new software engineering courses, many of which had never been taught anywhere and had to be designed without adequate textbooks or other materials. This wealth of experience in software engineering education lets me see things differently from many of my colleagues.

Briand wrote a phrase that I've said many times: "Software engineering isn't a branch of computer science; it's an engineering discipline relying in part on computer science, in the same way that mechanical engineering relies on physics." Some of my colleagues respond by saying "well, of course," but many traditional CS professors think it's almost heretical. In this old-fashioned view, software engineering has always been part of computer science and always will be. But things change. This analogy has been 100 percent convincing to my colleagues in civil engineering, electrical engineering, and other traditional engineering disciplines.

Differentiating Software Engineering from Computer Science

If software engineering isn't a branch of computer science, then we must ask practicing software engineers what they need to know that they don't learn in CS degrees. Most computer science undergraduate students take one software engineering class, typically with a week or two spent on each phase in the traditional waterfall lifecycle model. A lot of the semester is devoted to process theory. It's worth noting that in modern software engineering projects, the traditional waterfall model is almost never used, and the ideas about process change constantly—often much faster than textbooks or instructors' knowledge. Moreover, as David Parnas asserts, process must be shown, not taught, by mentoring young engineers through actual projects.[2]

This answers what CS students learn about software engineering, but what useful knowledge are they not learning? It turns out that this is an easy question with lots of answers: usability, testing, security, design modeling, project management, quality control, standards, architecture, embedded applications, evolution, web applications, ethics, and so on. The list constantly changes. Yes, CS students learn a little

about some of these topics—for example, they spend about one week on testing in a semester-long class. But is a week really enough for an activity that consumes well over 50 percent of the effort in large software engineering projects?[3]

I like to ask my students a question: What is your expected job title when you graduate? If they know anything about the computing field, they don't say "computer scientist." Most answer correctly, "software engineer." Isn't it just a little strange that we prepare software engineers by teaching them computer science? This fact alone should obviously imply that we either need computer science programs to include more software engineering or more software engineering degrees. After all, nobody would argue that students study physics to become mechanical engineers!

Although both approaches would be an improvement, I believe that CS education shouldn't change to include more software engineering. Rather, I agree that they should continue to diverge.

Software Engineering Is Engineering, Not Science

Can you imagine mechanical engineering being part of physics? Can you imagine ME faculty performing research within the physics department? Can you imagine an ME program flourishing within the confines of a physics department? You probably can't, but that's where it originated.

Over a century ago, universities had departments and programs in physics and some taught really practical applications in physics, such as how to use principles from physics to build bridges, dams, cars, airplanes, and electrical circuits. Over time, physics fissioned into fields now known as mechanical engineering, civil engineering, electrical engineering, mining engineering, aerospace engineering, and a host of others. The process continues, but from our 21st century perspective, it's now obvious that engineering fields are different pedagogically and should be taught differently from physics. It's equally obvious that the hundreds of thousands of engineering students should take three or four courses in physics and even more in math. This is why my employer, George Mason University, has a successful and long-lasting MS program in software engineering and, more recently, a BS concentration in software engineering.

So how exactly does software education differ from computer science education? The most obvious is in topics; the list in the earlier paragraph is much of what we should teach in software engineering programs. Software engineering education also must focus on multiple quality attributes—not just efficiency, but reliability, scalability, security, availability, maintainability, and usability. Good engineering is also about making the right tradeoffs based on contextual requirements, which CS students aren't taught. Parnas emphasized that engineering students should learn applications and how to build complete products, as opposed to how to confirm known facts and extend knowledge.[2]

Another aspect that differentiates software engineering is that it needs to include non-computing topics other than CS. Software is part of larger systems, so we need systems engineering. Software is developed by teams, so we need project management. Software must be high quality, so we need statistical quality control. Deeper differences can affect not just *what* we teach but *how* we teach. Traditional CS courses emphasize single solutions, developed by students working alone, and evaluated primarily by efficiency. This approach doesn't work for engineering.

Three Principles for Teaching Software Engineering

Computer science, with its strong roots in mathematics, is usually taught using "convergent thinking," meaning problems have one answer and successful students should tend toward that answer. Engineering, however, especially software engineering, needs divergent thinking, where multiple answers are possible and the most successful students should find a solution that's unique when compared with other students' solutions. Divergent thinking is encouraged by assigning problems that have many solutions.

In CS, traditionalists become adamant supporters of "individual learning," discouraging cooperation at all costs. This is partly because instructors worry about plagiarism, not surprising since computers make it so very easy to copy. Thus a side effect of spending so much energy discouraging cheating is that we alienate the educational empowering benefits of collaboration and social processes. Practical software engineering is an extremely collaborative discipline, and I've found that software engineering students are

best taught with collaborative learning. Students should be encouraged to work together, to solve problems together, and to learn together. Instead of focusing exclusively on discouraging plagiarism, we should encourage students to learn more by learning together. Students learn more through collaboration than competition!

Computer science projects and homework assignments tend to be assessed on a uniform scale that measures every student's work with the same yardstick. But in engineering, especially software engineering, the notion of what will succeed often varies depending on the context, including users, market, platform, and release date. This suggests that we, as educators, should use differentiated assessments. Instead of every student trying to accumulate exactly the same points for the same requirements, we could offer a menu of potential features and attributes for students to choose from, each of which accumulates some number of points.

The Path Forward

Clearly, if software engineering is really the "best job,"[4] and employment is continuing to increase throughout the great recession with no end in sight, universities must shift from a computer "science" focus to a software engineering focus. Universities should create more undergraduate software engineering degrees. If that's not possible, undergraduate CS programs should add more software engineering—a one-semester course is clearly insufficient. Physics is still going strong, and society still needs physicists. But what's the ratio of engineers to physicists—100 to 1? 1,000 to 1?

Also, when we teach software engineering, we must remember that divergent thinking and collaborative learning are essential abilities for practicing engineers, and differentiated assessment is essential for teaching software engineering. I've successfully used all of these techniques in my classes, and so can you. As software engineering continues to move out of the shadow of CS to establish itself as a separate, independent discipline, industry will be more satisfied with our graduates, and companies will create more high-quality software. Don't we owe this to society?

Acknowledgments

I'm grateful to Lionel Briand, Rich LeBlanc, Paul Ammann, and Stephanie Offutt for helping me refine my thinking on this topic.

References

1. L. Brand, "Embracing the Engineering Side of Software Engineering," *IEEE Software*, vol. 29, no. 4, 2012, pp. 93–96.
2. D. Parnas, "Software Engineering Programs Are Not Computer Science Programs," *IEEE Software*, vol. 16, no. 6, 1999, pp. 19–30.
3. B. Boehm, *Software Engineering Economics*, Prentice-Hall, 1981.
4. Y. Klugerman, "Software Engineer Ranked Best Job for 2011," *Brain Track*, Jan. 2011; www.braintrack.com/college-and-work-news/articles/software-engineer-ranked-best-job-for-2011-11010502.

Jeff Offutt *is professor and director of the software engineering MS program at George Mason University. He was awarded the George Mason University Teaching Excellence Award, Teaching With Technology, in 2013, was named a GMU Outstanding Faculty member in 2008 and 2009, received the Best Teacher Award from GMU's Volgenau School of Engineering in 2003, and coauthored the textbook* Introduction to Software Testing *(Cambridge University Press, 2008). Offutt is co-editor in chief of Wiley's* Software Testing, Verification, and Reliability *and one of the founders of the IEEE International Conference on Software Testing, Verification, and Validation. Contact him at offutt@gmu.edu.*