

ICER White Paper

David Notkin
Computer Science & Engineering
University of Washington
Seattle WA 98195-2350
notkin@cs.washington.edu

December 2005

Computing is a field of demonstrable importance, of rapid change, and of ever-increasing breadth. All three of these characteristics make the question of how to educate future participants in the field especially challenging. The importance of the field leads to extreme pressure to meet industrial – and societal – needs. The rapidity of change in computing leads to the core question of how to provide a stable base of material while ensuring that it remains useful and appropriate. The increasing breadth of the field leads to the need to interact with people with legitimate and significant computing requirements, but with a different base of knowledge, a different set of motivations, goals and reward structures, and different terminologies.

Given these (and surely other) pressures, what kind of educational model for computing might have the promise of being significantly more effective a decade from now?

Until the mid-1980's, our department explicitly focused our curriculum on key *concepts* in computing: examples of concepts might include Turing machines, abstract data types, algorithmic complexity, ray tracing, etc. We specifically did not commit to teach particular technologies in any given course; we took the intellectual “high ground” and said that education is where you teach concepts, not skills.¹ However, this led to untenable situations: one example was when students came into the senior-level operating systems course, often never having programmed in C, and they were expected to just “pick it up” (along with make, etc.) and do a large project while (in a 10 week quarter) learning concepts about concurrency, process management, memory management, etc.

We then agreed to complement the concepts we taught with a clearly defined mixture of specific *abilities* and practical *skills*. Examples of abilities might include the ability to build and test a program of roughly a thousand lines of code from scratch, to debug a program you didn't write, etc. Examples of skills might include using C++ templates to implement parameterizable data types, setting up and using a CVS repository, etc. By defining, along with key concepts, the various complementary skills and abilities that we expect our students to acquire, and by identifying how we expect our students to acquire them (in courses, through guided self-study, etc.), we have improved our curriculum and better prepared our students for industrial careers.

Some variant of this distinction among concepts, abilities and skills could form the basis for a far more effective computing curriculum. If nothing else, it can convey to students (and potential

¹ I once heard, and have oft repeated, a way to remember that there *is* a distinction between education and training: “If you think education and training are the same thing, then consider whether you'd send your teenager to sex training, instead of sex education, classes.” (I have also observed that it might not be hard to figure out which the teenager might prefer!)

students and, to be honest, industry) that there is a genuine difference between knowing specific skills (usually represented by the long list of acronyms) and understanding key concepts and having particular abilities. The distinction also allows us to more directly address the issue of rapid change, since it is reasonable to expect that in most cases skills will be updated most frequently, abilities less frequently, and concepts least frequently of all. And as the breadth of the field increases, the set of concepts, abilities and skills can be reformed, and the places in which students acquire the different material can be spread across appropriate units.

Now, this is obviously extraordinarily simplistic. It doesn't address key concerns about computing education including at least:

- we present computing as programming alone, starting with our introductory courses;
- we focus on individual achievements when most “real world” activities take place in teams;
- we focus more on projects that are manageable in time, scope, and scale than on realism, because we have constraints in terms of the students' time, the academic calendar, etc.;
- and,
- we encourage, or at least do not dispel, negative (and often inaccurate) perceptions of the field.

Neither does it give us any specific principles or guidance about how to manage the age-old question of “what is the core of computing?” that in practice leads us to find a balance between the union and the intersection of material that various faculty (and, indeed, various companies) believe must be included. This problem becomes even worse as the field grows in scope, and may be the most critical issue that we must address. Although I don't have any deep thoughts about how to address this problem, we clearly must find a way to distinguish, albeit softly, between computing as a discipline and computing as tool. Computing education definitely must address the needs of the discipline, but it must too be thoughtful about its service role to others.

Industry, I must note, often has wildly unrealistic hopes and demands for computing education. At one focus group meeting, people from a large aerospace company argued that our undergraduate program should produce graduates who would immediately be able to come onboard and save a multimillion dollar failing database system integration project – a project that a number of very experienced software engineers had designed and were having trouble fixing. You have to be kidding! At the same meeting, a manager at a major software manufacturer complained that our graduates take two or three years to become highly productive once hired. At the same time, he could identify little if anything that we could remove from the curriculum. I continue to wonder how they expect us to teach that productivity in an already full undergraduate program when they themselves are unable to teach it more quickly than in a couple of years. (By the way, the aerospace company also argued that we should no longer teach programming, but rather “only” software integration; not surprisingly, the software manufacturer took exception to that point of view. Balancing the needs of different parts of the industry might be more difficult than balancing the opinions of faculty members!)

In other words, rethinking computing education is a critical goal, and one that must be based on some principles and explicit goals, with a clear recognition of the specific issues that arise in computing, including our importance to industry, the rapidity of change, and the continual broadening of the field. The notion of concepts, abilities and skills is at best a minor step towards one of the principles.

