The Three Bears Problem

Richard P. Gabriel

We have a real problem with computer science education: We do not educate to maximize the effectiveness of software design talent to produce master designers and implementers; and we have not yet climbed out of the primordial understanding-of-computing soup but have almost stopped the research that would find us the way out. Getting us on track requires changing dramatically how we educate undergraduates, but the changes won't be limited to something as simple as rethinking the role of programming language design, implementation, and application in undergraduate computer science education.

The Problem with Practitioner Education

This problem is a little easier to understand than the problem with researchers, so I'll start with it.

One of the most valued people on a software project is the architect/designer, the person who produces the plan for project (whether with agility or not) and is responsible for making the project a success by guiding it and imbuing it with a maintainable structure and style. Using the term Fred Brooks uses: the architect/designer provides the *conceptual integrity* for the design, which goes a long way toward insuring the project's success. Neither our undergraduate nor our graduate educational approach takes the training of such a person seriously. Centuries of experience with other, similar disciplines—painting, sculpture, poetry, and even architecture itself—have shown that the best approach to getting good at design is to practice while reflecting. This is a mentor-apprentice way of learning, based on learning the proper application and use of what I'll call *craft elements* for the discipline/art, studying the use of those elements by masters, and producing a real portfolio while under the guidance of a master.

Computer science is thought of either as a discipline akin to mathematics or as engineering, and so we educate our practitioners the way the guy is who figures out what girders to use in a highrise and not the way the architect is who designs the entire highrise. And in fact, most companies who hire undergraduates right out of school either put them through a watered-down mentoring process for a number of years before relying on them for design or else depend on out-of-the-classroom experience as their guide to talent level. In fact, most of our best software-related designers are autodidacts when it comes to design.

The Problem With Researcher Education

When a discipline supports an undergraduate major, there is, typically, much well-settled knowledge about it. In general, undergraduate programs for something at least as broad as CS appear quite a long time into the development of the discipline. In our case, the pressure from industry to provide trained workers has forced us to teach CS to undergraduates before the time was ripe. When a subject is taught as an undergraduate major, students believe the knowledge taught is not debatable—and that's our problem.

My worry comes from some specific things I've seen. The first is that "big idea" research has nearly come to a halt. My observation is that the slowdown began toward the end of the 1980s. I was not the only one to notice this: in 2004, then-President of ACM, Dave Patterson wrote the following in his President's Letter, "The Health of Research Conferences and the Dearth of Big Idea Papers":

> *Calls for papers often include encouraging words for big idea or new direction papers. The problem is that reviewers see so many regular papers it is just too difficult to switch gears and be more understanding when evaluating bolder papers with holes in arguments or missing measurements.*
>
> —Communications of the ACM December 2004 / Vol. 47, No. 12, page 23

Second, I believe that recent work in understanding the nature of errors in systems, of how to keep systems running robustly, of how to program parallel (including multicore) machines, of ultra large scale systems, and even what are the underlying foundations of computation and programming—to name some examples—requires us to start producing those big ideas pretty soon.

Third, the slowdown in big-idea papers—coincidentally or not—began about a generation after the first undergraduate programs began for CS—just about when you'd expect the first wave of faculty who majored in CS to start publishing, and the slowdown advanced at about the pace that the faculties and population of journal editors and program committee members filled up with academics holding undergraduate CS degrees.

Worse, we are required to educate, in addition to future generations of researchers, the computing grease monkeys of the future—the practitioners I talked about—at the same time. Not to disparage them—we need people who can competently design, build, and maintain the software systems in the future—but these folks will not be called upon to figure out, for example, how to solve the API and other versioning problems for systems in excess of a billion software and hardware components nor how to program effectively swarms of possibly pseudo-computing components to repair damage to an infected system (or even to figure out what I mean by this).

Practitioners Versus Researchers

At the same time: The impulse to educate practitioners had the effect of creating an educational approach that attempts to educate both the practitioner and future researcher using roughly the same courses (though there are courses that distinguish between the two groups) and other material. To properly educate a researcher, topics in programming languages, for example, should take a critical and perhaps even historical approach so that it is clear that the current state of the art is simply a point along a (possibly evolutionary, if we're lucky) path, while for the practitioner, teaching programming languages should be (substantially) aimed at creating a thorough competence at designing for and programming in those languages. (This requires, I believe, also being knowledgeable about the principles behind the mechanisms, if not expert.) Because we don't aim at either extreme of student (practitioner versus researcher), we are serving neither particularly well.

An aspiring researcher should emerge from an undergraduate education with the sense that there is much unfinished business, and that perhaps even some of the seemingly finished business is actually provisional. An aspiring practitioner should feel at home with and have great confidence in the taught tools, and should be equipped to learn new ones with a certain ease. Let's compare this to a related discipline: The mathematics major—at least when I was an undergraduate—was broken into applied and pure math. Someone in applied math was interested in using mathematics at an expert level for some purpose other than mathematics itself. Someone in pure math was interested in how mathematics proceeds with an aim toward working on hard problems after an advanced degree. An applied math major practices (as an undergraduate) by using mathematics while a pure math major practices by proving theorems. Many of the courses were the same for both types of math major, but the pure math guys proved a lot more theorems, and the applied math guys solved more practical problems using math.

By taking the middle course in CS—by trying to train practitioners and researchers the same way—we achieve mediocre results for all.

Educating Practitioners

When it comes to educating practitioners, I believe we do an especially poor job. We are fond of thinking of our discipline as engineering, but the sad truth is that we are not very engineering-like. Having a viable engineering discipline requires a lot more science and practical experience than we have in computing. However, for now we can pretend. Software engineering at this stage is really more of an exercise in metaphor making than actual engineering. Programming—some don't like this word—(and the designing that goes with it) really is not exactly like any single other thing people do, but is an amalgam. Or to put it another way, many other human activities can be viewed more effectively as a form of programming than turning the metaphor around. Designing and programming are not merely about producing effective artifacts, but in order to make the result maintainable, the artifacts need to communicate with other people, perhaps long after the original designers and programmers have passed on or even away. There is a degree of artistry to it. Programming—because the eventual requirements cannot all be gathered beforehand—is more akin to discovery than to engineering or manufacturing.

For these reasons I believe we need to think of the creation of software as a craft for the medium term. We can still use the word "engineering" to talk about it, if people insist, but we will serve our students best if we take teaching techniques from the toolbox of the crafts and fine arts. In saying this I speak with some experience. In addition to my PhD in Computer Science from Stanford, I have a Master of Fine Arts in Creative Writing (Poetry) from one of the better schools for that sort of thing: Warren Wilson College.* A fine-arts degree is earned by doing a couple of things:

- creating stuff under the supervision of a mentor; usually there are several mentors over the course of the program (I had 4)
- revising under supervision after receiving criticism
- engaging often in a workshop-type process in which other students and a teacher or two critique work using a mostly formal process
- reading or studying critically the work of masters in whatever fine art it is
- writing short essays on the use of craft elements in the fine art
- taking courses and seminars
- producing a dissertation which is focused narrowly on a craft-related topic
- producing a manuscript or portfolio; this typically must be judged adequate by both a faculty advisor, a panel of faculty, and an outside artist

These techniques have been tried for teaching software design in a test semester at the University of Illinois with Brian Marick, Ralph Johnson, and me serving as faculty. Teaching programming languages, design, and implementation can be done this way, and probably much more effectively than using the usual methods. Mostly, better success is achieved because the focus of the MFA approach is to *do* under supervision while *reflecting* on what you're doing. This is a form of apprenticeship.

Today the number of designs and amount of software produced in an undergraduate or masters level program in CS is, to put it bluntly, ridiculously small. Training practitioners is one of the important things we do in the universities, and the approach we take verges on being immoral. I believe we should learn from fine-arts education and use some of their techniques to teach practitioners. A proposal I wrote for this years ago (and which formed the basis of the Illinois experiment) can be found here: `http://dream-`

* I should point out that of the two degree programs, the MFA was by far more rigorous and required more discipline and hard work than the PhD.

songs.com/MFASoftware.html. Also, a group of educators and I wrote up a description of an entire CS undergraduate education based on a new *bottega* (or studio) model and submitted it to OOPSLA—I can supply a copy if there is interest. [1]

Lest you believe that I am too hard on CS regarding engineering, consider that Filippo Brunelleschi, who built the dome for Santa Maria del Fiore (Il Duomo in Florence) and is considered one of the great mechanical geniuses and architects of the Renaissance apprenticed as a goldsmith—and this was several if not many millennia into the development of building as an engineering discipline.

Educating Researchers

Aspiring researchers deserve to be taught in a way that prepares them both temperamentally and in terms of knowledge to be good at it. Mathematics and the hard sciences do a good job of this. Part of the reason is that these other disciplines can teach the material in layers or stages that reflect or recapitulate the evolution of the discipline. In mathematics we begin with simple algebra and trigonometry, progress to calculus, topology, group theory, and more advanced algebras, and finally make our way through whatever subdiscipline is the focus of research interest. Here the material spans the work of mathematicians over thousands of years, and so the accumulation of ideas, as well as techniques, actually reveals the course of research history. The same is true of the other hard sciences.

We(in CS)'ve got a relatively simple set of mathematical foundations reaching back to computability and decidability theory from the early 20th century, and we culminate more or less in design principles for programming languages and a mathematically simple set of results in type theory. Naturally we have compilers, a bunch of programming languages, data structuring, AI, complexity, and a host of other topics we can cover, but the point is that we have only about 100 years of research history to draw on, and most of the most relevant parts of that was aimed at getting absurdly underpowered computers to do something/anything useful. And the way we teach the material is to teach a programming language first with some exercises, and then move on to the more theoretical things in a more or less monolithic group or single layer. We rarely teach historically, which would be another way to prepare researchers. The truth is that we teach aspiring researchers as if they were simply smarter and more ambitious practitioners (and we hardly train the practitioners at all).

Summing Up

So here's what I think. When designing how to teach undergraduates programming language stuff, we need to think about these things:

- how should the education of practitioners and researchers differ?
- should we introduce a bifurcation into Applied CS and Pure CS?
- what can we learn about how to teach practitioners from disciplines that produce master practitioners but lack a deep and extensive science—and perhaps also lack a practical knowledge base? (that is, until we can truly be an engineering discipline)
- what can we learn about how to teach researchers from disciplines that do that well, such as mathematics, physics, and biology?

Once we've done that, we can probably come up with good ideas on what to teach undergraduates about programming languages and how to teach it.†

References‡

[1] David West, Pamela M. Rostal, Eugene Wallingford, Joseph Bergin, Robert C. Duvall, Rick Mercer, Richard P. Gabriel, *The Road: Reinventing Education*, submitted to Onward! OOPSLA 2008, October 2008.

† Based on early experiments with the bottega-style approach, I am hopeful it will be well suited to a variety of learning styles, and hence will be more congenial to women and under-represented groups.

‡ This white paper is written assuming it will not be published; its content overlaps with that of the paper submitted to Onward!.