

How should computing education evolve in the next ten years? In numerous ways, with numerous new ideas attempted, given serious scrutiny, and the results of such scrutiny disseminated for the common good. The worst that could happen to computing education is for it to remain exactly as it is now; while there is much that is good in it, there is room for improvement. Failure rates in introductory computing courses are still disappointingly high; diversity in computing fields seems to be getting worse (even while it seems to be improving in other NSF STEM (Science, Technology, Engineering, and Mathematics) areas; and enrollments in computing majors are still suffering in the wake of the dot-com bust combined with continuing pessimistic reports about outsourcing of technology-related jobs.

Yet the scope of computing fields is broadening rather than contracting, and more people than ever need to have some familiarity with computing. Consider such areas as bioinformatics, multimedia, computer gaming, human-computer interaction, computer security, and e-commerce. People are needed to customize existing applications as well as to write new systems from scratch; security and privacy are being recognized as more vital than ever before; and a lowly home computer can be harnessed to attack other computers without its owner's knowledge, if that owner does not know how to prevent it. There are exciting opportunities for expanding computing education if we are willing to take some risks and perhaps challenge our assumptions about how things "ought" to be done.

There are serious challenges to improving the preparation of undergraduates for computing careers. On the student side, too many students have poor mathematical backgrounds, and too many have trouble with reading and writing. Many students have to work in outside jobs while working on an undergraduate degree, which sometimes means they are unwilling or unable to give the time to coursework that is needed. Many such students seem to operate in "triage" mode with regard to their coursework: they do what they think "has" to be done to get by in a course, only when they feel it must be done; approaches to teaching that depend on their doing additional work on their own that isn't "required" will, I fear, not produce good outcomes for such students.

What are the challenges on the instructor side? Time is the biggest challenge. The student challenges mentioned require new approaches and experimentation, but heavy teaching loads and increasing responsibilities (and expectations) for instructors make these difficult. Those of us at public institutions are facing tighter budgets, and side-effects such as fewer course offerings and more students per course exacerbate the situation. And when instructors are time-challenged, they are sometimes too burnt-out to be willing to entertain new ideas about teaching and curriculum.

Addressing some of the challenges on the student side would require system-wide changes -- for example, better K-12 education so that students are better prepared in the foundation areas of mathematics, reading, and writing, and better financial aid so that students don't have to try to work full-time while trying to attend school full-time. These, of course, would help students in all fields, not just computing. In terms of computing students, better programming environments for novices, especially with multiple language levels, can take away some of the "accidental" barriers that make learning introductory problem-solving and programming concepts harder than they need to be. Introductory courses that stress problem-solving over syntax details could make programming more accessible to more students (and also be beneficial to those not intending to major in computing). Given that the computing field is broadening, making computing more accessible at the introductory level might encourage more students to consider these new areas.

How might we improve student success after the introductory programming course? Computer applications are becoming ever larger and more complex; aspects of project management could be integrated through more courses rather than focused in a single project management or software engineering course. Another way to approach complexity is to bring some of the ideas of extreme programming and other agile programming methodologies to education. While extreme programming is not suitable for the largest projects, it is suited to medium-sized projects, and some of the techniques are accessible and good first steps toward more heavy-duty software engineering. For students to be competitive in a global workforce, they need to be flexible. Not being afraid to learn another language, not being afraid to apply different project management techniques to different projects; these traits will be necessary for a would-be worker's continued success.

What kind of extreme programming techniques do I mean? Pair-programming, for one -- if this also encourages more kinds of people to find computing attractive, so much the better. Unit-testing, writing tests for one's code before writing that code, is the other that quickly comes to mind. Writing tests first ties in nicely with design recipes as in the "How to Design Programs" and "How to Design Chapter Hierarchies" projects (<http://www.htdp.org/>, and <http://www.ccs.neu.edu/home/vkp/HtDCH>) -- and these design recipes emphasize problem solving and a logical approach to designing programs.

What can be done to address the challenges on the instructor side? Release time for trying new educational approaches would help; if tenure-track faculty could also be assured that such experimentation would not hurt them in the tenure process, that could encourage them to innovate as well. Imagine if, for each computing department, at least one instructor per area -- one

willing to try new things, or two for better continuity and home persuasion -- received travel and release-time support to attend workshops and training sessions in state-of-the-art educational techniques for that area. As an example, consider the TeachScheme! workshops (<http://www.teach-scheme.org>) -- these can have a tremendous impact, if enough instructors are given the chance to experience them.

What kinds of areas in computing need such support? Areas most in need of innovation, and "new" areas, and those that need to be taught more broadly than they currently are -- some examples could include introductory computing, to make it more effective and more accessible; computer security, because of its importance to the economy and to national security; computer interface design and human-computer interaction, because designing usable programming interfaces is more important than ever; software engineering, because of the never-ending, always-mutating "software crisis"; and technical writing, because the state of communication skills amongst those who work in computing is not where it needs to be.

Instructors could receive release time to try new teaching techniques in courses in their home institutions. They could also receive release time for weekend courses, which could be on-campus or elsewhere -- local high schools, malls, etc. -- for interested members of the general public on topics that would raise the public's knowledge of certain issues or have local strategic importance: for example, how to avoid the spread of computer viruses and worms, how to avoid Internet fraud, simple introductions to programming, and use of application software determined to be in demand in that particular area. Some of these could deliberately be designed to help computing workers to become more current in topics relevant either to their jobs or to tasks they may need to be able to do in the future in order to be attractive to potential employers. But I don't want to imply that such sessions have to be primarily organized or staffed by local universities or community colleges (although they could be); local computing users groups of various kinds, such as Linux Users Groups or local consortiums of those interested in technology, could sponsor such sessions as well. The more stakeholders involved, the better.

In five years, I would like to see different universities using different languages -- but all initially using environments geared toward novices. The ideal model for computing education would include students learning multiple languages in their undergraduate years as the normal expectation -- with better ideas for instructors about how to ease the transition. The model would assume use of open-source software that can run on all platforms, and students comfortable using multiple operating systems. Different universities would have different specialties and different emphases, but most would stress problem solving first (and language arcana second). Appropriate writing would be stressed throughout the curriculum, as would unit-testing. Pair-programming would be used in closed labs in the early courses, and later courses would involve team projects.

Different courses and concepts within computing will thrive best with different approaches, so the way to improve these is to attack each core area individually. But here is one thought: is the common practice of "killer" junior-level course projects pedagogically sound? Do large monolithic assignments drive too many students to either throw up their hands in despair and fail, or hire someone from an Internet programming service? How do we challenge students without overwhelming them, and how do we convince them that doing the work we ask them to do is worth it for their education?

Fred Brooks has said that a large programming task should be broken up into milestones so sharp that a programmer cannot deceive himself or herself whether they have been met [Brooks, [The Mythical Man-Month](#)] -- we do not set students on that path with large monolithic assignments. We could start out with large projects that instructors have broken into such milestones in, say, the junior-level courses, and then, by the senior-level capstone, have students come up with the milestones (that the instructors then critique). This would serve the dual purposes of giving students valuable experience as well as increasing the chances that they actually complete large assignments. It can be more work for the instructor -- the milestones should be looked over, after all -- but when an instructor sees the work along the way, it might make it more difficult for a student to buy and submit an entire assignment right at the end. And if one points out to the student at the end just how much he or she has accomplished, it will likely amaze them (and build their confidence for the next project).

Note that breaking large course tasks into reasonable milestones does not imply that the nature of those large course tasks has to be homogeneous -- these can be designed to suit the topics involved, or to complement the projects in other parts of the curriculum. For example, students need the experience both of building things from scratch and of working within existing code -- the idea of breaking a large task into milestones fits within either of these, and ideally students should experience both before they graduate with an undergraduate degree. Projects can involve real clients or not, involve teams or not, even be hardware-based, software-based, or some combination. And don't we all agree at this point that a senior capstone team project experience is part of the ideal computing education? Many students do not like working in teams -- and some instructors do not enjoy dealing with student teams -- but it is an important experience for students to have, for many reasons well-covered in the literature.

Those designing strategies to improve computing education should come from a broad swath -- industry (including large companies and small), academia (including research universities, teaching-oriented institutions, community colleges, and high schools), and professional societies. The government's ideal role would be to support the endeavor, and then back off -- fund meetings and dissemination, but not try to push the discussion.