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 “imposter 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 imposter 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 programable 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 and a great understanding of runaway which gave me prematurely grey hair aiming to replace our banking system, and system programmers supporting code supporting packaged software and managed software developers writing databases, networks, and protocols. I

COBOL, as well as various file types,

Fujitsu and IBM mainframes, CICS-

nal controllers. We used ICL machines for a later Philips version of the termi-

COBOL and assembler type language)

and then CREDIT (a combination of a
to overwrite when you wrote your code;

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.

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, bio-informatics, 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 sub-disciplines. 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 breadth 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.

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. 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 that the 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).

References

Biography
Tony Clear is an associate professor within the School of Computing 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 committees for conferences such as ICSE, EASE, ITiCSE, ICSR, ACE, FIE, LaFEE, CITRENZ, APRES, ECIS, and Siemens, and reviewed for journals such as TSE, IJiT, JSEP, UJE, and CELIN. He supervises and has examined doctoral students in global software engineering and CS education topics, and has chaired or participated in several doctoral consortia.

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

2 [Link](http://bit.ly/2M1b4A)