CS Education in the U.S.: Heading in the Wrong Direction?

By Robert Dewar and Owen Astrachan

Last year, Edmond Schonberg and I published an article in *CrossTalk* (a U.S. Department of Defense software engineering journal) titled "Computer Science Education: Where Are the Software Engineers of Tomorrow?" in which we criticized the state of computer science education in U.S. universities [9]. The article caused quite a mini-storm of discussion and was picked up by Slashdot and also by *Datamation* in an article titled "Who Killed the Software Engineer? (Hint: It Happened in College)" [6].

In our *CrossTalk* article, we expressed the general concern that the computer science curriculum was being "dumbed down" at many universities, partly in an effort to bolster declining enrollments. The enrollment decline at many universities has been dramatic, and still has not shown much sign of recovery. The twin effects of the dot-com crash and the concern of outsourcing of IT jobs seem to have convinced many parents and students that IT is not a field with a future, despite studies that project a shortage of software engineers in the near future [5]. Perhaps the global economic meltdown will slow this cycle a bit, but I tend to agree that we will be facing a real shortage of well-trained software engineers in the future.

So obviously the question is what do I mean by a well-trained software engineer? To me, the critical need is the knowledge required to build large complex reliable systems. It is undeniable that our society depends in a critical manner on complex software. This is not only in the familiar areas of safety-critical software like avionics systems, but also in everyday financial systems. For example, consider the report from Moody stating a bug in the Moody computer system caused an incorrect AAA rating to be assigned to $1 billion worth of "constant proportion debt obligations" [4]. Now I do not know exactly what this means but it is surely one of the variety of peculiar economic instruments that have been factors in the current financial crisis: the credit ratings provided by agencies such as Moody are a critical element.

I frequently give talks on safety- and security-critical software, and whenever I give such a talk, I peruse the news the week before for stories on computer security failures. Prior to a talk last year, the high-profile stories receiving the most media attention included the break-in to vice presidential candidate Sarah Palin's email account and the successful hacking of the Large Hadron Collider Web site. Recently, one of my credit card companies reissued a card to me because a third-party database had been hacked (the credit card company would not identify the database).

I often encounter CS faculty members who take it for granted that all large computer systems are full of bugs and unreliable, and of course our experience with popular software such as Microsoft Windows reinforces this notion. The very use of the word "virus" is annoyingly misleading because it implies that really such infections are expected and impossible to eliminate, when in fact it is perfectly possible to design reliable operating systems that are immune to casual attacks. Early in

This article originally appeared in the "Viewpoints" section of Communications of the ACM 52*, 7, pp. 41-45. It is reprinted here with the authors' permission.*

the history of eBay, its auction software failed for nearly a week, and the company lost billions of dollars in capitalization. At the time I wrote to the founders of eBay that they had a company with a huge value depending on one relatively simple software application, and that there was no excuse for this application being other than entirely reliable. I commented that if their software people were telling them that such failures were inevitable, they should all be fired and replaced; I never received a reply.

So just what do we need to teach our students if they are to have the right viewpoint and skills to construct the complex reliable software systems of tomorrow, and to maintain, extend, and fix the systems in use today? In my experience, undergraduate computer science curricula simply do not regard complex software construction as a central skill to be taught. Introductory courses are dumbed down in an effort to make them fun and attractive, and have sacrificed rigor in designing and testing complex algorithms in favor of fiddling around with fun stuff such as fancy graphics. Most of these courses at this stage are using Java as a first language, and all too often Java is the only language that computer science graduates know well.

The original *CrossTalk* article was widely regarded as an anti-Java rant (one follow-up article was titled "Boffins Deride Java") [8]. It is indeed the case that the use of Java complicates basic education of programmers. It's not impossible to teach the fundamental principles using Java, but it's a difficult task. The trouble with Java is twofold. First it hides far too much, and there is far too much magic. Students using fancy visual integrated development environments working with Java end up with no idea of the fundamental structures that underlie what they are doing. Second, the gigantic libraries of Java are a seductive distraction at this level. You can indeed put together impressive fun programs just by stringing together library calls, but this is an exercise with dubious educational value. It has even been argued that it is useless to teach algorithms these days. It's as though we decided that since no one needs to know anything about how cars work, there is no point in teaching anyone the underlying engineering principles. It is vitally important that students end up knowing a variety of programming languages well and knowledge of Java libraries is not in itself sufficient.

Although the article was regarded as being anti-Java that misses the main point, which is that the curriculum lacks fundamental components that are essential in the construction of large systems. The notions of formal specification, requirements engineering, systematic testing, formal proofs of correctness, structural modeling, and so forth are typically barely present in most curricula, and indeed most faculty members are not familiar with these topics, which are not seen as mainstream. For an interesting take on the importance of a practical view, see Jeff Atwood's column discussing the need to teach deployment and related practical subjects [1].

Another area of concern is that the mathematics requirements for many CS degrees have been reduced to a bare minimum. An interesting data point can be found in the construction of the iFacts system [7], a ground-based air-traffic control system for the U.K. that is being programmed from scratch using SPARK-Ada and formal specification and proof of correctness techniques [2]. It has not been easy to find programmers with the kind of mathematical skills needed to deal with formal reasoning. And yet, such formal reasoning will become an increasingly important part of software construction. As an example, consider that of the seven EAL levels of the Common Criteria for security-critical software, the top three require some level of formal reasoning to be employed [3].

It is true that a lot of software development is done under conditions where reliability is not seen as critical, and the software is relatively simple and not considered as safety- or security-critical. However, if this is all we train students for then we won't have the people we need to build large complex critical systems, and furthermore this kind of simple programming is exactly the kind of job that can be successfully transferred to countries with less expensive labor costs. We are falling into a trap of training our students for outsourceable jobs.

The original article in *CrossTalk* was based on our observations as faculty members and as software company entrepreneurs, rather than on a carefully researched study. When several people asked us for data to back up our claims, we had none to offer. Since then, however, it has been very interesting to read the flood of email we received in response to this article. In hundreds of messages, we did not get anyone saying "what are you talking about? We have no trouble hiring knowledgeable students!" On the contrary, we got hundreds of messages that said "Thank you for pointing out this problem, we find it impossible to hire competent students." One person related an experience where he had a dump from a customer for a program that had blown up and was sifting through it trying to determine what was causing the problem. A newly hired student asked him what he was doing, and he said that he was disassembling the hex into assembly language to figure out the problem. The student, who had always considered himself superior because of his computer science degree, replied "Oh yes, assembly language, I've heard of that," and was amazed that the senior programmer (whose degree was in music) could in fact figure out the problem this way.

Another company noted that it had found it a complete waste of time to even interview graduates from U.S. universities, so they added at the end of the job description the sentence "This work will not involve Web applications or the use of Java," and that had served to almost completely eliminate U.S. applicants. Here was a case of domestic outsourcing where they were looking for people in the U.S. who had been trained in Europe and elsewhere and were better educated in the fundamentals of software engineering. These are just two examples of many similar responses, so it is clear that we have hit on a problem here that is perceived by many to be a serious one.

Biography

*Robert Dewar (**dewar@adacore.com**) is a professor emeritus of computer science at the Courant Institute of New York University and is cofounder, president, and CEO of AdaCore.*

References

1. Atwood, J. 2008. How should we teach computer science? Coding horror. **http://www.codinghorror.com/blog/archives/001035.html**.

2. Barnes, J. 2003. *High Integrity Software—The SPARK Approach to Safety and Security*. Addison-Wesley.

3. Common Criteria. 2006. Common criteria for information technology security evaluation, Version 3.1. **http://www.commoncriteriaportal.org**.

4. Farrell, N. 2008. Boffins deride Java. *The Inquirer*. **http://www.theinquirer.net/gb/inquirer/news/2008/01/08/boffins-deride-java**.

5. Maloney, P. and Leon, M. 2007. The state of the national security space workforce. **http://www.aero.org/publications/crosslink/spring2007/01.html**.

6. McGuire, J. 2008. Who killed the software engineer? (Hint: It happened in college.) *Datamation*. **http://itmanagement.earthweb.com/career/article.php/3722876**.

7. National Air Traffic Services. NATS pioneers biggest ATC advance since radar. **http://www.nats.co.uk/article/218/62/nats_pioneers_biggest_atc_advance_since_radar.html**.

8. Oates, J. 2008. Moody's to fix sub-prime computer error. *The Register*. **http://www.theregister.co.uk/2008/07/03/moodys_computer_bug**.

9. USAF Software Technology Support Center (STSC). 2008. Computer science education: Where are the software engineers of tomorrow? *CrossTalk*. **http://www.stsc.hill.af.mil/CrossTalk/2008/01/0801DewarSchonberg.html**.

Counterpoint: Owen Astrachan

Robert Dewar has graciously shouldered the task of castigating the language commonly used in introductory programming courses. Dewar, like Edsger Dijkstra [13] and others before him, holds the language at least partially responsible for, and decries the state of, computer science curricula; he then attempts to use the programming language as a lever to move curricula in a particular direction. However, the lever of the introductory programming language is neither long enough nor strong enough to move or be responsible for our curricula. Attempts to use it as such can generate discussion, but often more heat than light. The discussion is often embroiled in fear, uncertainty, and doubt (aka FUD) rather than focused on more profound issues.

There are definite elements of FUD in the arguments offered by Dewar just as there have been by his predecessors in making similar arguments. Whereas Dijkstra lamented "the college pretending that learning BASIC suffices or at least helps, whereas the teaching of BASIC should be rated as a criminal offense: it mutilates the mind beyond recovery" we see Dewar noting that "It's not impossible to teach the fundamental principles using Java, but it's a difficult task." Dewar and Dijkstra perhaps would like us to return to the glorious days of text editors and punch cards rather than "fancy visual IDEs." However, the slippery slope of assumption that the new generation just doesn't get it leads to the Sisyphean task of pushing the pebble of language, be it BASIC or Java, uphill against the landslide of boulders that represents the reality of computer science. This is the case regardless of whether we're in Dijkstra's world of 25 years ago, the world of 2009, or the Skynet world of tomorrow—which is probably closer than we think.

I don't mean to suggest that Dewar and Dijkstra are arguing for the same thing. Dewar would like computer science programs to produce well-trained software engineers who can build large complex reliable systems. Dijkstra excoriated software engineering at every opportunity fixing as its charter the phrase "how to program if you cannot." Both miss part of the bigger picture in the same way that Stephen Andriole missed it in the July 2008 *Communications* Point/Counterpoint "Technology Curriculum for the Early 21st Century" [10]. In his Counterpoint, Eric Roberts points out the flaw of "generalizing observations derived from one part of the field to the entire discipline." Computer science programs must embrace a far wider audience than software engineers building secure systems. Many top programs are housed in schools of Arts and Sciences rather than in Engineering, many have chosen not to be accredited by CSAB/ABET Students may choose computer science as a stepping-stone to law, medicine, philosophy, or teaching rather than as a foundation for becoming a programmer or software engineer. Schools like Georgia Tech are developing innovative programs to address the different needs of diverse audiences: students looking to computer science as the basis for visual studies or biology rather than preparing them for a software-oriented career. There is no one-size-fits-all solution to addressing the skills and knowledge needed to succeed in these areas. Should we expect Craig Venter or Gene Myers to ask computer science programs to include more computational biology because the demand for bioinformaticians exceeds supply? Will we be surprised if Ken Perlin asks for programs to embrace games and graphics more than they do to ensure a steady supply of people interested in animation or computer-generated imagery? We are discussing the requirements and curricula of an undergraduate degree! Our programs can certainly build a superb foundation on which students can continue to gain knowledge and skills as they work and study in different areas, but we should no more expect students to be expert or even journeymen than we expect our premed students to be able to remove an appendix after four years of undergraduate study.

As Fred Brooks reminded us more than 20 years ago, there is no silver bullet that will solve the problems endemic to software development nor is there a panacea to cure the ills that may plague computer science curricula and programs [11]. Studying more mathematics will not make software bugs disappear, although both Dijkstra and Dewar seem to think so. Dewar points out the need for "formal specification and proof of correctness techniques" as foundational for software development using Ada. Dijkstra tells us "where to locate computing science on the world map of intellectual disciplines: in the direction of formal mathematics and applied logic," but pines for Algol rather than Ada. Both miss Brooks' point about the essential complexity of building software, the *essence* in the nature of software. In a wonderful treatise that has more than stood the passage of 20 years and in which he presciently anticipated the tenets of Agile software methodologies, Brooks claims that "building software will always be hard," and that this essence will not yield dramatic improvements to new languages, methodologies, or techniques.

Brooks has hopes that the essential aspects and difficulties of software may be improved by growing software rather than building it, by buying software rather than constructing it, and by identifying and developing great designers. He differentiates between essential and accidental aspects of software where accidental is akin to incidental rather than happenstance. Changing programming languages, using MapReduce or multicore chips, and employing a visual IDE in introductory courses address these accidental or incidental parts of software development, but these don't mitigate the essential problems in developing software nor in educating our students. As Brooks notes, addressing these accidental aspects is important—high-level languages offer dramatic improvements over assembly-language programming both for software design and for introductory programming courses.

Brooks' view, which I share, calls for "Hitching our research to someone else's driving problems, and solving those problems on the owners' terms, [which] leads us to richer computer science research" [12]. I will return to problem-driven approaches later.

It would seem from the juxtaposition of amusing anecdotes regarding flawed software systems that Dewar would like to make the academic community and the larger computer science and software communities aware that a simple change in attitude and programming language in our colleges and curricula will help make the world more secure and safe with respect to the reliable systems on which it depends. Although software runs on computers it produces outputs and effects that transcend computers. It was not a simple bug in Moody's computer system that caused constant proportion debt obligations to be incorrectly assigned the AAA rating. The model that Moody used was likely incorrectly parameterized. Even if the flaw was related to code, rather than to a model, Moody's correction of the model did not lead to a change in the AAA rating as it should have because of larger and more deeply entrenched financial and political concerns. Standard and Poor's model also assigned the AAA rating to the same constant proportion debt obligations. Both services eventually lowered their ratings, but arguably these actions were insufficient.

Blaming the current economic crisis even in part on software errors is more than a stretch. Similarly, Dewar notes that U.S. vice presidential nominee Sarah Palin's email account was compromised and that a Web site was hacked, implying these are security failures that might be fixed if only we didn't use Java in our introductory courses. Because Governor Palin used Yahoo mail for what appears to be at least semiofficial business, her password recovery mechanisms were based on publicly available information such as her birthday, and her hacked email was posted on 4chan and Wikileaks: this is a case study in social engineering rather than one in secure systems.

Dewar's claim that Java is part of a "dumbing down" of our curricula has been echoed in other venues, notably by Joel Spolsky [15] and Bjarne Stroustrup [14]. However, Stroustrup notes that it isn't the language that's a problem—it is attitude. He says, and I agree that: "Education should prepare people to face new challenges; that's what makes education different from training. In computing, that means knowing your basic algorithms, data structures, system issues, etc., and the languages needed to apply that knowledge. It also means having the high-level skills to analyze a system and to experiment with alternative solutions to problems. Going beyond the simple library-user level of programming is especially important when we consider the need to build new industries, rather than just improving older ones."

These articles, like Dewar's, associate Java with a "dumbing down" of curricula. Spolsky specifically mentions the school at which I teach as one of the new *JavaSchools*. He laments that our students are lucky in that: "The lucky kids of JavaSchools are never going to get weird segfaults trying to implement pointer-based hash tables. They're never going to go stark, raving mad trying to pack things into bits."

We didn't become a JavaSchool because we wanted to avoid segfaults, pointers, and bits. We use the same assignments and have the same attitude we did when we used C++. We switched from C++ for well-founded pedagogical reasons: Java is a better teaching language for the approach we were using than C++. Note that I'm not claiming Java is the best language for every program, but we spend much more time in our courses dealing with the Brooksian essence of programming, algorithms, and software using Java rather than with the accidental aspects symbolized by the kind of cryptic error messages that result from misusing the STL in C++. Our switch to Java was grounded neither in perceived demands from industry nor in an attempt to attract majors to our program, but in working to ensure that our beginning courses were grounded in the essence of software and algorithms.

We must work to ensure we attract motivated and capable students, not because it is incumbent on us as faculty to train the next generation of software engineers, but because it is our responsibility as educators and faculty to encourage passion and to nurture and increase the amazing opportunities that computing is bringing to our world. It is highly likely that some programming languages are better for teaching, others are better for Ajax applications, and the right flavor of Linux makes a difference. But we shortchange our students and ourselves if we live at the level of what brand of brace and bit or drill is best for a carpenter. Instead, we should look for problems that motivate the study of computing, problems that require computation in their solution.

Just as we cannot escape the essential complexity and difficulty of developing software we cannot escape the essence of undergraduate education. We each bear the burden of our past experiences in constructing models for education. In my case this is the grounding of computer science as a liberal art, since my education began in that realm. For others, computer science is clearly an engineering discipline and to others still it is a science akin to biology or physics. We don't need to look for which of these is the correct view; they are all part of our discipline. The sooner we accept differing views as part of the whole, rather than insisting that our personally grounded view is the way to look at the world, the sooner we will make progress in crafting our curricula to meet the demands and dreams of our students.

Biography

*Owen Astrachan (**ola@cs.duke.edu**) is professor of the practice of computer science at Duke University and the department's director of undergraduate studies for teaching and learning.*

References

1. Andriole, S. J. and Roberts, E. 2008. Technology curriculum for the early 21st century. *Comm. ACM 51*, 7. 27-32.

2. Brooks, F. 1987. No silver bullet: Essence and accidents of software engineering. *IEEE Computer 20*, 4. 10-19. (Reprinted in *The Mythical Man-Month: Essays on Software Engineering*, Anniversary Edition. Addison-Wesley, 1995.)

3. Brooks, F. 1996. The computer scientist as toolsmith II. *Comm. ACM 39*, 3. 61-68.

4. Dijkstra, E. 1984 Keynote address at ACM South Central Regional Conference. **http://www.cs.utexas.edu/users/EWD/transcriptions/EWD08xx/EWD898.html**.

5. Maguire, J. 2008. Bjarne Stroustrup on educating software developers. *Datamation*. **http://itmanagement.earthweb.com/features/article.php/3789981/**.

6. Spolsky, J. 2005. The perils of JavaSchools. *Joel on Software*. **http://www.joelonsoftware.com/articles/ThePerilsofJavaSchools.html**.