COMMUNICATIONS OF THE ACM
TRUSTED INSIGHTS FOR COMPUTING'S LEADING PROFESSIONALS

VIEWPOINTS

What Should We Teach New Software Developers? Why?

Fundamental changes to computer science education are required to better address the needs of industry.

Bjarne Stroustrup

Communications of the ACM
Vol. 53 No. 1, Pages 40-42
10.1145/1629175.1629192

Computer science must be at the center of software systems development. If it is not, we must rely on individual experience and rules of thumb, ending up with less capable, less reliable systems, developed and maintained at unnecessarily high cost. We need changes in education to allow for improvements of industrial practice.

The Problem

In many places, there is a disconnect between computer science education and what industry needs. Consider the following exchange:

Famous CS professor (proudly): "*We don't teach programming; we teach computer science.*"

Industrial manager: "*They can't program their way out of a paper bag.*"

In many cases, they are both right, and not just at a superficial level. It is not the job of academia just to teach run-of-the-mill programmers and the needs of industry are not just for "well-rounded high-level thinkers" and "scientists."

Another CS professor: "*I never code.*"

Another industrial manager: "*We don't hire CS graduates; it's easier to teach a physicist to program than to teach a CS graduate physics.*"

Both have a point, but in an ideal world, both would be fundamentally misguided. The professor is wrong in that you can't teach what you don't practice (and in many cases, never have practiced)

and therefore don't understand, whereas the industrial manager is right only when the requirements for software quality are set so absurdly low that physicists (and others untrained in CS) can cope. Obviously, I'm not referring to physicists who have devoted significant effort to also master computer science—such combinations of skills are among my ideals.

CS professor (about a student): "*He accepted a job in industry*."

Another CS professor: "*Sad; he showed so much promise*."

This disconnect is at the root of many problems and complicates attempts to remedy them.

Industry wants computer science graduates to build software (at least initially in their careers). That software is often part of a long-lived code base and used for embedded or distributed systems with high reliability requirements. However, many graduates have essentially no education or training in software development outside their hobbyist activities. In particular, most see programming as a minimal effort to complete homework and rarely take a broader view that includes systematic testing, maintenance, documentation, and the use of their code by others. Also, many students fail to connect what they learn in one class to what they learn in another. Thus, we often see students with high grades in algorithms, data structures, and software engineering who nevertheless hack solutions in an operating systems class with total disregard for data structures, algorithms, and the structure of the software. The result is a poorly performing unmaintainable mess.

For many, "programming" has become a strange combination of unprincipled hacking and invoking other people's libraries (with only the vaguest idea of what's going on). The notions of "maintenance" and "code quality" are typically forgotten or poorly understood. In industry, complaints about the difficulty of finding graduates who understand "systems" and "can architect software" are common and reflect reality.

But my Computer Hasn't Crashed Lately

Complaining about software is a popular pastime, but much software has become better over the last decade, exactly as it improved in previous decades. Unfortunately, the improvements have come at tremendous cost in terms of human effort and computer resources. Basically, we have learned how to build reasonably reliable systems out of unreliable parts by adding endless layers of runtime checks and massive testing. The structure of the code itself has sometimes changed, but not always for the better. Often, the many layers of software and the intricate dependencies common in designs prevent an individual—however competent—from fully understanding a system. This bodes ill for the future: we do not understand and cannot even measure critical aspects of our systems.

There are of course system builders who have resisted the pressures to build bloated, ill-understood systems. We can thank them when our computerized planes don't crash, our phones work, and our mail arrives on time. They deserve praise for their efforts to make software development a mature and trustworthy set of principles, tools, and techniques. Unfortunately, they are a minority and bloatware dominates most people's impression and thinking.

Similarly, there are educators who have fought the separation of theory and industrial practice. They too deserve praise and active support. In fact, every educational institution I know of has programs aimed at providing practical experience and some professors have devoted their lives to making successes of particular programs. However, looking at the larger picture, I'm unimpressed—a couple of projects or internships are a good start, but not a substitute for a comprehensive approach to a balanced curriculum. Preferring the labels "software engineering" or "IT" over "CS" may indicate differences in perspective, but problems have a nasty way of reemerging in slightly different guises after a move to a new setting.

My characterizations of "industry" and "academia" border on caricature, but I'm confident that anyone with a bit of experience will recognize parts of reality reflected in them. My perspective is that of an industrial researcher and manager (24 years at AT&T Bell Labs, seven of those as department head) who has now spent six years in academia (in a CS department of an engineering school). I travel a lot, having serious discussions with technical and managerial people from several dozen (mostly U.S.) companies every year. I see the mismatch between what universities produce and what industry needs as a threat to both the viability of CS and to the computing industry.

Many organizations that rely critically on computing have become dangerously low on technical skills.

The Academia/Industry Gap

So what can we do? Industry would prefer to hire "developers" fully trained in the latest tools and techniques whereas academia's greatest ambition is to produce more and better professors. To make progress, these ideals must become better aligned. Graduates going to industry must have a good grasp of software development and industry must develop much better mechanisms for absorbing new ideas, tools, and techniques. Inserting a good developer into a culture designed to prevent semi-skilled programmers from doing harm is pointless because the new developer will be constrained from doing anything significantly new and better.

Let me point to the issue of scale. Many industrial systems consist of millions of lines of code,

whereas a student can graduate with honors from top CS programs without ever writing a program larger than 1,000 lines. All major industrial projects involve several people whereas many CS programs value individual work to the point of discouraging teamwork. Realizing this, many organizations focus on simplifying tools, techniques, languages, and operating procedures to minimize the reliance on developer skills. This is wasteful of human talent and effort because it reduces everybody to the lowest common denominator.

Industry wants to rely on tried-and-true tools and techniques, but is also addicted to dreams of "silver bullets," "transformative breakthroughs," "killer apps," and so forth. They want to be able to operate with minimally skilled and interchangeable developers guided by a few "visionaries" too grand to be bothered by details of code quality. This leads to immense conservatism in the choice of basic tools (such as programming languages and operating systems) and a desire for monocultures (to minimize training and deployment costs). In turn, this leads to the development of huge proprietary and mutually incompatible infrastructures: Something beyond the basic tools is needed to enable developers to produce applications and platform purveyors want something to lock in developers despite the commonality of basic tools. Reward systems favor both grandiose corporate schemes and short-term results. The resulting costs are staggering, as are the failure rates for new projects.

Faced with that industrial reality—and other similar deterrents—academia turns in on itself, doing what it does best: carefully studying phenomena that can be dealt with in isolation by a small group of like-minded people, building solid theoretical foundations, and crafting perfect designs and techniques for idealized problems. Proprietary tools for dealing with huge code bases written in archaic styles don't fit this model. Like industry, academia develops reward structures to match. This all fits perfectly with a steady improvement of smokestack courses in well-delineated academic subjects. Thus, academic successes fit industrial needs like a square peg in a round hole and industry has to carry training costs as well as development costs for specialized infrastructures.

Someone always suggests "if industry just paid developers a decent salary, there would be no problem." That might help, but paying more for the same kind of work is not going to help much; for a viable alternative, industry needs better developers. The idea of software development as an assembly line manned by semi-skilled interchangeable workers is fundamentally flawed and wasteful. It pushes the most competent people out of the field and discourages students from entering it. To break this vicious circle, academia must produce more graduates with relevant skills and industry must adopt tools, techniques, and processes to utilize those skills.

Dreams of Professionalism

"Computer science" is a horrible and misleading term. CS is not primarily about computers and it

is not primarily a science. Rather it is about uses of computers and about ways of working and thinking that involves computation ("algorithmic and quantitative thinking"). It combines aspects of science, math, and engineering, often using computers. For almost all people in CS, it is an applied field—"pure CS," isolated from application, is typically sterile.

What distinguishes a CS person building an application from a professional in some other field (such as medicine or physics) building one? The answer must "be mastery of the core of CS." What should that "core" be? It would contain much of the established CS curriculum—algorithms, data structures, machine architecture, programming (principled), some math (primarily to teach proof-based and quantitative reasoning), and systems (such as operating systems and databases). To integrate that knowledge and to get an idea of how to handle larger problems, every student must complete several group projects (you could call that basic software engineering). It is essential that there is a balance between the theoretical and the practical—CS is not just principles and theorems, and it is not just hacking code.

This core is obviously far more "computer-oriented" than the computing field as a whole. Therefore, nobody should be called a computer scientist without adding a specialization within CS (for example, graphics, networking, software architecture, human-machine interactions, security). However that's still not enough. The practice of computer science is inherently applied and interdisciplinary, so every CS professional should have the equivalent to a minor in some other field (for example, physics, medical engineering, history, accountancy, French literature).

Experienced educators will observe: "But this is impossible! Hardly any students could master all that in four years." Those educators are right: something has to give. My suggestion is that the first degree qualifying to practice as a computers scientist should be a master's—and a master's designed as a whole—not as a bachelor's degree with an appended final year or two. People who plan to do research will as usual aim for a Ph.D.

Many professors will object: "I don't have the time to program!" However, I think that professors who teach students who want to become software professionals will have to make time and their institutions must find ways to reward them for programming. The ultimate goal of CS is to help produce better systems. Would you trust someone who had not seen a patient for years to teach surgery? What would you think of a piano teacher who never touched the keyboard? A CS education must bring a student beyond the necessary book learning to a mastery of its application in complete systems and an appreciation of aesthetics in code.

I use the word "professional." That's a word with many meanings and implications. In fields like medicine and engineering, it implies licensing. Licensing is a very tricky and emotional topic. However, our civilization depends on software. Is it reasonable that essentially anyone can modify a critical body of code based on personal taste and corporate policies? If so, will it still be

reasonable in 50 years? Is it reasonable that pieces of software on which millions of people depend come without warranties? The real problem is that professionalism enforced through licensing depends on having a large shared body of knowledge, tools, and techniques. A licensed engineer can certify that a building has been constructed using accepted techniques and materials. In the absence of a widely accepted outline of CS competence (as I suggested earlier), I don't know how to do that for a software application. Today, I wouldn't even know how to select a group of people to design a licensing test (or realistically a set of tests for various subspecialties, like the medical boards).

What can industry do to close the gap? It is much more difficult to characterize "industry" and "industrial needs" than to talk about academia. After all, academia has a fairly standard structure and fairly standard approaches to achieving its goals. Industry is far more diverse: large or small, commercial or non-profit, sophisticated or average in their approach to system building, and so forth. Consequently, I can't even begin to prescribe remedies. However, I have one observation that related directly to the academia/industry gap: Many organizations that rely critically on computing have become dangerously low on technical skills:

Industrial manager: "*The in-sourcing of technical expertise is critical for survival.*"

No organization can remain successful without an institutional memory and an infrastructure to recruit and develop new talent. Increasing collaboration with academics interested in software development might be productive for both parties. Collaborative research and an emphasis on lifelong learning that goes beyond mere training courses could play major roles in this.

Conclusion

We must do better. Until we do, our infrastructure will continue to creak, bloat, and soak up resources. Eventually, some part will break in some unpredictable and disastrous way (think of Internet routing, online banking, electronic voting, and control of the electric power grid). In particular, we must shrink the academia/industry gap by making changes on both sides. My suggestion is to define a structure of CS education based on a core plus specializations and application areas, aiming eventually at licensing of software artifacts and at least some of the CS professionals who produce them. This might go hand-in-hand with an emphasis on lifelong industry/academia involvement for technical experts.

Author

Bjarne Stroustrup (bs@cs.tamu.edu) is the College of Engineering Chair in Computer Science Professor at Texas A&M University in College Station, TX.

Footnotes

DOI: http://doi.acm.org/10.1145/1629175.1629192

USER COMMENTS (6)

Excellent article.

Re: "The idea of software development as an assembly line manned by semi-skilled interchangeable workers is fundamentally flawed and wasteful." This has been a common theme in much SDLC-related commentary. Some argue that *no* metaphor is applicable. I have more recently begun to ask whether product development is actually the appropriate metaphor. Product development and portfolio management in industry is costly, iterative, and risky process, subject to many of the same issues as software. Yet I have seen little if any exploration of potential parallels.

Charles T. Betz
Enterprise Architect
Wells Fargo Bank, N.A.

— Charles Betz, December 22, 2009

As an academic once tasked with overhauling the software engineering undergrad course, I can wholeheartedly underwrite this article. One sentence, in particular, stood out for me: "Proprietary tools for dealing with huge code bases written in archaic styles don't fit this model."

I believe that /the/ core issue is that academic computer science has a hard time working in a meaningful way on "huge code bases". Yet, as Mr. Stroustrup often mentions and I wholeheartdely agree with, scale is precisely the distinctive difference.

However, it seems that it is not entirely clear how to achieve this. My suggestion would be that we should create a meaningful incentive system for academic research. That is, some way that allows research to do its job and measure its applicability to the real world issues. I bet many academics would love to do this, instead of work on purely theoretical issues.

My suggestion how to gou about this would be to create accepted tools for dealing with large-scale code bases. Tools to measure, quantify, summarize, correlate, generalize, and more abstract analysis on top of those. Some tools in this direction are already available. We also need tools to /modify/, or maybe "refactor" such code-bases and make comparisons, so that the effect of different engineering approaches can be verified. Also, some early tools in this direction exist.

Such tools would enable researchers to work on large-scale issues, as is needed.

Last, not least and probably not easiest, collaboration with experts on the "soft" issues could enable psychologically-grounded metrics for, just as one example, API usability and complexity, so that again, the effect of different choices can be measured.

— Ingo Lutkebohle, January 6, 2010

>>For many, "programming" has become a strange combination of unprincipled hacking and invoking other people's libraries (with only the vaguest idea of what's going on).<<

This is an excellent observation, and unfortunately, very true...

— Robert Szymczak, January 19, 2010

Bjarne makes an insightful and provocative plea for a more professional software industry. People that will write programs need to be taught by people who are skilled in detailed design, data structure, code design and all the other computer science skills. There is no excuse for Computer Science Professors not writing some code or at least being able to critically read code. The computer science education by Bjarne teaches the student to write professional programs. Fred Brooks teaches that we need another order of magnitude effort to take the program and produce a programming system product from it. This additional effort is learned in software engineering programs.

Must a software engineer be a professional programmer? i think not, but they must be taught the essentials of program design. I think that software engineers must be able to solve system problems for software intensive systems. Alternatively, these people could be called systems engineers well educated in software technologies and processes.

We need people educated in software who can solve real problems. No single standard educational course of study focuses soley on this need.

— Lawrence Bernstein, February 3, 2010

Excellent and thought provoking article. Here is a suggestion for improving the quality of CS graduates: What if the entire University were to run on the software produced by the University CS students?

Granted, this is not possible on day one, but if the University were to adopt this policy, over time, CS students would have hands on experience with real-life code bases and will learn all parts of software design and development. They will learn how to fix bugs, test, refactor code, appreciate the importance of good software coding practices. They will also learn how to add new features by interacting with customers. Professors can take actual code snippets and teach students with it. Best of all, students can see the results of their efforts before they graduate.

I bet if the University is bold enough to adopt this policy, I can guarantee that the students of this university will be highly valued by industry. This is a win-win for all: Students, Professors, University, and the Industry.

Some of the great companies have such an internal policy - all systems sold by the company must be used internally for the same purpose. Companies like Oracle, Microsoft, Amazon.com do this everyday. The University can do the same and should do this if it is serious about the quality of the CS graduates it produces.

Kiran Achyutuni,
Founder, www.mashedge.com

— Kiran Achyutuni, March 4, 2010

Interesting discussion. CS is about complexity theory, computability, algorithms, data structures, automata theory, quantum computation science, formal languages and much more. There's a whole theoretical background that is unique to CS.

Definitely, there should be separated majors, such as Software Engineering or Data and Information Management, in order to avoid misconceptions and to address industry's specific needs.

In short, CS is a basic science in wich we can learn about modelling the complex processes that occur in Nature by using abstract mathematical tools. The models we construct in CS (basically, algorithms) have to (i) be given formal descriptions and proofs; (ii) have its fundamental properties investigated (complexity bounds, completeness, etc.); and much more.

In short, CS is a basic science and other sciences can benefit from it as well by using its outocomes. An example of this is Artificial Intelligence. Although it first got inspiration from early mathematical models of the brain (neuroscience), it became clear thereafter that the 'algorithimic' way of modelling natural processess would be fundamental to AI. Today, AI is commonly regarded as a standard discipline in CS, though still multidisciplinary.

So, why not segregate the lots of teaching contents flooding CS students minds into separate majors?

Furthermore, let's not forget the revealing quote of a prominient computer scientist:

"Computer Science is no more about computers than Astronomy is about telescopes" Edsger Dijkstra

That summarizes well what CS is definitely not about.

I believe that multidisciplinarity should be raise from a well planned reform in the graduate educational system, instead of requiring that a CS

major covers every single topic from software engenieering to computation theory. Both are, separetedly, deep and complex enough disciplines to be studied in their own terms.

Carlos R. B. Azevedo
MSc Candidate in CS
Center for Informatics (CIn)
Federal University of Pernambuco
Recife, Brazil

twitter.com/crbazevedo

— The account that made this comment no longer exists., April 6, 2010