We Can Teach Software Better

Mary Shaw
Carnegie Mellon University

Appeared in *Computing Research News*, 4,4 September 1992 (pp. 2, 3, 4, 12)

Introduction

In recent issues of CRN, Bill Wulf and Dave Patterson ask some questions about undergraduate computer science programs: Are we teaching the best content in the best way? Can we do so without fragmenting the discipline or creating administrative obstacles? [Wulf 91, Patterson 92] As they observe, the last two decades have seen radical changes in hardware technology, networking, system interconnection, and sophisticated applications, but our curricula generally ignore these changes. Further, software production problems lead the list of problems in developing computer applications. Wulf and Patterson ask why our current programs don't teach these improved technologies to the students who will need to apply them.

I would like to look specifically at education in software development: programming, programmed systems, and the engineering of software. This is not the whole of computer science, but it includes a large share. The typical software curriculum features dinosaur courses, classroom presentations that don't use new technology, naive approaches to software development, innocence of engineering design considerations, a severe shortage of examples relevant to anyone but a systems programmer, and ignorance of the system context of most useful software.

Focus on Ideas, not Artifacts

Let's organize our courses around ideas rather than around artifacts. This helps make the objectives of the course clear to both students and faculty. Engineering schools don't teach boiler design -- they teach thermodynamics. Yet two of the mainstay software courses -- "compiler construction" and "operating systems" -- are system-artifact dinosaurs.

We do not need forty students per university per semester who think they are compiler builders, as the description of the usual compiler course would suggest. Indeed, most of the faculty who design these courses will say, when pressed, that the real objectives include learning about:

- the structure of a well-understood medium-sized system,
- describing interactions of several modules,
- more sophisticated algorithms and data structures (related to symbol tables, parse trees, and graph traversal),
- practical problems of applying a well-understood piece of theory (syntactic analysis),

and similar topics that apply equally well to other kinds of software. Compilers were among the first sizable well-understood software systems, so this was the logical when these courses were first developed. Unfortunately, the course title, investment in textbooks, and old habits seem to make it impossible to replace the compiler now that there are other good examples.

In the Carnegie-Mellon curriculum design, we proposed to redistribute the conventional comparative programming language and compiler material, plus new material, to three new courses. [Shaw 1985] First, a junior-level course about the nature of languages and interfaces would introduce programming language structures, "little languages", and user interface problems. Second, a follow-on course about transducers of programs would cover editors, macro systems, programming environments, test data generation, and program generators as well as compiling techniques sufficient to handle a simple language. Finally, we specified a senior elective for the specialized programming language and compiler topics such as code optimization, fine points of language design, and detailed interactions between languages and their implementations.

In that curriculum design we also proposed re-organizing the topics usually covered in operating systems and database courses, bringing in selected topics from hardware and formal methods. We planned courses on time and resource allocation, on issues of large data, on communication and networks, and on classes of program organizations. Thus, for example, one course on "time and resources" would bring together ideas about coordinating multiple processes competing for resources. This includes synchronization mechanisms (locks, semaphores, monitors, rendezvous), scheduling (deadlock, starvation, fairness, contention), real-time response, hardware interrupts, clocks, transactions, programming language constructs for concurrency, and temporal logic. These reorganizations were somewhat speculative; their challenge has not yet been answered. I continue to believe they are good ideas.

This is not, of course, to say that we should abandon the applications. Some engineering schools have gone too far, producing students who know only principles and can't design a boiler. However, we are far from having that problem. We do need a healthier balance that emphasizes important ideas and places them in the context of good practice.

Improve the Way we Teach Software Development

Any student who claims an education in software must be good at software development. This includes proficiency in both programming and engineering design. The best software engineering education we can provide undergraduates emphasizes these topics, which are integral to the computer science curriculum. These changes do not require separate software engineering courses, let alone separate curricula. Moreover, they will improve the curriculum for all students who learn about software, not just the majors.

Software Development Skills

Programming skills provide the base for software systems education. Even a cursory look at what programmers know and do reveals problems in the current software curriculum. Shortcomings include:

Programming from scratch: Most courses teach students to code from scratch, rather than by modifying existing programs or by working from model solutions. Moreover, students rarely read good programs. It's as if we asked students to write good prose without first reading good prose.

Equating program text with software: A complete software product includes not only the code, but also the analysis that led to the design, the user documentation, the test suites, and records of design decisions that will be important to the maintainer. Students too often focus on the code, do ad hoc testing, neglect the user documentation, and ignore everything else.

Learning abstract skills at the expense of specific content: Our curricula are very strong in techniques for formulating solutions from first principles. We present too few well-known examples of good solutions for study and emulation. We fail to teach respect for and reliance on existing results such as code libraries.

Programming before reasoning: Although the situation is improving, coding and debugging still seems to win out over specification, analysis, and careful construction or derivation.

We can cure these problems without major disruption to our course structure by changing the emphasis within individual courses:

Study good examples of software systems: Doing this properly requires case studies organized for presentation. Meanwhile, do careful guided reading of good code and make assignments that start from running code provided with the assignment.

Learn more facts: Software developers won't use resources they don't know about. Teach more specific facts such as available subroutine libraries and interface standards. Reinforce these with assignments that require students to use them.

Incorporate reference material as it becomes available: There is currently a dearth of good reference material to help software developers avoid re-invention. As such material

becomes available, use it. Meanwhile, teach students to use reference manuals, library documentation, and the like effectively.

Present theory and models in the context of practice: Emphasize durable ideas that will transcend a major shift of technology. Students often learn them best when they appear in concrete examples; good examples will themselves be worth remembering for reuse.

Engineering Skills

Practical, useful software doesn't happen by accident. It requires design skills not unrelated to traditional engineering design. Some of the engineering shortcomings of our curricula are similar to the programming shortcomings: failure to study good systems, failure to develop reasoning skills, failure to understand maintenance and support issues. Some others are:

Implementing the first design: Problems often admit of more than one solution. The best solution in a given setting often depends heavily on facts about the user or the intended use of the system.

Designing for the implementor: Implementors often chose solutions that match their own tastes, not the needs of the customer.

Failing to understand problem scale: Class assignments usually emphasize functionality but neglect performance requirements, especially scale requirements such as size and throughput.

Writing throwaway exercises: When assignments are discarded as soon as they are graded, students have no incentive for creating comprehensible, well-documented, maintainable software.

Ignoring reliability, safety, and other system requirements: Class assignments usually focus on getting correct results for correct inputs. They occasionally require rudimentary checking of inputs, and they occasionally require performance measurement. Students rarely do systematic analyses of reliability and safety. Similarly, class assignments address asymptotic performance of algorithms and sometimes speedy code, but many students never confront a requirement for practical real-time response.

We can address these flaws, too, within the existing course structure:

Require consideration of at least two serious designs: Make students choose between design alternatives. Require these choices to address customer needs.

Require consultation with end users: Use projects with actual clients. Unless end users have a voice in reviewing a design, students won't understand that their needs and preferences are different from the students' own.

Teach back-of-the-envelope estimation: Students often believe that they can't do any analysis until all the facts are in hand. Teach them to do quick estimates of usage levels, throughputs, sizes, bandwidths. Show them how this can provide early guidance about scale and performance.

Modify and combine programs as well as creating them: Teach students to work with program structures devised by others, to reuse components, to adhere to standards, and to value good documentation.

Test student implementations with bad data: Run test cases chosen by the instructor, not just demonstration data from the student. Include not only correct inputs, but also erroneous and even malicious inputs. Do this not only for isolated assignments, but as a matter of course.

Make assignments with embedded system requirements: Bad data isn't the only source of real-world demands. Make assignments that expose students to nondeterminism, end-to-end time requirements, and race conditions.

Make the Content More Interesting and Relevant

Patterson suggests a number of new courses that should be more available to students. To his list I'd add real-time systems, architectures of software systems, parallel computation, and human-computer interaction. Each of us probably has a few favorite topics.

We can't add fresh material to a curriculum that is already full. We have accumulated a lot over the last 25 years; tradition and inertia make it hard to prune. Patterson suggests contracting lower-level courses. We can also add new material by replacing the examples that carry the ideas. After all, our main goal is mastery of the basic concepts of the discipline; the specific examples matter less.

For example, instead of making the introductory programming course optional, we could refocus it on more interesting examples. Most students who enter the university with programming experience have skills in writing code, but they often lack much of what the course objectives (should) specify: real mastery of the concept of algorithm, a certain kind of problem-solving skill, and a systematic approach that leads to readable, understandable, maintainable code. Why don't we start by studying good programs that do things students find interesting, then move on to changes that make them even more interesting? In addition to introducing algorithmic reasoning and problem solving, we could set better standards of style and teach the use of some of the simpler tools.

Patterson says new course models won't be adopted unless they have textbooks. Certainly there is inertia and social pressure. But any teacher who knows an area can put together a readings collection. And any teacher willing to put in just a little effort can use one that someone else has prepared. Undergraduate seniors and most juniors should be perfectly capable of reading papers from *IEEE Software* and *Computer*; these journals are edited for the practitioner. I've done two courses this way recently (a software engineering project course and a course on architectures for software systems), and I'm happier with the papers than with textbooks. Descriptions with reading lists appear in appropriate places, so anyone who doesn't want to work from scratch can work from these [SBC 91, SGOSS 92]. It's true that a textbook stabilizes a course -- but published collections of readings can do that, too.

We don't encourage interdisciplinary study. It's true that a student can construct a double major by using the electives of one major to satisfy the requirements of the second. But this falls short in two ways: first, it prevents the student from exploring the interesting byways of either major; second, and more seriously, it has no way to teach the computational aspects of the other discipline. A few computer science departments are now setting up joint majors with other departments. More should. Both departments must be willing to reduce the normal requirements and to develop one or more advanced computational courses that rely on prerequisites in both departments.

What we do not need is fragmentation within computer science. The current pressure for separate software engineering programs and departments rejects the historical productive interaction between the practical and theoretical sides of the discipline. I believe this interaction is one of the reasons for our rapid development over the past three decades. None of my proposals even hints that separate curricula or departments are appropriate. In fact, many of these changes would be *harder* to make in separate curricula than in a unified program. As Wulf argued, separating software engineering from computer science is absolutely counter-productive, both intellectually and administratively.

Use Our Own Technology in Courses

We are poor users of our own technologies. Except for compilers, editors, and the occasional syntax-directed editor, parser generator, or grading program, software education makes little use of software technology. Many other possibilities spring to mind: subroutine or component libraries, integrated environments, simulations, program skeletons, test harnesses, spreadsheets and project planning software, educational-strength versions of industrial tools, and living case studies are just a few. Some courses take advantage of these opportunities, but this is far from the norm. We

have lots of excuses for not doing so: "it's not available at my school", "we can't afford it", "it isn't compatible with my textbook", and so on.

These seem to be roundabout ways of saying we don't realize that these technologies are a vital part of modern software practice. Yes, there are practical problems: a single instructor can't acquire or develop a sophisticated environment for a single course. But we *can* do a great deal for a single course, and we should undertake facility development for education as seriously as we take facility development for research. The same problems with getting support for software and maintenance will, alas, reappear.

For a decade or more, we've been talking about distributing software for particular courses -- often coupled to particular textbooks. I recall offering code to support the major examples in a textbook in 1981. However, there was no effective distribution mechanism, and almost no one took me up on the offer. Publishers tell me that the instructor's materials, including answers to exercises and overhead projection masters, are major factors in selling textbooks. We should be targeting the day that the portable software support is an even stronger selling point.

Summary

We can do many things to revitalize the software curriculum. Most don't require complete replacement or separate departments. As it stands, we are serving no one well. Our courses don't represent the best of computer science, they are only marginally relevant, and they are too often neither fresh nor exciting.

References

GSOSS 92

David Garlan, Mary Shaw, Chris Okasaki, Curtis M. Scott, and Roy F. Swonger. "Experience with a Course on Architectures for Software Systems." *Proc. Sixth SEI Workshop on Software Engineering Education*, Springer-Verlag 1992 (to appear).

Patterson 92:

David A. Patterson. Has CS changed in 20 years? *Computing Research News*, 4, 2 (March 1992), pp.2-3.

SBC 91

Mary Shaw, Bernd Bruegge, and John Cheng. A Software Engineering Project Course with a Real Client." *Software Engineering Institute Educational Materials Package* CMU/SEI-91-EM-4, Carnegie Mellon University, July 1991.

Shaw 85:

Mary Shaw (ed). *The Carnegie-Mellon Curriculum for Undergraduate Computer Science*. Springer-Verlan 1985.

Wulf 91:

William A. Wulf. SE programs won't solve our problems. *Computing Research News*, 3, 5 (November 1991), p.2.