DOI:10.1145/2998438

Peter J. Denning

Viewpoint
Remaining Trouble Spots with Computational Thinking

Addressing unresolved questions concerning computational thinking.

IMAGE BY OMELCHENKO

COMPUTATIONAL THINKING HAS been a hallmark of computer science since the 1950s. So also was the notion that people in many fields could benefit from computing knowledge. Around 2006 the promoters of the CS-for-all K–12 education movement claimed all people could benefit from thinking like computer scientists. Unfortunately, in attempts to appeal to other fields besides CS, they offered vague and confusing definitions of computational thinking. As a result today's teachers and education researchers struggle with three main questions: What is computational thinking? How can it be assessed? Is it good for everyone? There is no need for vagueness: the meaning of computational thinking, evolved since the 1950s, is clear and supports measurement of student progress. The claims that it benefits everyone beyond computational designers are as yet unsubstantiated. This examination of computational thinking sharpens our definition of algorithm itself: an algorithm is not any sequence of steps, but a series of steps that control some abstract machine or computational model without requiring human judgment. Computational thinking includes designing the model, not just the steps to control it.

Computational thinking is loosely defined as the habits of mind developed from designing programs, software packages, and computations performed by machines. The Computer Science for All education movement, which began around 2006, is motivated by two premises: that computational thinking will better prepare every child for living in an increasingly digitalized world, and that computational thinkers will be superior problem solvers in all fields.

Since 2006 hundreds of educators have participated in workshops, studies, committees, surveys, new courses, and public evaluations to define computational thinking for "CS for all" curricula. The Computer Science Teachers Association issued an operational definition in 2011 (see Box 1), the Computing at School subdivision of the British Computer Society followed in 2015 with a more detailed definition (see Box 2), and the International Society for Technology in Education followed in 2016 with a generalized technology

Box 1.
Computer Science Teachers Association's Concepts of Computational Thinking:[4]

Formulating problems for computational solution

Logically organizing and analyzing data

Abstractions including models and simulations

Algorithmic thinking

Evaluation for efficiency and correctness

Generalizing and transferring to other domains

Supported by: dispositions of confidence in dealing with complexity, persistence with difficult problems, tolerance for ambiguity, open-ended problems, communication and collaboration

Box 2.
Computing at School's Concepts of Computational Thinking:[3]

Logical reasoning

Algorithmic thinking

Decomposition

Generalization

Patterns

Abstraction

Representation

Evaluation

Supported by: techniques of reflecting, coding, designing, analyzing, and applying

Box 3.
ISTE's Standards for Students in Computational Thinking:[15]

Leverage the power of technological methods to develop and test solutions

Collect data

Analyze data

Represent data

Decomposition

Abstraction

Algorithms

Automation

Testing

Parallelization

Simulation

Supported by: empowered learner, digital citizen, knowledge constructor, designer, communicator, collaborator

definition (see Box 3). There are other frameworks as well.[21,27]

Given all this work, I was surprised recently when some teachers and education researchers asked for my help answering three questions with which they continue to struggle:

1. What is computational thinking?
2. How do we measure students' computational abilities?
3. Is computational thinking good for everyone?

To support my answers, I reviewed many published articles. I learned that these three questions are of concern to teachers in many countries and that educators internationally continue to search for answers.[21]

It concerns me that teachers at the front lines of delivering computing education are still unsettled about these basic issues. How can they be effective if not sure about what they are teaching and how to assess it? In 2011, Elizabeth Jones, then a student at the University of South Carolina, warned that lack of answers to these questions could become a problem.[16]

I believe the root of the problem is that, in an effort to be inclusive of all fields that might use computing, the recent definitions of computational thinking made fuzzy and overreaching claims. Is it really true that any sequence of steps is an algorithm? That procedures of daily life are algorithms? That people who use computational tools will need to be computational thinkers? That people who learn computational thinking will be better problem solvers in all fields? That computational thinking is superior to other modes of thought?

My critique is aimed not at the many accomplishments of the movements to get computer science into all schools, but at the vague definitions and unsubstantiated claims promoted by enthusiasts. Unsubstantiated claims undermine the effort by overselling computer science, raising expectations that cannot be met, and leaving teachers in the awkward position of not knowing exactly what they are supposed to teach or how to assess whether they are successful.

My purpose here is to examine these questions and in the process elucidate what computational thinking really is and who is it good for.

Question 1: What Is Computational Thinking?

A good place to look for an answer is in our history. Computational thinking has a rich pedigree from the beginning of the computing field in the 1940s. As early as 1945, George Polya wrote about mental disciplines and methods that enabled the solution of mathematics problems.[29] His book *How to Solve It* was a precursor to computational thinking.

In 1960, Alan Perlis claimed the concept of "algorithmizing" was already part of our culture.[18] He argued that computers would automate and eventually transform processes in all fields and thus algorithmizing would eventually appear in all fields.

In the mid-1960s, at the time I was entering the field, the pioneers Allen Newell, Alan Perlis, and Herb Simon were defending the new field from critics who claimed there could be no computer science because computers are man-made artifacts and science is about natural phenomena.[23] The three pioneers argued that sciences form around phenomena that people want to harness; computers as information

transformers were a new focal phenomenon covered by no other field. They also argued that "algorithmic thinking"—a process of designing a series of machine instructions to drive a computational solution to a problem—distinguishes computer science from other fields.

In 1974, Donald Knuth said that expressing an algorithm is a form of teaching (to a dumb machine) that leads to a deep understanding of a problem; learning an algorithmic approach aids in understanding concepts of all kinds in many fields.

In 1979, Edsger Dijkstra wrote about the computational habits of mind he learned to help him program well:[9] separation of concerns; effective use of abstraction; design and use of notations tailored to one's manipulative needs; and avoiding combinatorially exploding case analyses.

Seymour Papert may have been the first to use the term computational thinking in 1980, when in his book *Mindstorms* he described a mental skill children develop from practicing programming.[24,25]

In 1982, Ken Wilson received a Nobel prize in physics for developing computational models that produced startling new discoveries about phase changes in materials. He went on a campaign to win recognition and respect for computational science. He argued that all scientific disciplines had very tough problems—"grand challenges"—that would yield to massive computation.[33] He and other visionaries used the term "computational science" for the emerging branches of science that used computation as their primary method. They saw computation as a new paradigm of science, complementing the traditional paradigms of theory and experiment. Some of them used the term "computational thinking" for the thought processes in doing computational science—designing, testing, and using computational models to make discoveries and advance science. They launched a political movement to secure funding for computational science research, culminating in the High Performance Communication and Computing (HPCC) Act passed in 1991 by the U.S. Congress. Computer scientists were slow to join the movement, which grew up independently of them. Easton noted that, as computational science matured, computational thinking successfully infiltrated the sciences and most sciences now study information processes in their domains.[12]

The search for computational models pervades all of computational science.

The current surge of interest in computational thinking began in 2006 under the leadership of Jeannette Wing.[35–37] While an NSF assistant director for CISE, she catalyzed a discussion around computational thinking and mobilized resources to bring it into K–12 schools. Although I supported the goal of bringing computer science to more schools, I took issue with the claim of some enthusiasts that computational thinking was a new way to define computing.[7] The formulations of computational thinking at the time emphasized extensions of object-oriented thinking to software development and simulation—a narrow view the field. Moreover, the term had been so widely used in science and mathematics that it no longer described something unique to the computing field.

In 2011, on the eve of Alan Turing's 100th birthday, Al Aho wrote a significant essay on the meaning of computational thinking[2] for a symposium on computation in ACM Ubiquity.[5] He said: "Abstractions called computational models are at the heart of computation and computational thinking. Computation is a process that is defined in terms of an underlying model of computation and computational thinking is the thought processes involved in formulating problems so their solutions can be represented as computational steps and algorithms."[2, a]

Aho emphasized at great length the importance of computational models. When we design an algorithm we are designing a way to control any machine that implements the model, in order that the machine produces a desired effect in the world. Early examples of models for computational machines were Turing machines, neural networks, and logic reduction machines, and, recently, deep earning neural networks for artificial intelligence and data analytics. However, computational models are found in all fields. The Wilson renormalization model is an example in physics, the Born-Oppenheimer approximation is an example in chemistry, and the CRISPR model is an example from biology. Aho says further, "[With new problems], we discover that we do not always have the appropriate models to devise solutions. In these cases, computational thinking becomes a research activity that includes inventing appropriate new models of computation."[2]

As an example, Aho points out that in computational biology there is a big effort to find computational models for the behavior of cells and their DNA. The search for computational models pervades all of computational science. Aho's insight that computational thinking relies on computational models is very important and has been missed by many proponents.

Aho's term computational model is not insular to computer science—it refers to any model in any field that represents or simulates computation. I noted several examples above. Moreover, his definition captures the spirit of computational thinking expressed over 60 years of computer science and 30 years of computational science.[b] It also captures the spirit of computational thinking in other fields such as humanities, law, and medicine.

This short survey of history reveals two major sources of ambiguity in the post-2006 definitions of computational thinking. One is the absence of any mention of computational models. This is a mistake: we engage with abstraction, decomposition, data repre-

a Aho's definition Aho's definition was noted by Wing in 2010[35] and is used as the definition of computational thinking by K12cs.org.

b I was an active researcher in the computational sciences field during the 1980s and 1990s and can attest that his definition captures what the computational scientists of the day said they were doing.

sentation, and so forth, in order to get a model to accomplish certain work.

The other is the suggestion contained in the operational definitions that any sequence of steps constitutes an algorithm. True, an algorithm is a series of steps—but the steps are not arbitrary, they must control some computational model. A step that requires human judgment has never been considered to be an algorithmic step. Let us correct our computational thinking guidelines to accurately reflect the definition of an algorithm. Otherwise, we will mis-educate our children on this most basic idea.

Question 2: How Do We Measure Students' Computational Abilities?

Most teachers and education researchers have the intuition that computational thinking is a skill rather than a particular set of applicable knowledge. The British Computer Society CAS description quoted earlier seems to recognize this when discussing what "behaviors" signal when a student is thinking computationally.[3] But we have no consensus on what constitutes the skill and our current assessment methods are unreliable indicators.

A skill is an ability acquired over time with practice—not knowledge of facts or information. Most recommended approaches to assessing computational thinking assume that the body of knowledge—as outlined in Boxes 1–3—is the key driver of the skill's development. Consequently, we test students' knowledge, but not their competence or their sensibilities. Thus it is possible that a student who scores well on tests to explain and illustrate abstraction and decomposition can still be an incompetent or insensitive algorithm designer. Teachers sense this and wonder what they can do. The answer is, in a nutshell, to directly test for competencies.[c]

The realization that mastering a domain's body of knowledge need not confer skill at performing well in the domain is not new. As early as 1958, philosopher Michael Polanyi discussed the difference between "explicit knowledge" (descriptions written down) and "tacit knowledge" (skillful actions).[28] He famously said: "We know more than we can say." He gave many examples of skilled performers being unable to say how they do what they do, and of aspirants being unable to learn a skill simply by being told about it or reading a description. Familiar examples of tacit knowledge are riding a bike, recognizing a face, or diagnosing an illness. Many mental skills fall into this category too, such programming or learning a foreign language. Every skill is a manifestation of tacit knowledge. People learn a skill only by engaging with it and practicing it.

To certify skills you need a model for skill development. One of the most famous and useful models is the framework created by Stuart and Hubert Dreyfus in the 1970s.[10] They said that practitioners in any domain progress through six stages: beginner, advanced beginner, competent, proficient, expert, and master. A person's progress takes time, practice, and experience. The person moves from rule-based behaviors as a beginner to fully embodied, intuitive, and game-changing behaviors as a master. Hubert Dreyfus gives complete descriptions of these levels in his book on the Internet.[11] We need guidelines for different skill levels of computational thinking to support competency tests.

The CAS and K12CS organizations have developed frameworks for defining computational thinking that feature progressions of increasingly sophisticated learning objectives in various tracks including algorithms, programming, data, hardware, communication, and technology.[d] These knowledge progressions are not the same as skill acquisition progression in the Dreyfus model. The CAS framework does not discuss abilities to be acquired during the progression. The K12CS framework gets closer by proposing seven practices[e]—only three of which are directly

c In 1992, Ted Sizer of Brown University started a national movement for competency-based assessment in schools.[31] He used the term "exhibitions" for assessment events. I gave examples for engineering schools.[8] According to the Christensen Institute, competency-based learning is a growing movement in schools.[34] In 2016, Purdue became the first public university to fully embrace competency-based learning in an academic program in its Polytechnic Institute.

d CAS: https://community.computingatschool.org.uk/resources/2324; K12CS: https://k12cs.org

e Fostering an inclusive and diverse computing culture, collaborating, recognizing and defining computational problems, developing and using abstractions, creating computational artifacts, testing and refining, communicating.

related to competence at designing computations. Their notion of practice is "way of doing things" rather than an ability accompanied by sensibilities. Teachers who use these frameworks are likely to find that the associated assessment methods do not test for the abilities they are after.

Employers are turning to competency-based assessment faster than educational institutions. Many employers no longer trust transcripts and diplomas. Instead they organize interviews as rigorous problem-solving sessions with different groups in the company. An applicant will not be hired without demonstrating competence in solving the kinds of problems of interest to the employer. The idea of assessing skill by performance is actually quite common in education. In sports, music, theater, and language departments, for example, students audition for spots on the team, places in the orchestra, roles in the play, or competency certificates at a language. Although code-a-thons are becoming more prevalent and many computing courses include projects that assess skill by performance, computing education would benefit from a deep look at competency-based assessment.

Given that so much education is formulated around students acquiring knowledge, looking carefully at skill development in computational thinking is a new and challenging idea. We will benefit our students by learning to approach and assess computational thinking as a skill.

Question 3: Is Computational Thinking Good for Everyone?

The third question addresses a bundle of claims about benefits of computational thinking. Let us unpack them and see which claims are substantiated and which are not.

Wing's vision for the computational thinking movement was that "everyone, not just those who major in computer science, can benefit from thinking like a computer scientist"[36,37] At a high level it is hard to question this claim—more tools in the mental toolbox seems like a worthy goal. However, on a closer look not everyone benefits and some claims do not seem to benefit anyone. Consider the following.

There is little doubt that people who design and produce computational models and software in many fields—let's call them computational designers—develop strong skills of computational thinking. Experienced computational designers believe they are sharper and more precise in their thinking and are better problem solvers.

Recognizing this early on, Alan Perlis was one of the first to generalize (1960): he claimed that everyone can benefit from learning computational thinking. Other luminaries have followed suit.[9,18,19,24,33,35] However, this general claim has never been substantiated with empirical research.

For example, it is reasonable to question whether computational thinking is of immediate use for professionals who do not design computations—for example, physicians, surgeons, psychologists, architects, artists, lawyers, ethicists, realtors, and more. Some of

Traditional versus New Computational Thinking

The companion article traces the history of computational thinking from its origins in the 1950s until the present time. It is a story of the mental habits and disciplines for designing useful and reliable programs. It began with Alan Perlis in the 1950s, was well characterized for CS by Donald Knuth and Edsger Dijkstra in the 1970s, and expanded as the third way of science in the computational science movement of the 1980s. We call this Traditional CT.

After 2006 a new version emerged, seeded by an article by Jeannette Wing and then propelled when the U.S. National Science Foundation put a lot of resources into using CT as a conceptual lever to get computing into all K–12 schools. This massive effort defined its own version of CT independent of the past history. It is a story of how problems might be solved by expressing their solutions as computational steps. We call this New CT.

The Traditional CT and the New CT are not the same. One of the important differences is that in Traditional CT programming ability produces CT, and in New CT learning certain concepts produces programming ability. The direction of causality is reversed. The table here may help readers understand the origins of the trouble spots discussed in this Viewpoint.

Traditional CT	New CT
Mental habits and disciplines for designing useful software	Formulating problems so that their solutions can be expressed as computational steps
Extensively practicing programming cultivates CT as a skill set	CT is a conceptual framework that enables programming
Skills of design and software crafting—for example separation of concerns, effective use of abstraction, devising notations tailored to one's needs, and avoiding combinatorically exploding case analyses	Set of problem solving concepts such as representation, divide-and-conquer, abstraction, information hiding, verification, and logical reasoning
A new way of conducting science, alongside theory and experiment—a revolution in science	Useful in sciences and most other fields
Algorithms are directions to control a computational model (abstract machine) to perform a task	Algorithms are expressions of recipes for carrying out tasks; no awareness of computational models is needed
Programs are tightly coupled with algorithms; programs are algorithms expressed in a computer language; algorithms derive their precision from a computational model	Programs are loosely coupled with algorithms; algorithms are for all kinds of information processors including humans—it is completely optional whether an algorithm will ever be translated into a program
Designing computations in a domain requires extensive domain knowledge	Someone schooled in the principles of CT can find computational solutions to problems in any domain
End users can follow algorithms and get the result without any understanding of the mechanism	People engaging in any step-by-step procedure are performing algorithms and are (perhaps unconsciously) thinking computationally
Engaging in a computational task without awareness is not computational thinking	People who are engaging in any task that could be performed computationally are engaging in subconscious computational thinking

these professionals may become computational designers when they modify tools, for example by adding scripts to document searchers—but not everybody. It would be useful to see some studies of how essential computational thinking is in those professions.

Another claim suggested in the operational definitions is that users of computational tools will develop computational thinking. An architect who uses a CAD (computer aided design) tool to draw blueprints of a new building and a VR (virtual reality) tool to allow users to take simulated tours in the new building can set up the CAD and VR tools without engaging in computational thinking. The architect is judged not for skill in computational thinking but for design, esthetics, reliability, safety, and usability.[f] Similar conclusions hold for doctors using diagnostic systems, artists drawing programs , lawyers document searchers, police virtual reality trainers, and realtors house-price maps. Have you noticed that our youthful "digital natives" are all expert users of mobile devices, apps, online commerce, and social media but yet are not computational thinkers? As far as I can tell, few people accept this claim. It would be well to amend the operational definitions to remove the suggestion.

Another claim suggested in the operational definitions is that computational thinking will help people perform everyday procedural tasks better—for example, packing a knapsack, caching needed items close by, or sorting a list of customers. There is no evidence to support this claim. Being a skilled performer of actions that could be computational does not necessarily make you a computational thinker and vice versa.[13,14] This claim is related to the idea I criticized earlier, that any sequence of steps is an algorithm.

The boldest claim of all is that computational thinking enhances general cognitive skills that will transfer to other domains where they will manifest as superior problem-solving skills.[3,37] Many education researchers have searched for supporting evidence but have not found any. One of the most notable studies, by Pea and Kurland in 1984, found little evidence that learning programming in Logo helped students' math or general cognitive abilities. In 1997, Koschmann weighed in with more of the same doubts and debunked a new claim that learning programming is good for children just as learning Latin once was.[20] (There was never any evidence that learning Latin helped children improve life skills.) Mark Guzdial reviewed all the evidence available by 2015 and reaffirmed there is no evidence to support the claim.[14]

Guzdial does note that teachers can design education programs that help students in other domains learn a small core of programming that will teach enough computational thinking to help them design tools in their own domains. They do not need to learn the competencies of software developers to be useful.

Finally, it is worth noting that educators have long promoted a large number of different kinds of thinking: engineering thinking, science thinking, economics thinking, systems thinking, logical thinking, rational thinking, network thinking, ethical thinking, design thinking, critical thinking, and more. Each academic field claims its own way of thinking. What makes computational thinking better than the multitude of other kinds of thinking? I do not have an answer.

My conclusion is that computational thinking primarily benefits people who design computations and that the claims of benefit to non-designers are not substantiated.

Underlying all the claims is an assumption that the goal of computational thinking is to solve problems.

Conclusion

Promoters of computer science have long believed computational thinking is good for everyone. The definition of computational thinking evolved over 60 years applies mainly to those involved in designing computations whether in computer science or in other fields. The promoters of computer-science-for-all, believing that "designing computations" is an insular computer science activity, sought a broader, more encompassing definition to fit their aspiration. The result was a vague definition that targeted not only designers but all users of computational tools, anyone engaging in step-by-step procedures, and anyone engaging in a practice that could potentially be automated. Teachers who find the vagueness confusing have asked for a more precise definition that also clarifies how to assess student learning of computational thinking.

My advice to teachers and education researchers is: use Aho's historically well-grounded definition and use competency-based skill assessments to measure student progress. Be wary of the claim of universal value, for it has little empirical support and draws you back to the vague definitions. Focus on helping students learn to design useful and reliable computations in various domains of interest to them. Leave the more advanced levels of computational design for education in the fields that rely heavily on computing.

In the late 1990s, we in computer science (including me) believed everyone should learn object-oriented programming. We persuaded the Educational Testing Service to change the Advanced Placement curriculum to an object-oriented curriculum. It was a disaster. I am now wary of believing that what looks good to me as a computer scientist is good for everyone. The proposed curriculum for computational thinking looks a lot like an extended object-oriented curriculum. This is not a good start for a movement aiming to define something greater than programming. Early warnings that the object-oriented vision was not working came from the front-line teachers who did not understand it, did not know how to assess it, and could not articulate the

f If the architect were to specify how to erect the building by assembling 3D printed parts in a precise sequence, we could say the architect thought computationally for the manufacturing aspect but not for the whole design.

benefit for their students. We are now hearing similar early concerns from our teachers. This concerns me.

Underlying all the claims is an assumption that the goal of computational thinking is to solve problems. Is everything we approach with computational thinking a problem? No. We respond to opportunities, threats, conflicts, concerns, desires, etc by designing computational methods and tools—but we do not call these responses problem-solutions. It seems overly narrow to claim that computational thinking, which supports the ultimate goal of computational design, is simply a problem-solving method.

I have investigated three remaining trouble spots with computational thinking—the definition, the assessment methods, and the claims of universal benefit. It would do all of us good to tone down the rhetoric about the universal value of computational thinking. Advocates should conduct experiments that will show the rest of us why we should accept their claims. Adopting computational thinking will happen, not from political mandates, but from making educational offers that help people learn to be more effective in their own domains through computation. C

References

1. ACM. Computer Science Curriculum 2013; https://www.acm.org/education/CS2013-final-report.pdf
2. Aho, A. Computation and Computational Thinking, 2011; http://ubiquity.acm.org/article.cfm?id=1922682
3. Computing at School, a subdivision of the British Computer Society (BCS). 2015. Computational Thinking: A Guide for Teachers; http://community.computingatschool.org.uk/files/6695/original.pdf
4. CSTA. Operational Definition of Computational Thinking. 2011; http://www.csta.acm.org/Curriculum/sub/CurrFiles/CompThinkingFlyer.pdf
5. Denning, P., Ed. Ubiquity symposium: What is computation? (Oct. 2011); http://ubiquity.acm.org/symposia2011.cfm?volume=2011
6. Denning, P. and Martell, C. *Great Principles of Computing.* MIT Press, 2015.
7. Denning, P. Beyond computational thinking. *Commun. ACM 52*, 6 (June 2009), 28–30; DOI: 10.1145/1516046.1516054
8. Denning, P. Educating a new engineer. *Commun. ACM 35*, 12 (Dec. 1992), 82–97; 10.1145/138859.138870
9. Dijkstra, E. My hopes for computing science. 1979; https://www.cs.utexas.edu/users/EWD/transcriptions/EWD07xx/EWD709.html
10. Dreyfus, S. and Dreyfus, H. A five-stage model of the mental activities involved in directed skill acquisition. Storming Media, 1980; http://www.dtic.mil/cgibin/GetTRDoc?AD=ADA084551&Location=U2&doc=GetTRDoc.pdf
11. Dreyfus, H. *On the Internet.* Routledge 2003 (2d ed. 2008).
12. Easton, T. Beyond the algorithmization of the sciences. *Commun. ACM 49*, 5 (May 2006), 31–33.
13. Guzdial, M. HCI and computational thinking are ideological foes? Computing Education Blog 2011, (2/23/11); https://computinged.wordpress.com/2011/02/23/hci-andcomputational-thinking-are-ideological-foes/
14. Guzdial, M. *Learner-Centered Design of Computing Education: Research on Computing for Everyone.* Morgan-Claypool, 2015.
15. International Society for Technology in Education. ISTE Standards for Students, 2016; http://www.iste.org/standards/standards/for-students-2016
16. Jones, E. The trouble with computational thinking, 2011; http://www.csta.acm.org/Curriculum/sub/CurrFiles/JonesCTOnePager.pdf
17. Kafai, Y. From computational thinking to computational participation in K–12 education. *Commun. ACM 59*, 8 (Aug. 2016), 26–27.
18. Katz, D. The use of computers in engineering classroom instruction. *Commun. ACM 3*, 1 (Oct. 1960), 522–527.
19. Knuth, D. Computer science and its relation to mathematics. *American Mathematical Monthly 81*, 4 (Apr. 1974), 323–343.
20. Koschmann, T. Logo-as-Latin Redux. *J. Learning Sciences 6*, 4 Lawrence Erlbaum Associates, 1997.
21. Mannila, L. et al. Computational thinking in K–9 education. In *Proceedings of the Working Group Reports of the 2014 on Innovation & Technology in Computer Science Education Conference, ITiCSE-WGR '14* ACM, NY, 2014, 1–29.
22. National Research Council, Computer Science and Telecommunications Board. Being Fluent with Information Technology. National Academies Press, 1999.
23. Newell, A., Perlis, A.J., and Simon. Computer Science, [letter] *Science 157* (3795): (Sept. 1967), 1373–1374.
24. Papert, S. *Mindstorms: Children, Computers, and Powerful Ideas.* Basic Books, 1980.
25. Papert, S. An exploration in the space of mathematics educations. *Int'l Journal of Computers for Mathematical Learning 1*, 1 (1996), 95–123; http://www.papert.org/articles/AnExplorationintheSpaceofMathematicsEducations.html
26. Pea, R. and Kurland, M. On the cognitive effects of learning computer programming. *New Ideas in Psychology 2*, 2 (1984), 137–168.
27. Perkovíc, L. et al. A framework for computational thinking across the curriculum. In *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education, ITiCSE '10*, (2010), ACM, NY, 123–127.
28. Polanyi, M. *The Tacit Dimension.* University of Chicago Press, 1966.
29. Polya, G. How to Solve it (2nd ed.). Princeton University Press, 1957; https://math.berkeley.edu/~gmelvin/polya.pdf
30. Simon, H. *The Sciences of the Artificial*, 3rd ed. MIT Press, 1969.
31. Sizer, T.R. *Horace's School.* Houghton-Mifflin, 1992.
32. Snyder, L. *Fluency with Information Technology.* Pearson, 2003 (6th edition 2014).
33. Wilson, K. Grand challenges to computational science. In *Future Generation Computer Systems*. Elsevier, 1989, 33–35.
34. Weise, M. and Christensen, C. Hire Education. Christensen Institute for Disruptive Innovation, 2014; http://www.christenseninstitute.org/wpcontent/uploads/2014/07/Hire-Education.pdf
35. Wing, J. Computational thinking. *Commun. ACM 49*, 3 (Mar. 2006), 33–35; DOI: 10.1145/1118178.1118215
36. Wing, J. Computational thinking—What and why? Carnegie-Mellon School of Computer Science Research Notebook (Mar. 2011). https://www.cs.cmu.edu/link/research-notebookcomputational-thinking-what-and-why.
37. Wing, J. Computational thinking, 10 years later. Microsoft Research Blog (March 23, 2016); https://blogs.msdn.microsoft.com/msr_er/2016/03/23/computational-thinking-10-years-later/

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, is Editor of ACM Ubiquity, and is a past president of ACM. The author's views expressed here are not necessarily those of his employer or the U.S. federal government.

I extend personal thanks to Douglas Bissonette, Mark Guzdial, Roxana Hadad, Sue Higgins, Selim Premji, Peter Neumann, Matti Tedre, Rick Snodgrass, and Chris Wiesinger for comments on previous drafts of this Viewpoint.

Calendar of Events

June 5–7
Web3D '17: The 22nd International Conference on Web3D Technology
Brisbane, QLD, Australia,
Sponsored: ACM/SIG,
Contact: Matt Adcock,
Email: matt.adcock@csiro.au

June 5–8
ICMR '17: International Conference on Multimedia Retrieval
Bucharest, Romania,
Sponsored: ACM/SIG,
Contact: Niculae Sebe,
Email: sebe@disi.unitn.it

June 5–9
SIGMETRICS '17: ACM SIGMETRICS/International Conference on Measurement and Modeling of Computer Systems
Urbana-Champaign, IL,
Sponsored: ACM/SIG,
Contact: Bruce Hajek,
Email: b-hajek@uiuc.edu

June 6–10
SACMAT'17: The 22nd ACM Symposium on Access Control Models and Technologies (SACMAT)
Indianapolis, IN,
Sponsored: ACM/SIG,
Contact: Elisa Bertino,
Email: bertino@cerias.purdue.edu

June 7–9
PerDis '17: The International Symposium on Pervasive Displays
Lugano, Switzerland,
Sponsored: ACM/SIG,
Contact: Marc Langheinrich,
Email: marc.langheinrich@usi.ch

June 10–14
DIS '17: Designing Interactive Systems Conference 2017
Edinburgh, U.K.,
Sponsored: ACM/SIG,
Contact: Oli Mival,
Email: o.mival@napier.ac.uk

June 12–13
SCF '17: ACM Symposium on Computational Fabrication
Cambridge, MA,
Sponsored: ACM/SIG,
Contact: Stefanie Mueller,
Email: stefanie.mueller@student.hpi.uni-potsdam.de