Where To From Here?

Our curriculum is not up to the challenges the world is throwing at us.

Peter J. Denning, *Naval Postgraduate School*

When I received my doctorate in EE from MIT in 1968, the name of the young field of computer science was just getting settled. I was amazed at the audacity of the dreams of the founders and pioneers. I was completely drawn in and I developed a romance with computing that has never faded.

I have had the good fortune to witness the growth and maturing of this field of education and research for the entirety of SIGCSE's fifty-year existence. My purpose here is to look at the high points of computing and computing education over the past fifty years. Several major forces shaped the computing curriculum we have today. Our curriculum is not up to the challenges the world is throwing at us.

SHAPING FORCES OF COMPUTING

Our modern age of electronic computing began in the late 1930s and spawned computing education in the late 1940s. Computing in the sense of methods and machines to automate calculation and logical deduction is much older—it evolved over at least 40 centuries before our age. Prior developments such as procedures for doing algebra, solving equations, evaluating series, Pascal's arithmetic calculator, Napier's logarithms, Newton-Leibniz calculus, LLull's logic wheels, Babbage-Lovelace analytical engine, slide rules, and human calculator teams set the framework of computational thinking that existed when computer science was founded in the 1940s. I will focus here on the main historical forces that shaped our field and how we approached our curriculum since that time.

Machinery and systems. The first electronic digital computers were built in the 1930s and early 1940s—Zuse in Germany in 1938, Atanasoff and Berry in the US in 1942, Eckert and Mauchly at Pennsylvania in 1945. All were engineers who believed that high speed computing would benefit science and engineering and would automate many human computational tasks that were prone to errors.

Their machines were great feats of engineering. They had to work out everything—how to represent data as signals in the machines, how to build reliable circuits that would perform logic operations on the data, how to store data, how to get data in and out of the machines, and how to design algorithms that would control the machines. There was no theory to guide them.

Although Alan Turing proposed his Turing machine theory of computation in 1936, his work was initially known primarily by a handful of mathematical logicians and was completely unknown to the engineers who built the first electronic computers [6]. Turing became more known among computer builders when he circulated his own detailed engineering design of his ACE computer, inspired by von Neumann's notes on the design of the stored program computer in 1945. It was not until the 1950s, when the first academic programs were being born, that Turing's work offered the theoretical basis to make computer science credible as a new department in universities. In other words, as important as Turing's work is, it did not inform or inspire the first electronic computers or the stored program concept. Instead, the success of the first stored-program electronic computers created the opening for Turing's theoretical work to become important.

For the first 40 years of computing, much of our energy was focused on advancing the technology for reliable computing and networking. Our early curricula reflected this by organizing around core technologies, such as programming languages, operating systems, and networks. The 1989 computing report named 9 core technologies [2]. The 2013 curriculum report named 16 core technologies among its 18 main knowledge areas [1]. Today's curriculum bears the imprint of the engineering concerns that started the field.

By 1982, there were about 120 departments in the US and Canada. Most of these early departments were formed amidst resistance from other departments in their universities, which saw computer science as a specialty of math or electrical engineering, but not as a separate department.

Academic Resistance. The first computing and programming courses appeared in the late 1940s. The first CS departments were Purdue and Stanford both in 1962. By 1982, there were about 120 departments in the US and Canada. Most of these early departments were formed amidst resistance from other departments in their universities, which saw computer

science as a specialty of math or electrical engineering, but not as a separate department. Most early departments were therefore founded within the most hospitable school—some in science, some in engineering, and few in business. The academic pioneers of the day—notably Forsythe, Newell, Perlis, Simon—spent a great deal of effort defending the new field against skeptics who thought it was neither a new field nor deserving to be called a science [5,7]. Their ideas guided the early development of computer science education. Because most CS departments were in schools of science or engineering, the term CS&E (computer science and engineering) became the collective term for all the departments. After 1989, at the recommendation of the ACM/IEEE computing-as-discipline committee, the term "computing" was used instead of "CS&E" [2]. Europeans preferred the name "informatics." By 2000, the resistance was pretty much gone as biology, physics, aeronautics, and other fields declared they dealt with natural information processes.

Formal methodists see mathematical proof as the only means to establish that software is error-free. Software engineers see formal methods as one tool of many—other tools are needed to address defects in hardware, deterioration of hardware, error confinement, and detection of malware.

Software Engineering. In 1968-69, software developers called for a new field, software engineering, because the existing approaches to software development were not able to take care of concerns for dependable, reliable, usable, safe, and secure (DRUSS) production software. The founders of software engineering believed that engineering perspectives such as fault tolerance, redundancy, and interface design could help. CS departments responded by setting up a software engineering course and over time some developed "tracks" containing several courses. Some universities set up a separate IT curriculum, sometimes as a track in the CS department and sometimes as a stand-alone department. In a few rare cases, software engineers formed their own departments where all the faculty could engage with engineering perspectives without being constrained by the more abstract ways of CS departments.

Networks. In the late 1970s many CS departments were unhappy that only a few of them were connected to the ARPANET, which was restricted to defense contractors. They banded together and won National Science Foundation support to design and build CSNET (computer science network). I was one of the four co-PIs. By 1986 we had adapted ARPANET technology and built a CS research community network of 50,000 users at 120 member institutions. The network significantly increased research productivity in all participating CS departments. CSNET gave NSF confidence it could manage a large network project, NSFNET, which became the backbone of the modern internet. Around 1989, ARPANET, CSNET, and NSFNET were decommissioned because everyone got the networking they needed from the internet.

Computational Science. Scientists had a long-standing interest in computation well before the 1940s. By the 1950s, with help from numerical analysts, they were using electronic digital computers for calculating prediction from complex models and for analyzing experimental data. By the 1980s they had gone well beyond this—they had developed their own brand of computational thinking that they saw as a new method of doing science. CS researchers responded with mixed reactions. Some embraced computational science as a welcome expansion of computing; others resisted it as the work of amateurs with little experience with computing. Eventually, CS researchers and educators overcame their reluctance and embraced it. Computational science appeared as a core area in the 2013 curriculum.

Formal Methods. A very large debate opened in the 1980s about the power of formal methods to give us reliable and dependable software [9]. Proponents called it the most important of all computing research. Skeptics cited all sorts of reasons that mathematical proof was insufficient for the DRUSS software objectives. The sharp words from all sides eventually quieted down, but the underlying tension endured. It is the same tension as between the traditional math-science oriented computer scientists and the software engineers. Formal methodists see mathematical proof as the only means to establish that software is error-free. Software engineers see formal methods as one tool of many—other tools are needed to address defects in hardware, deterioration of hardware, error confinement, and detection of malware. I wish the tension would go away, but it is still there. The two views are complementary and mutually reinforcing.

Artificial Intelligence and Machine Learning. AI was founded in 1954 in pursuit of the goal of general machine intelligence. It looked at language translation, image recognition, checkers, chess, problem-solving, robots, expert systems, and machine learning as steps on the way to that goal. By the mid-1980s the field had delivered so little of its big promises that the research funding agencies began to withdraw their support, precipitating the long doldrums dubbed "AI winter." The controversial 1987 book *Understanding Computers and Cognition* [11] argued that the quest for machine intelligence was fatally flawed and proposed that we devote our energy to designing machines that support human practices. That advice became the key to a resurgence of machine learning (ML) in the 2000s, when researchers found that they could apply the technology

of neural networks to large classes of pattern recognition and prediction, with astonishing success. In recent years Machine Learning and Big Data Analytics (BDA) have come to depend closely on one another. Referring to the success of ML and BDA as "AI" is a misnomer because neural networks are unintelligent pattern recognizing machines.

Parallel and distributed computing. From the 1960s computer scientists developed strong interests in computations performed by many processors working together. Because of their significantly greater speed, parallel processors became the mainstay of supercomputing, which became very popular in science, engineering design, medical and drug research, entertainment, and more, and fomented the revolution of computational science. Also, from the 1960s, computers distributed into many physical locations connected by a network provided access to remote services and showed great resiliency to failures. The 1960s dream of computer utility matured into the modern distributed systems making up "the cloud." This long line of developments left a permanent imprint on our curricula.

The idea that computing is universally valuable pervaded the thinking of the founders of computer science. Beginning in 1960, pioneer Alan Perlis repeatedly said that computer automation would spread to many fields ...

EVOLVING VIEW OF WHAT COMPUTING IS

The question of whether computer science was unique or was a science persisted for many years. Our views of what computing is evolved through four stages over the years [4].

- In 1960s, we said we studied phenomena surrounding computers.
- In the 1970s, we said we studied programming and all that entailed about algorithms, analysis, and correctness.
- In the 1980s, we said we studied automation, what could be efficiently automated by digital computers.
- In the 1990s, as other fields of science started to claim their fields included naturally occurring information processes, we said we studied information processes natural and artificial. After that, the old debate about whether computer science is science, or deserves its own academic department, completely faded.

The curricula we taught evolved along with these maturing views of the nature of the field. In 1968, ACM produced its first curriculum recommendation, the first attempt at standardizing what a computer science degree meant. In 1972 the NSF-sponsored COSINE (computer science in engineering) project advocated placing systems courses in the core curriculum, alongside the traditional math courses already there; operating systems was the first systems course to be accepted into the CS core. In 1989 the ACM and IEEE Computer Society (IEECS) cooperated on the first joint recommendation that emphasized the integration of theory, abstraction, and design—representing math, science, and engineering—in the core. They sought to ease the tensions between these three subgroups of the field. They began calling the field "computing" rather than "computer science and engineering." Those ideas dominated the 1991 ACM/IEEE computing recommendations. ACM and IEEECS continued their cooperation and produced major updates in 2001 and 2013. The growth of the field can be seen in the increasingly complex recommendations over the years. The 1968 curriculum had three major subdivisions; the 2013 curriculum had 175.

In recent times, our virtual machine technologies and platforms have improved so much, and chips and sensors shrank so much, that most designers of software are seldom aware of hardware. Some educators have argued that we no longer need to be concerned about hardware; we should drop our insistence that algorithms and software are intended to control machines. Instead, we should view algorithms and software as expressions of methods to solve problems that can be shared and communicated with others, a view that dominated the design of the ALGOL language in the 1950s. In his 1968 ACM A.M. Turing lecture, Richard Hamming took a dim view of the idea that we can abstract the machine out of the picture. He argued that the computer is at the heart of computing; without it, almost everything computing professionals do would be idle speculation. Hamming's insight remains valid today: there can be no computing without computers.

Along the way there has been an ongoing debate about what programming language(s) to use in CS courses. Should they be languages used heavily in industry, such as C, Java, or Javascript? Or languages designed for easy learning of basic programming concepts, such as Pascal or Python? It is ironic that on the one hand computer languages are equivalent in expressive power, while on the other hand language choice is the most fiercely debated issue in teaching computing. This debate is unlikely to end.

QUEST FOR COMPUTING EVERYWHERE

The idea that computing is universally valuable pervaded the thinking of the founders of computer science. Beginning in 1960, pioneer Alan Perlis repeatedly said that computer automation would spread to many fields and draw many people into "algorithmizing"—his term for what we now call computational thinking.

Computing educators became interested in the 1970s in bringing computing's general-purpose thinking tools into K-12 schools. That was a major challenge: few schools had teachers with computer science knowledge. Computer literacy was seen by many as a gentle first step toward getting computer courses into grade schools. The first attempts at literacy courses were little more than training in how to use word processors and spreadsheets. They were not popular with students or teachers. A turning point came in 1999, when a task force of the National Academy of Engineering issued a report reframing the goal from literacy to fluency. Larry Snyder, the chair of the task force, wrote *Fluency in Information Technology*, a textbook that became popular with high school teachers [8].

We arrive at our 50th anniversary of the founding of SIGCSE with a curriculum specifying what (and how) we teach computer science, a curriculum that evolved over half a century.

Also in the 1990s, the College Board became interested in an upgrade to the advanced placement test in computer science. With help from ACM and IEEE, they launched around 2000 an advanced placement (AP) curriculum focused on object-oriented programming with the Java language. Within a few years, AP enrollments plummeted as students and teachers discovered the material was too complex for beginners. The College Board, in cooperation with the US National Science Foundation, undertook a new advanced placement curriculum organized around computer science principles, which it hoped would provide a better return on their investment. The upgrade was rolled out in 2016.

In 2006 Jeannette Wing reframed the issue again around "computational thinking" (CT) which she characterized as the thought processes that computer scientists used to solve problems [10]. This formulation resonated with many people who saw computing permeating into their fields and wanted to learn how to harness the technology. As an assistant director for the Computing & Information Science & Engineering (CISE) directorate at the National Science Foundation, Wing mobilized many people and resources around the goal of getting a computing curriculum based around computational thinking into every K-12 school. They sought to train 10,000 teachers in computer science. They supported the development of the CS principles advanced placement curriculum and concurrently the development of a new genre of CS principles first courses in universities. Many organizations stepped up to define K-12 curricula around computational thinking, producing several proposals for teachers to choose [3].

Even with all this backing the new curricula have been slow to find their way into K-12 schools and some of the teachers are still concerned about what they should teach and how to assess whether students have learned it.

The definitions of CT in these proposals are quite narrow compared to the breadth of pressing computational issues in the world—they do not apply to complex systems, reliability concerns, hardware, or emerging technologies such as quantum computing. CT is not the defining characteristic of computer science. Neither is it "the way of thinking of computer scientists" because many in other fields have contributed significantly to our understanding of computation.

EDIFYING CONVERSATIONS ON BIG QUESTIONS

We arrive at our 50th anniversary of the founding of SIGCSE with a curriculum specifying what (and how) we teach computer science, a curriculum that evolved over half a century. The specification was shaped by many factors noted here.

- Strong emphasis on building technologies at the beginning
- Resistance to forming CS departments from other academic departments that did not accept computing as a legitimate field
- Developing our own community network at the dawn of the internet era
- Being torn by intense debates over the roles of science, math, and engineering in our field, manifested as struggles over how to teach software engineering and information technology, and how much to trust formal methods for software development
- Coming to grips with the emergence of computational science and now the penetration of computing into nearly every field of human endeavor
- The death of artificial intelligence and its resurrection as machine learning and its claims about automation and the future of humanity

This battle-hardened inheritance does not help us with many of the pressing issues of the world emerging around us. The worldwide connectivity we helped bring about through the Internet has brought many benefits from shrinking the world and globalizing trade. But it has also spawned conflicts between non-state organizations and traditional nations, trade wars, protectionism, terrorism, widespread detachment, fake news, political polarization, and considerable unease and uncertainty about how to move in the world. Access to troves of information via the internet has begun to show us that knowledge does not confer wisdom, and we long for wise leaders who have yet to appear. The world we encounter in our daily lives is full of surprises, unexpected events, and contingencies that not even our best learning machines and data analytics can help us with. We are now finding that many resources including sea and air access are contested among nations; we lack means to resolve the resulting disputes and we worry that the resulting conflicts

[H]ow shall we shape computing education so that our graduates can develop the design sensibilities, wisdom, and caring they will need to navigate in this world of which they will be citizens? Our current curriculum, chock full of courses covering the 2013 body of knowledge, is not up to this task.

could trigger wars or economic collapses. We see that collective human action affects the global environment but have yet to find ways to protect the environment we will bequeath to our children and grandchildren.

This leaves us with a big question—how shall we shape computing education so that our graduates can develop the design sensibilities, wisdom, and caring they will need to navigate in this world of which they will be citizens? Our current curriculum, chock full of courses covering the 2013 body of knowledge, is not up to this task.

A place to start would be to open space in our crowded curriculum to have conversations on big questions about the consequences of computing throughout the world. These conversations need to be interdisciplinary and intergenerational. Their purpose would not be to solve problems but to edify—develop mutual understanding, appreciation, and respect around these issues. Some examples of big questions are:

- How far can automation take us? Can everything be automated? Is there always something important left over that cannot be automated?
- Will AI displace more jobs through automation than it generates?
- How can we help people whose jobs are displaced by software and hardware we have designed?
- How do we cultivate good designers?
- Can we trust decisions by neural networks when given inputs outside their training sets?
- Will drones and robots combine to create an automated surveillance society?
- Is there a technological solution to the cybersecurity problem?
- Can we make our world work when computers have been embedded into almost all devices connected to the global network?
- Can blockchains and cryptocurrencies solve our problems with trust in central authorities? Are they too expensive to maintain?
- Is civilization so dependent on computing that an attack on a component of infrastructure, like electric grid, could collapse civilization?
- What is the difference between wisdom and knowledge? How are we fooled into thinking that massive internet information is wisdom?
- What are the social implications of brain-computer interfaces and implants into our brains and bodies?

I do not believe any of us has answers to any of these questions. But we need to be having the conversations about them. In so doing we need to embrace the mathematicians, scientists, and engineers in our field. It is time to give up the old tensions that we inherited from times long past, and work together as brothers and sisters, mothers and fathers, old and young on these big questions. ❖

Acknowledgements

I thank Matti Tedre for historical and technological insight in our conversations about the topics discussed here. I also thank Fernando Flores for giving the name "edifying" to the kind of conversation we need more of in education, and for edifying conversations to better understand the world our technology has shaped.

References

1. ACM. *Computer Science Curricula 2013*; https://www.acm.org/binaries/content/assets/education/cs2013_web_final.pdf. Accessed 15 Feb 2018.
2. Denning, P., Comer, D.E., Gries, D., Mulder, M. C., Tucker, A., Turner, A. J, Young, P. R. Computing as a discipline. *Communications of ACM* 32, 1 (1989), 9–23.
3. Denning, P. Remaining trouble spots with computational thinking. *Communications of ACM* 60, 6 (2017), 33–39.
4. Denning, P. and Martell, C. *Great Principles of Computing*. (MIT Press, 2015).
5. Forsythe, G. E. (1968). What to do till the computer scientist comes. *American Mathematical Monthly* 75 (May 1968), 454–461.
6. Haigh, Thomas. Actually, Turing did not invent the computer. *Communications of ACM* 57, 1 (2014), 36–41.
7. Newell, A., Perlis, A. J. and Simon, H. A. Computer science. *Science* 157, 3795 (1967), 1373–1374.
8. Snyder, L. *Fluency with Information Technology: Skills, Concepts, and Capabilities*, 7th Edition (Pearson, NY, NY, 2017).
9. Tedre, M. *The Science of Computing: Shaping a Discipline*. (CRC Press / Taylor & Francis, New York, NY, USA, 2014).
10. Wing, Jeannette. Computational thinking. *Communications of ACM* 49, 3 (2006), 33–35.
11. Winograd, T. and Flores, F. *Understanding Computers and Cognition*, (Addison-Wesley Professional, 1987).

Peter J. Denning
Naval Postgraduate School
Monterey, California, 93943 USA
pjd@nps.edu

Peter J. Denning (pjd@nps.edu) is Distinguished Professor of Computer Science and Director of the Cebrowski Institute for information innovation at the Naval Postgraduate School in Monterey, CA. He is Editor of ACM *Ubiquity* and is a past president of ACM. The views expressed here are his alone and are not necessarily those of his employer or the U.S. Federal Government.

DOI: 10.1145/3191833