Emphasizing Design in the Computer Science Curriculum

Thad R. Crews, Jr.
Assistant Professor
Department of Computer Science
Western Kentucky University

Introduction

My goals as an educator and my opinions about the future of computer science education are a direct result of my experience. I knew I wanted to work in the field of computer science since I was a sophomore in high school. Two years of high school programming courses were followed by a computer science degree from a CSAB accredited university. Upon graduation, I accepted a typical industry position: a software engineer for a large information technology company. Almost immediately I was astonished to realize that, despite a solid education, I was ill prepared for my role as a software *engineer* (emphasis mine). Specifically, I had a great deal of experience in writing code and in the theoretical foundations of my discipline, but little experience with the software engineering process. This sparked my interest in the education process, and solidified my resolve to earn a terminal degree and re-examine the process that had left me wanting.

I decided to pursue my doctorate at Vanderbilt University for the strength of both its computer science and education programs. My involvement as a member of the Cognition and Technology Group at Vanderbilt (CTGV) proved invaluable in stretching my understanding of what computer science education could entail. Specifically, my work with the Jasper Project involving the development of instructional software to assist curriculum has influenced my attitudes regarding the computer science curriculum.

In my two years as a computer science faculty member at Western Kentucky University I have paid close attention to opportunities to supplement the established curriculum with new instructional technology and pedagogical approaches which will, hopefully, provide computer science students with an understanding of engineering principles. In particular, I have attempted to emphasize *design* whenever possible.

Design in Computer Science

The term *software engineering* reflects the belief that software production should be an engineering like activity. The philosophies and paradigms of established engineering disciplines should be used to improve the quality of software. As a result of industry following software engineering principles, over the past 20 years there has been a steady productivity increase of roughly 6 percent per year, comparable to what has been observed in many manufacturing industries [1].

Software development involves a series of phases: requirements, specification, design, implementation, and maintenance. In most undergraduate curricula, however, students spend a disproportionate amount of time and effort on implementation issues, including language-specific syntax and the implementation of classical algorithms. This is largely due to the fact that requirements, specifications and maintenance activities are difficult to adequately model while covering the breadth of the undergraduate curriculum. It is my strong belief, however, that design can and should be stressed throughout the educational process. A solid grounding in design benefits the student. Effective maintenance depends more on ones ability to understanding design than implementation. Likewise, specifications and requirements are more related to design than implementation.

For the remainder of this essay I would like to present my educational strategies for integrating design into the computer science curriculum.

Strategies for Teaching Freshmen

"Introducing Design"

Traditionally the first computer science course has been a language-specific programming course. While the textbooks for these courses acknowledge the stages of software engineering, the reality is that they focus almost entirely on issues of language-specific syntax and implementation. As a result, students are mislead as to the relatively small role implementation plays in the software development process. (Schach [1] uses industry data-points to suggest that basic coding makes up only 5% of the software development effort.)

I am leading an effort at Western Kentucky University to redesign our introductory computer science course so that design and testing are given equal importance to implementation. In addition, the course is language independent and avoids the temptation to focus on issues of syntax. To support this course, we are developing instructional technology that allows students to design, develop, execute and evaluate algorithms as flowcharts. A paper discussing this instructional software appears in the conference proceedings.

Strategies for Teaching Sophomores
"Design of Large Systems"
One of my goals for a redesigned computer science curriculum is to offer a formal course on software engineering in the second year. In far too many programs this course is delayed until the junior or even senior year. This, I believe, is an injustice to the student. As I stated earlier, the breadth of the curriculum necessitates frequent programming exercises, and that frequency demands assignments be manageable in size. The result is that students are not exposed to the dynamics of large systems until they take a course in software engineering. Furthermore, it is this course – more than any other – which gives students a perspective of industry practice. The sooner students are exposed to this perspective, the sooner they are able to develop a professional mindset regarding their coursework.

Strategies for Teaching Juniors
"Theoretical Limits and Classical Techniques"
Students in the third year are exposed to the topics that distinguish a software engineer from, say, an electrical engineer that happens to know how to write C++ code. In this year, students learn about the theoretical foundations and associated limits of the field. Additionally, students are introduced to the classical algorithms that all software engineers have in their toolbox.

My suggestion for these courses is that they be taught in a problem-based methodology. For example, a typical activity would be to introduce Dijkstra's shortest path algorithm followed by a discussion of a programming application. The approach I take with my students is to confront them with various real-world problems whose solutions depend on Dijkstra's and related techniques and ask them to derive solutions. I intentionally place students in situations where they must design solutions on their own before I expose (inoculate?) them to classical solutions. Using this approach, some students will come across the basic solution, some will be *this* close to the solution, and some will admittedly be no where near a solution. Yet my experience has been that all the students appreciate the classical design more after having attempted the problem first. Even more importantly, students appreciate the design behind the algorithm and are therefore, I suspect, better able to apply their knowledge.

When students are involved in this type of learner-centered approach, they develop analysis and design skills. Other learner-centered approaches should be examined for their applicability during these cognitively foundational courses. For example, I typically ask students to work independently on their design activities. I can see how team-based efforts could be beneficial in other ways. I don't think the particular approach matters just so long as it is learner-centered. In fact, I would hope to see multiple learner-centered approaches followed over the entire curriculum.

Strategies for Teaching Seniors
"Capstone Experience and Applied Research"
I am a supporter of a senior capstone experience. Students benefit in many ways from these courses. Students revisit the full software lifecyle and are afforded one more opportunity to realize software development as more than implementation. Students typically work in groups and gain useful experience with the issues involving team organization. I also encourage students to make use of CASE (computer-assisted software engineering) tools such as a coding tool, online documentation, version control tool, and a data dictionary. While these (and other) tools are discussed in the software engineering course, I believe their applied use is quite valuable in preparation for industry.

There is no difficulty finding project ideas for such a course. Many schools generate valuable ties with local industry by having students develop software that addresses local needs. Another option for these courses is to expose students to research issues and develop software based on the faculty member's research interests. I personally favor the second approach (with the understanding that other efforts to secure close ties with local industry are pursued, such as the formation of a professional advisory board.) There are two reasons why I support a research based capstone course. First, by involving students in research we encourage further learning on their part. It has been my experience that students respond well to the opportunity to contribute to the overall discipline. My second reason for supporting a research based course is that it benefits the faculty member. When teaching such a course, one must quickly bring the class up-to-speed on the topic of study. This process of explaining in clear and succinct terms my research frontiers has always proved to be beneficial in my own personally thinking. This is further supported by the fresh and novel reactions students offer on the subject, the result being that such a course generates more good ideas for further study than I could ever have derived on my own.

Summary
My education strategy addresses the need for computer science faculty to support the notion that software development is an engineering line activity and, as such, we need to better integrate engineering principles in our curriculum. The suggestions I put forth in this essay are fully consistent with the requirements of traditional accredited curriculum. (Western Kentucky University is CSAB accredited.) I see no need to radically redesign the classic curriculum. However I do see a benefit from integrating the following:

- A modification of the first semester course to better support design and testing;

- A software engineering course early in the curriculum that exposes students to the entire software lifecycle using a learning-by-doing model;
- An openness to explore learner-centered designs the more cognitive and theoretical courses;
- A capstone experience in the senior year that revisits again the full software lifecycle.

It is my hope that these integrations will result in students completing their computer science education with a better appreciation of the activities associated with being a software engineer.

References

[1] Schach, S.R., *Classical and Object-Oriented Software Engineering*, 4th edition, Prentice-Hall (1999).