The Arbitrary Nature of Computing Curricula

Computing is still a young discipline with new topics emerging daily, spawning an extended family of disciplines, which makes negotiating a curriculum an inherently fraught process that will not meet everybody's needs.

By Tony Clear

DOI: 10.1145/3265905

Any academic discipline is by nature a rather arbitrary thing. It is shaped by key leaders who define professional or curriculum boundaries that selectively address the topics of the area. They prescribe what is in and what is out, and thereby serve to exclude many topics and groups of people.

For instance the ACM/IEEE curriculum for computer science reflects a broad range of CS topics, but only a very narrow view of computing [1]. Computing is still a young discipline with new topics emerging daily. It could be better thought of as a family of disciplines, which encompass not only the topics that are the focus of academics, but those that are the focus of professionals in the field.

For me the idea of a "computing discipline family" is a more productive and inclusive way of answering the questions "what is a computer scientist?" and "what is the curriculum that should be taught?" In an attempt to answer those questions here, I will take you through some of my own experiences in education, in industry as a computing professional, and in academia as an educator, researcher, and developer of computing courses and curricula.

The term "impostor syndrome" is given to the case where people don't feel they really belong in a role, or are not as expert in their field as those around them may think, and live in fear of being caught. Many of us in computing, with its often overly critical mindset, suffer from impostor syndrome. So if you feel a bit excluded or ignorant at times, don't worry—the field is enormous and ever changing. You will never know everything. And those who are insecure enough to have to boast about their prowess and arrogantly put others down, probably don't really know that much. It's merely important to be open to learning and be able to acknowledge what you don't yet know.

My own career in computing has been atypical. I began with an undergraduate arts degree in Latin and English language, I then went on to study for my master's degree. After a period in high-school teaching, I went into industry in 1979, being trained through a combination of block courses and in-house training as a COBOL programmer and systems analyst. I then took on progressively more senior roles in software development. Besides COBOL, we used languages such as TPS and MPS (effectively assembly languages in small Olivetti data-capture terminals, octal machines with 1.5KB of programmable memory); then LIMO, the assembly language for the Olivetti banking terminals that had 24KB memory in addition to the operating system routines, which you had to be careful not

to overwrite when you wrote your code; and then CREDIT (a combination of a COBOL and assembler type language) for a later Philips version of the terminal controllers. We used ICL machines with a version of the IBM 360 assembler, Fujitsu and IBM mainframes, CICS-COBOL, as well as various file types, databases, networks, and protocols. I managed software developers writing code supporting packaged software and system programmers supporting operating systems. I also became embroiled in a major project failure aiming to replace our banking system, which gave me prematurely grey hair and a great understanding of runaway projects, including how to rescue some dignity from the ashes. After the *n*th restructure and change of ownership at the bank I then worked for, I decided I was not a banker and joined academia, where my industry skills and management experience were valued. But even so, I had to start from scratch and study for a further master's and then Ph.D. in computer and information sciences to become credentialed as an academic. In today's academy, I would probably not be hired!

So my views on computing curricula originate from a hybrid practitioner and pragmatic perspective, and tend not to see the hard line that academia has historically drawn between computer science and information systems disciplines, largely based on their origins in engineering and business schools respectively.

The ACM curriculum overview report in 2005 did a nice job in depicting a continuum of engineering and computing disciplines from those closer to the hardware and the machine (electrical engineering and computer engineering), those in the middle (computer science and software engineering), and those closer to the organization and the people (information technology and information systems) [2]. However, I tend to disagree with the report's depiction of information technology, as a bit muddled (given the nature of networking for instance as close to the machine), and presenting a very U.S.-centric perspective. In Australia and New Zealand for instance, information technology is more of an umbrella term for the industry, not a subset of a discipline. But with the continual growth in computing curricula and expansion of the discipline (e.g. new curricula for data science and cybersecurity), a new version of the overview report is underway, due to come out in 2020.

Defining the discipline of computing (not to mention gaining agreement on any definition) is very difficult. In his 2014 book, *The Science of Computing: Shaping a Discipline,* Matti Tedre noted the tripartite origins of computer science drawing simultaneously from science, mathematics, and engineering [3]. But it can be more widely viewed too. In a 1997 report on historical perspectives on the computer science curriculum, we talked of computing from a multiplicity of perspectives [4]. These further included: computing as literature and computing as an artistic endeavour, computing as a social science, anthropology and computing, computing as politics, and computing as interdisciplinary. These may seem a very broad collection, but as we see the scope and range of computing related disciplines grow, deciding what is core computing becomes harder.

At my own institution, Auckland University of Technology in New Zealand (which follows the U.K. three-year bachelor's degree model, and a four-year Bachelor of Engineering with Honours degree) we have a broad undergraduate degree in computer and information science. That includes majors in computer science, software development, computational intelligence, networks and security, IT service science, and analytics. I was involved in creating the curriculum for most of these majors starting in 2001. Initially we worked with the late Professor John Hughes of UTS in Sydney—a wonderful scholar, colleague, Ph.D. supervisor, and friend—who was for a time our Head of School, and widely versed in computing curricula from his Australian and global experiences. The process involved reviewing other curriculum models; identifying trends in the discipline and profession; relating those to our research and teaching strengths; liaising with our industry advisory committee to align our directions with pressing needs, and to see what the demand for such graduates would be; determining with colleagues the desired graduate profiles for each major; and charting a suitably challenging and tailored sequence of core and major specific courses.

We need to be aware of the considerable power that lies in the hands of a computer scientist or software engineer, and the need to responsibly wield that trust.

As a university of technology, our courses tend to have a stronger practical dimension and closer industry alignment than other universities may choose. However, John and I were both comfortable with that and sought to have our university produce employable and productive professional graduates with the awareness, adaptability, and insight to make a broader and ongoing contribution to society. But our research strengths were also drivers for curriculum initiatives, with the computational intelligence major later added, reflecting the school's strengths in knowledge engineering,[1] as well as various forms of artificial intelligence and a software engineering major added into the Bachelor of Engineering (Hons) degree.

Research interests have tended to more strongly drive the postgraduate curriculum though. The initial conception of our master's degree (originally a Master of Information Technology, and now a Master of Computer and Information Sciences) was to combine technical up-skilling and professional tracks, underpinned with a strong research dimension. We aimed to cater to students wishing to move up in their professional careers into leadership roles, or higher research studies. The professional track therefore included managerially focused courses such as "Information Technology Strategy and Policy" and "Service Relationship Management." Research methods was a core initial course, where we aimed

1 www.kedri.aut.ac.nz

to enable our students to read and understand the literature, the research process, and the wide range of different approaches to undertaking research in the computing field. Many courses had a "CS-plus-x" flavor reflecting the domains in which our professors conducted their research: geo-informatics, bioinformatics, neuro-informatics, health informatics, artificial intelligence and robotics, nature inspired computing, IT security, data warehousing, data mining, requirements engineering, and user-centred design. In my own case, I developed a course in collaborative computing.

Many of these courses significantly expand on a narrow vision of computer science, address the needs of professionals in the field, and reflect the wider and expanding family of subdisciplines. We have since created additional and more specialized master's degrees in digital forensics, service oriented computing, and health informatics. But I can see this being an ongoing debate, whether specialized or more general postgraduate degrees have more merit. Of course there is an accompanying debate over whether more broadly educated or specifically trained graduates have greater merit. There is much talk now of T-shaped individuals, i.e. those with a depth of expertise in one area (e.g. software design) complemented by a breath of perspective across many areas (e.g. user experience design, requirements engineering, negotiation, ethical awareness, technical writing, test driven development, product strategy, domain knowledge, release planning, estimating, costing, and business case development). In a way, this echoes current political debates over the merits of increasing the number of science, technology, engineering, and mathematics (STEM) discipline graduates, over and above those from the humanities.

On this topic, based on my own education, I clearly favor a hybrid approach, but I grew up in an era when I had the luxury of being able to make such a choice. University education in New Zealand's egalitarian society was then largely free to those with the ability to study, and only some 5 percent of the population went to university. While attending this year's International Conference on Software Engineering (ICSE), it was especially gratifying to hear Margaret Hamilton—a pioneering software engineer who wrote the safety critical code for NASA's early space missions—recall that among the varied people NASA employed were several philosophers and artists who made wonderfully creative programmers.[2]

As we see the scope and range of computing related disciplines grow, deciding what is core computing becomes harder.

In concluding these reflections, I turn to the debate at the recent International Conference on Global Software Engineering (GSE): Is there a continuing need for a specialized conference on GSE? The arguments revolved around whether GSE was now the new normal for all software engineering. But one theme that came through strongly was the need to consider the people aspects in computing in a global context, and to what extent the wider software engineering discipline had fully grasped that point. Tom De Marco's 1987 book (now in its third edition) addressed this issue in software engineering directly, coining the term "peopleware," so the notion that people are important in software is far from new [5]. To think about it simply, we develop software with people in teams, and we develop software to serve the needs of people. A wholly technically defined science of computing that omitted this critical reality would be a dismal one indeed and would carry its own dangers. We need to be aware of the considerable power that lies in the hands of a computer scientist or software engineer and the need to responsibly wield that trust.

Already we are seeing challenges to the technically defined business models of hugely powerful tech companies like Google, Facebook, and Amazon. New technologies are raising increasingly thorny ethical and privacy issues, which will constrain what they may do and even challenge their right to exist. One could, for instance, argue their huge data repositories should be handed over to a neutral third party to curate. Data access could be allowed by data guardians only on a permissions-based model, where the users have the right to decide how to share the sensitive data that carries traces of their everyday lives. Such sharing could also come with a micro-payment option, for each fragment of personal data shared with a tech behemoth, so the value derived is shared more equally.

A computing curriculum that still develops the needed technical capabilities, but with a much stronger focus on philosophical, ethical, cultural and human concerns may well be what is needed to produce tomorrow's societally acceptable computer scientist (of whatever flavor).

2 http://bit.ly/2Ml1B4A

References

[1] Joint Task Force on Computing Curricula, ACM and IEEE Computer Society. *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*. 2013.

[2] ACM/IEEE-CS Joint Task Force for Computer Curricula. *Computing Curricula 2005: The Overview Report*. 2005.

[3] Tedre, M. *The Science of Computing: Shaping a discipline*. CRC Press, Boca Raton, FL, 2014.

[4] Goldweber, M. et al. *Historical perspectives on the computing curriculum - Report of the ITiCSE'97 working group on historical perspectives in computing education*. In *The Supplemental Proceedings of the Conference on Integrating Technology into Computer Science Education: Working group reports and supplemental proceedings* (ITiCSE-WGR '97). ACM, New York, 1997, 94-111.

[5] DeMarco, T. and Lister, T. *Peopleware: Productive projects and teams*. Addison-Wesley, Boston, 2013.

Biography

Tony Clear is an associate professor within the School of Engineering, Computer and Mathematical Sciences at Auckland University of Technology. His research interests are in computer science education, global software engineering, collaborative computing, and global virtual teams. He holds positions as an associate editor for *ACM Transactions on Computing Education (TOCE)*, *Computer Science Education and ACM Inroads* (for which he is also a regular columnist). He is active in research within the software engineering and computer science education communities. Tony has chaired or served on the programme committee for conferences such as ICGSE, EASE, ITiCSE, ICER, ACE, FIE, LaTiCE, CITRENZ, APRES, ECIS, and SIESC, and reviewed for journals such as *TSE*, *IST*, *JSEP*, *IJEE*, and *CLEIej*. He supervises and has examined doctoral students in global software engineering and CS education topics, and has chaired or participated in several doctoral consortia.