**Changing the way we teach Computer Science**

Brendan Best

University of Wisconsin – Stout

Learning Technologies, ICT710

Professor Kris Isaacson

October 18, 2017

Changing the way we teach Computer Science

There is a problem in computer science. It’s not artificially intelligent robots, not smart cars gone awry, not Skynet awakening and sealing our doom. It’s a lack of brain power. A need for experts. Despite an ever-increasing need for qualified computer science graduates, participation and graduation rates in the United States have actually *fallen* over the last decade, especially in comparison to the growing number of open positions in the field. What’s more, according to the Bureau of Labor Statistics (2015), due to the growing demand for software developers, employment in the field “is projected to grow 17 percent from 2014 to 2024” a number far greater than overall job growth in this economy. Compounding the problem of decreased participation is a dropout rate as high as 60%, not just in computer science, but across the gamut of science related programs (Becerra-Fernandez, I., Elam, J., & Clemmons, S. (2010)). So, who’s to blame? Or, better stated, *what’s* to blame? Why, in this increasingly technical, knowledge-based job market, are many young people eschewing a career field that, ironically, brings them the technology they now find so ubiquitous in their lives? Is it an incompatibility between the rapidly evolving subject material and older, more static twentieth century teaching models? Is it outdated technology in the classroom that hampers progress and limits flexibility? Is it a matter of training educators and administrators in the educational and career needs of the modern computer science student? While there are plenty of other factors that likely contribute to these trends, this paper aims to answer these questions by focusing on existing learning methods and technologies used in higher education, and seeking out new trends in the field. The factors behind what motivates students to get into the field in the first place – societal, economic, personal – and the factors that motivate them to drop out or avoid it entirely are varied and complex. Finding solutions through an academic approach is no easy challenge, either – just as students in computer science, or those who would otherwise be drawn to the field, are often unique in their motivations, they’re equally unique in their learning needs (Giannakos, M., Pappas, N., Jaccheri, I., & Sampson, O. (2017)). These unique sets of motivations and learning needs will require more than just a singular, one-size-fits-all solution in both technology and methodology. Either an entirely new approach, or a hybrid of existing solutions will be needed to address these fundamental issues.

Computer science (which this writing may lapse into usage of its acronym, CS or CIS, i.e., Computer Information Systems), has been around as a field of study since as early as the 1950’s (Hopcroft, J. (1987)) and has rapidly increased in relevance since. Code was written to put a man on the moon, network engineers were born when the first email was sent, and since the dawn of the World Wide Web in the early 1990’s, the field has become a requirement for the economy. Long gone are the days of ecommerce being a ‘nice to have’ for a major retailer, or online access being just an extra option in school or the workplace. CS technologies are now an integral part of nearly every aspect of modern life, and its builders are a vital workforce to move it all forward. Now, educational systems are commonly large, bulky things, slow to change (often with good reason), and slow to respond to trends. This isn’t necessarily a bad thing, especially when applied to more stable fields of study like geology, history, or even medicine. Those programs can incorporate new knowledge year over year, largely without the need to commit to radical change in structure or technology. But computer science is a horse of a different color. Take, for example, the evolution of programming languages over the last fifty or so years – not only did the syntax change from machine code (programming the bits and bytes of each little step a processor needed to take) to higher level languages like Java or PHP (much more readable from a human perspective) but the very technology on which they ran, and the diversity of needs that they approached (from spitting out calculations to creating modules for the web, (Ghosh, D., Sheehy, J., Thorup, K., & Vinoski, K. (2012))). Continuing this example, this pattern of change has accelerated even further with online software development coming into the forefront. The bottom line: the world’s economy now runs on the continued work of computer science. It relies on continuous adaptation and improvement. These reasons and more are why the trends in graduation rates and participation are especially troubling. CS is a complex web of interrelated technologies and systems that necessitates an educational system up to the task. Without adopting new educational standards and practices to retain and educate current and future students, and as the need for more graduates in CS roles continues to increase at such a high rate, the impact on the technological economy, and, by proxy, the economy at large, might be significant. Without an adequate pool of talent to draw from, it isn’t just Silicon Valley that will need to refactor their way of doing business, it’s any company that builds its own software or manages its own online presence. Large portions of the economy could be affected in the future, as companies will either need to reduce their technology needs somehow, find yet-to-be-discovered ways in which to automate highly skilled labor, or circumvent the educational system entirely by finding and educating their own potential candidates.

**Literature Review**

There have been hundreds of studies on teaching methods and learning technology done in the several decades, but comparatively few have dealt with matters specifically related to computer science, as the needs and challenges outlined previously have only emerged recently. What studies have been done address some of the key issues in attracting and retaining computer science students by utilizing existing technologies and methods in unique ways.

In Hazzan, O., et al.’s (2010) paper, “Didactic Transposition in Computer Science Education” they explored two different research projects that adapted *didactic transposition* – the act of adapting real world frameworks and procedures for the classroom environment – specifically for software and software project development. First, they delve into the complexities and meanings of what didactic transposition is, explaining its origination in 1985 as a means by which to make mathematical education more attached to the ‘real world’ by introducing such utilities as the two-column proof (something used by mathematicians working in the field), and then expand the definition and scope to include its meaning and usage in what they call CSE, or, Computer Science Education.

The first project – of which we’ll give the most focus here, as it involved teaching in higher education – was a class taught by one of the paper’s authors, Dubinsky, at the Technion Israeli Institute of Technology. The class split students up into groups that functioned as individual units throughout the semester, working on a software project assigned to them. Each team also had its own mentor assigned, similar in role to what a project manager would do on a live software project. They introduced the usage of Agile methodology to each team as well, a reflection of how most software is developed in a modern business application. This led to the creation of a teaching model based on ten principles, namely, project-based (i.e., how the course is structured), cognition, adjustment, projection, connectivity, evaluation, listening, reflection, coaching, and the last ‘meta’ principle, inspiration. It organized these into a collection of practices – the Course, which determined the structure of the teaching model, the Role, or, how students were assigned within the group, Measurement, which included grading and student evaluation, Coaching, and Evaluation, the analysis of course results iteratively throughout the semester. Additionally, these practices were described by three ‘fundamental’ practices, or phases – Project-based, Adjustment, and Inspiration. This structure, in and of itself, also reflected the spirit of agile methodology, creating a tandem between the actual practice of software development and how software development was practiced by the students, and the learning-focused teaching model that was governed by those same agile principles. This iterative-loop style of teaching model is unique in its approach, taking a well-established practice in the business world of software development and applying it to the teaching of the same.

The second project addressed the issues and intricacies surrounding mentoring of software development in a high school environment. The research was divided into three stages: the first focused on the challenges faced by high school teachers in mentoring students in software development, both those with experience and those without. It categorized these challenges into four categories, or ‘themes’ – “schedule, required CS knowledge, students’ individual work, and project evaluation.” The second stage involved adaptations from what was learned in stage one, and the creation of the Agile Constructionist Mentoring Methodology, or the ACMM. The ACMM was based, again, on the Agile methodology in professional software development, and intended to be a set of guidelines based on those principles while remaining flexible and adaptable to most software environments and programming languages. It provides a time-based schedule of learning and achievement goals with interstitial assessment periods throughout the given term. The third and final stage was an evaluation of the ACMM using the four themes of challenges from the first stage. This had the effect of ensuring each of the challenges was addressed throughout the process.

In Fouh, et al. (2012), the theme is algorithm and program visualizations (AV and PV, respectively) and the importance of these to the teaching of computer science. They argue that static teaching tools – chalkboard, textbook, lecture – are an ineffective means of teaching the complex and abstract principles and details of programming, software development, and technical architecture. Many of these concepts require visualizations that cannot be represented statically, like phase changes in algorithms and programmatic runtimes, and need tools like animated presentations and interactive modeling to be taught effectively. The first part of the study details the history of visual presentations in computer science classrooms, enumerating such projects as the BALSA system from 1985, and MIT’s Project Athena from 1991, that each tried, but in some ways failed to gain a foothold in the broader education system due largely to problems with dissemination and available technology. While Project Athena was made available on the “X Windows Display System” to try to solve the dissemination issue, at the time too many college computer labs were too underpowered and lacked the hardware to run the program. The subsequent availability of languages like Java, JavaScript and Flash after the development of the World Wide Web in the 1990’s meant that visualization programs were no longer reliant on any individual system, but could be implemented in most environments.

The authors then go on to describe more recent assessments that have been done on the utilization of AV’s and PV’s in the classroom, detailing studies in which classrooms were divided into groups, some taught with visualization technology, some with standard classroom tools. The results were muddied by the fact that the measurement of the results is often an abstract goal, as in, ‘how well does an individual understand the processes of an algorithm’ or trying to determine if cognition was improved overall (improved, here, could also be an abstract term, given the subject material). The greatest impression of these studies, however, was that students found much greater success through the process of *engagement* with the material, rather than simply viewing. This led to a string of reports during 2003-2006 by the ACM Innovation and Technology in Computer Science Education conference that created six categories of student engagement in AV: no viewing, viewing, responding, changing, constructing, and presenting. Interaction was later added as a category. These layers of engagement provided a basis for understanding and implementing visualization tools for students, and subsequent studies showed that engagement had a much greater impact on student retention and understanding than mere viewing of material, and that viewing as effective as not viewing, which is to say, hardly at all.

The paper then outlines a collection of specific tools, and the studies done on them, such as the ViLLE “program visualization tool” used in Finnish universities for animated presentations of algorithmic state changes, et al., while also including interactive tests for the student. A study done on ViLLE found that, while improvement in students with prior CS knowledge was minimal as compared to a control group, gains in those without prior knowledge was significant. Jeliot, used to teach Java in high schools, used a student’s own program to create an animated visualization of that program (PV) to help the student see the structure and more abstract concepts behind it. An ensuing study found a drawback in that a student could be easily confused by different presentations of the model, and the programs inability to filter out complex detail from the visualization. Subsequent surveys of educators found that they, overall, had a positive view of visualization programs for teaching computer science, but implementation was lax, due in part to the challenges of integrating new technology into existing courses, the time investment, and the reticence of many instructors to be early adopters of new technology.

Overall, Fouh and friends believe the future is in electronic textbooks – online, interactive versions of their ancient counterpart, with animated visualizations and the potential for much more. Their ideal version would see a creative common of e-textbook creation with shareable, open source modules, and student participation in the process. In lieu of that, they propose the creation of complete AV/PV instructional blocks, rather than integrated components, which, they propose, will circumvent many of the issues discovered during the survey process.

The Wireless Intelligent agent Simulation Environment (WISE) developed by Cook, D., et al. (2004), was built to be a tool for interactive student learning that would be applicable to all aspects of computer science, intending to solve a problem that was observed with other interactive tools that only focused on individual aspects of CS. Specifically, WISE was a learning environment based on the ‘Wumpus World’ game – a turn by turn computer board game that blended a mix of strategy and chess-like moves. The game could be played simultaneously by autonomous bots and students alike, and provided students with a physical ‘avatar’ to be placed in the physical environment (like a gaming Roomba, essentially) to compete against or with the student. It ran in a Java environment in order to “support platform independence,” used a wireless network for communications between user agents, and responded to user interactions via a server/client connection that would load a new game state dynamically based on decisions made by the user.

The authors of the study stipulate that this type of project, bringing robotics, machine learning, video processing and network interaction, is well suited to studies of robotics, Artificial Intelligence (AI), wireless mobile computing, and multimedia. They argue that this multifaceted system would be flexible enough to adapt to different learning environments and topics, while also giving the students a more thorough hands-on experience with ‘real-world’ applications of computer science. The study conducted applications of WISE on three different CS classes: AI, multimedia computing, and wireless mobile networking. Students were then surveyed afterwards on metrics of understanding and ease-of-use to determine whether the tool had been both accessible and effective in increasing understanding of the given subject. On a scale of 1-5, difficulty tended towards the center (2.5) while the effect on learning ranged from 3.2 – 4.5 depending on the topic at hand. They also found an increased ratio of effectiveness the greater the difficulty of the subject matter.

While further work had been planned for the WISE program to be demonstrated and studied in additional subjects of computer science, this was the last publication authored by the team with little reference found later than 2004 when the study was originally published. It leads to belief the program may have died of natural causes, likely due to its complexity and difficulties in implementation, although this is pure conjecture.

Finally, our last literature review looks at a piece by David Patterson (2006), a notable professor at University of California, Berkeley. He argues four main points, two technological and two course propositions for future educators. Firstly, Patterson criticizes the current (then current) state of computer science education in the college classroom, that while they may teach more ‘modern’ languages like Java, they fail to quickly embrace useful tools and libraries like Eclipse or .NET. He suggests the modern student would be better served with better tools, and ones more in use and more relevant in the modern world. Second, he says that in the age of parallel processing, with single thread CPU’s giving over to multi-core processors, too much current curriculum is based in the single process world, and hasn’t adapted quickly enough to the new paradigm, putting students at a disadvantage when they encounter modern technological architecture in the business world. The first class he proposes (“course I would love to take”) is an open source development class – one that, rather than training students to build software from scratch (an unlikely real-world scenario), to build it using components of available open source components online, simulating how much enterprise software is built today. Tasks proposed include having students write documentation, debugging large systems, and suggesting new features. His second course proposal is to have students build their own ‘supercomputer’ with component parts and circuit boards, thus providing them real hands-on experience with computer architecture. Patterson concludes his thesis with the reiteration that classroom technology and courses are woefully behind current technology, and need more modern precepts as well as excitement brought to the fore.

**Discussion**

Drawing from Guzdial, M., & Robertson, J.’s (2010) article, “Too Much Programming Too Soon” as a counter-argument to the intuitive, or classical argument that novice CS students learn best through programming first. It suggests a “mixed bag” approach, teaching concepts and theory first while inoculating freshmen with small bits of hands-on practice. I take that approach in this discussion as well. Let’s talk about the two key challenges discussed in the introduction – student attraction and student retention, and how to solve for them.

I postulate that part of what causes dropouts to occur early on or dissuades new students from computer science entire is the fear of failure, and complexity of initial work requirements. Programming languages look strange and foreign to those not exposed to them previously, and forcing students to learn this peculiar new syntax right away for homework due in the first few weeks can create an imposing stumbling block to success, and drive away those with potential. I propose a completely redeveloped plan for computer science that would use introductory courses to ‘sell’ the field (computer science is exciting!), while gradually immunizing the new student with bits of complexity and syntax that are directly tied to the discussion.

The course structures should revolve, much like the case in Hazzan, O., et al.’s (2010) study, around agile methodology. Agile experts, working in the field, could be brought in on a regular basis to ensure practices are current and flexible enough to accommodate the subject material. In this spirit, the first few weeks of any course should reiterate fundamental aspects of computer science rather than making assumptions about the student’s knowledge, even at a more advanced level, due to the nature of agile sometimes being a flat circle. An example of this could be a, say, Software II course that for the first three weeks refreshes knowledge from Software I and Algorithms I before diving into more complex topics. The courses should be structured in such a way as to be divided into short 2-3 week *sprints*, to borrow an Agile term, with rigorous evaluation at the end of each sprint, and allowing for repeated iterations if necessary. This creates a short-term period of time in which students who are falling behind on the material can quickly be brought up to speed and needs can be reassessed.

As Fouh, et al. (2012) notes, interactive visualizations that engage the student are much more effective in teaching a complex subject than mere viewing, so this new proposed plan would need to be rich in both visuals and engagement. That would necessarily involve creating a cohesive, homogeneous system that would be familiar to the student from one subject matter to the next, so they wouldn’t need to learn a new set of engagement rules with each new topic. Part of an introductory course might simply get the student comfortable with the interactive system to give them a foundation for future courses. The visualization system would need to be flexible enough to adapt to different aspects of computer science – phase shift animation with a ‘choose your own adventure’ style of interactivity could be applied to algorithms, artificial intelligence, web development, and much more.

For this new program, the WISE system (Cook, D., Huber, J., Yerraballi, M., & Holder, R. (2004)) is inadequate and creates complications that, I believe, would hinder adoption in most universities. The concept is unique and attempts to use existing technology to create a more immersive learning environment, but the initial requirements of setup are prohibitive, and current technology (drones, VR, P2P networking, etc.) far outpaces what was available in 2004.

Finally, as Patterson (2006) recommends, software development classes in this new program should be constructed around real world methodologies and technology. It’s a challenge to stay current in this modern computing era, which is why a system should be set in place to evaluate the methods and materials being used at the end of every semester. Additionally, all software-related courses in the program should adopt the open-source mentality. Software is never created in a vacuum, so these courses should reflect that. For instance, an Introduction to Web Development class would, after the initial weeks of teaching the basics, have students create a collaborative piece of software online using tools like Node Package Manager to piece together complex components, and they could even propose their own.

**Conclusion**

The need for a well-educated, computer science literate workforce has become an absolute necessity today. But perversely, instead of a boom in enrollment in CS programs, the educational system is seeing a decline in both retention and enrollment. Piecemeal efforts have been made to change different aspects of technological education at the college level, but little has been done with the big picture problems. To forestall a looming economic crisis in the tech sector, we must draw on new and old ideas alike to completely reshape the current system. Computer science programs shouldn’t merely be modified, but torn out by the root and replaced. It isn’t enough to rely on twentieth-century methods of teaching, not for something as radical and complicated and constantly changing as computer science. We have to think forward, and fast.

References

Bureau of Labor Statistics, U.S. Department of Labor, Occupational Outlook Handbook,

2016-17 Edition, Software Developers. (2015, December 17). Retrieved September 28, 2017, from https://www.bls.gov/ooh/computer-and-information-technology/software-developers.htm

Becerra-Fernandez, I., Elam, J., & Clemmons, S. (2010). Reversing the landslide in

computer-related degree programs. Communications of the ACM, 53(2), 127-133.

Giannakos, M., Pappas, N., Jaccheri, I., & Sampson, O. (2017). Understanding student retention

in computer science education: The role of environment, gains, barriers and usefulness. Education and Information Technologies, 22(5), 2365-2382.

Hopcroft, J. (1987). Computer science: The emergence of a discipline. Communications of the

ACM, 30(3), 198-202.

Ghosh, D., Sheehy, J., Thorup, K., & Vinoski, K. (2012). Programming language impact on

the development of distributed systems. Journal of Internet Services and Applications, 3(1), 23-30.

Hazzan, O., Dubinsky, Y., & Meerbaum-Salant, O. (2010). Didactic transposition in computer

science education. ACM Inroads, 1(4), 33-37.

Fouh, Eric, Akbar, Monika, & Shaffer, Clifford A. (2012). The Role of Visualization in

Computer Science Education. Computers in the Schools, 29, 95-2, p.95-117.

Cook, D., Huber, J., Yerraballi, M., & Holder, R. (2004). Enhancing computer science

education with a wireless intelligent simulation environment. Journal of Computing in Higher Education, 16(1), 106-127.

Patterson, D. (2006). Computer science education in the 21st century.

Communications of the ACM, 49(3), 27-30.

Guzdial, M., & Robertson, J. (2010). Too much programming too soon? Communications of

the ACM, 53(3), 10-11.