Calhoun: The NPS Institutional Archive

DSpace Repository

NPS Scholarship | Publications

2017-08

Computational design

Denning, Peter J.

ACM

Denning, Peter J. "Computational design." Ubiquity 2017.August (2017): 2.
https://hdl.handle.net/10945/59444

This publication is a work of the U.S. Government as defined in Title 17, United States Code, Section 101. Copyright protection is not available for this work in the United States.

Downloaded from NPS Archive: Calhoun

http://www.nps.edu/library

Calhoun is the Naval Postgraduate School's public access digital repository for research materials and institutional publications created by the NPS community. Calhoun is named for Professor of Mathematics Guy K. Calhoun, NPS's first appointed -- and published -- scholarly author.

Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943

Articles

COMPUTATIONAL DESIGN

Ubiquity, Volume 2017 Issue August, August 2017 | BY **PETER J. DENNING**

Full citation in the ACM Digital Library | **PDF**

Ubiquity

Volume 2017, Number August (2017), Pages 1-9

Computational design

Peter J. Denning

DOI: **10.1145/3132087**

Computational thinking refers to a deliberative process that finds a computational solution for a concern. Computational doing refers to use of computation and computational tools to address concerns. Computational design refers to creating new computational tools and methods that are adopted by the members of a community to address their concerns. Unfortunately, the definitions of both "thinking" and "doing" are fuzzy and have allowed misconceptions about the nature of algorithms. Fortunately, it is possible to eliminate the fuzziness in the definitions by focusing on computational design, which is at the intersection between thinking and doing. Computational design is what we are really after and would be a good substitute for computational thinking and doing.

Just as we are coming to a common understanding of "computational thinking," a new term, "computational doing," has been finding some favor among educators. I used it once in a quip that our students ought to do more than think about programs: "We are most valued not for our computational thinking, but for our computational doing." [1]. George Thiruvathukal used it to mean computer scientists reaching out to people in other fields in a spirit of collaboration [2]. Valerie Barr used it to mean getting students to do useful work with computations, which significantly helped them learn basic computing [3].1

A little context is useful to understand the significance of this development. In previous articles, we have traced the history of computational thinking (CT) from its origins in the 1950s until the present time [4, 5]. Traditional CT was oriented on design and seen as a skill acquired from extensive practice with programming. Masters including Alan Perlis, Donald Knuth, and Edsger Dijktra gave accounts of the mental practices involved.

After 2006 a new version emerged as part of a U.S. National Science Foundation initiative to weave a computing thread into every K-12 curriculum,

seeded by Jeannette Wing's article in *Communications of the ACM* [6]. This massive effort defined its own version of CT independent of past history. New CT is oriented on solving problems by expressing their solutions as computational steps. It could be taught as a set of concepts without extensive programming practice.

The NSF initiative has accomplished much. Various organizations including CSTA.ORG and CODE.ORG have published curriculum guidelines for computing in K-12 schools. Five thousand teachers have been trained in CT. A new advanced placement (AP) curriculum is in place and many universities allow AP graduates to receive credit for the new college-level CS1 computer science principles course. Despite all this effort, a significant remaining trouble spot is educators have not yet converged on a common agreement defining computational thinking, on the basis of which they can firmly establish a curriculum and evaluate whether their students have learned computational thinking as a skill [4, 5].

The quest for a clear definition of CT has been complicated by a conflict between traditional CT and new CT. In traditional CT programming skill produces CT, whereas in new CT learning CT produces programming skill. The direction of causality is reversed. The accompanying table compares and contrasts the two versions of CT.

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

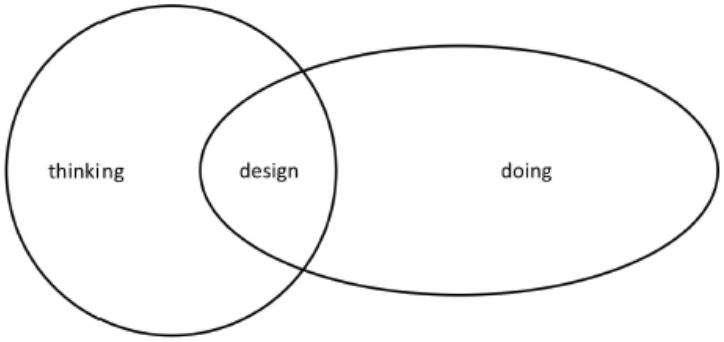
Table 1. Comparison of Traditional and New Computational Thinking (CT)2

In the years since, teachers have been grappling with the new CT. They have found new definitions vague and fuzzy. They continue to ask for a clear definition of computational thinking and the skill sets they are supposed to cultivate in their students [5]. The term "computational doing" appears as a cry for restoration of programming and design skills—a plea for a return to the original computational thinking.

Unfortunately, the term "computational doing" is also vague and fuzzy. My aim here is to give some workable distinctions between computational thinking and doing, and to propose the term "computational design" for their intersection. We can eliminate much of the apparent conflict between traditional and new CT by focusing on computational design. Perhaps the most difficult conflict is around the meaning of "algorithm" in the two interpretations of CT. I will discuss that problem and some possible ways to eliminate the conflict.

Thinking, Doing, Designing

What is the relationship among the terms "thinking," "doing", and "design"? Thinking refers to a deliberative process that finds a computational solution for a concern. Doing refers to use of computation and computational tools to address concerns. Design refers to creating new computational tools and methods that are adopted by the members of a community to address their concerns. (I use the term "concerns" instead of "problems" because we frequently use computation for tasks that are not seen as problems.) Clearly, designers are a subset of thinkers because you need to be a computational thinker to design computational tools; and not every thinker is a designer. Also, designers are tool users, but not all tool users are designers or thinkers. The figure below shows a Venn diagram of the relationships among these three terms.



I applaud the growing realization among computational educators that computational thinking is a skill. I am skeptical of models of learning that ignore skill development and rely solely on knowledge acquisition. Knowledge nowadays means "organized information" (as in scientific body of knowledge),and sometimes just "information" (as in what you find by searching the Internet). To rely solely on knowledge completely misses that the world is made of practices that we engage in with each other and embody as skills [7]. Computing technology has enabled much innovation because it opens many new possibilities for practice.

Graduates of high school and college are often shortchanged by overemphasis on knowledge acquisition relative to skill development. Employers say our graduates often lack important skills in software engineering, design, communication, and navigating in a fast-changing world full of surprises and unexpected turns. Were we to become more attentive to practices, our graduates would be better navigators of this world.

Computational Designers

Computational designers fit nicely into the intersection between computational thinkers and doers. The term designer often means someone who creates a fashion, as in apparel design, or someone who makes blueprints, as with engineering drawings of machines or buildings. Computational designers, however, do more than create fashions or plans: They craft computations that do jobs people care about getting done. The computational designer constantly moves back and forth between listening to the concerns of a user

community and proposing new computational approaches to take care of their concerns. The new approaches arise from new combinations of existing computing components. Computational doing overlaps with the expertise of designers, but includes many non-design activities, such as simply using computational tools or using systems with embedded computing.

Is someone who uses Excel a computational designer? A doer? It depends. If you are using a spreadsheet designed by someone else, you simply put the numbers in the right places and let the spreadsheet do the calculations; you are not designing. You do not know what formulas or algorithms are used but you are aware the spreadsheet is a computational tool. If you construct a spreadsheet to help someone do a task, you are a computational designer.

In the field of operations research, for instance, analysts design spreadsheets containing optimizing models for situations others work in, and then use the models to help optimize the way the work is done. Those professionals are computational designers. Similarly, other professionals design computations in Matlab, Mathematica, Photoshop, 3-D modeling, and more to help others solve problems in their areas. Most of these designers are domain experts but not computer scientists. In short, you can be a skilled computational designer in many fields without being a computer scientist and without even thinking of yourself as engaged in computer science [2].

The promoters of new CT have brought the issue of awareness into the conversation. They make a strong appeal that doing-without-awareness qualifies as computational thinking. This claim is part of their case that computational thinking is universal. They have argued people who use computational tools without knowing it, or engage in step-by-step procedures, can be counted as computational thinkers.

This claim compounds the confusion around computational thinking. It is hard to say those who are unaware they are using computation are thinking computationally. It is equally hard to say what they are doing computationally, because computational doing has a sense of intention behind it. For instance, most drivers are unaware their cars are distributed computing networks on wheels, often relying on 50 or more embedded processors. Are we sometimes doers (when we are aware) or sometimes merely end users (when we are unaware)? Many would prefer to not call an unaware end user a computational doer.

A related misconception is many everyday tasks, such as packing a knapsack, are computational, and therefore people performing such tasks are thinking computationally without being aware of it [6]. As a lifelong computer scientist, I tend to view every task with a computational lens and thus I may well see a knapsack problem in my backpack. However, my friends in other fields successfully pack backpacks without knowing a thing about computation or knapsack problems. They are obviously not computational thinkers or computational doers. Just because it looks computational to me, does not mean it is inherently computational or that anyone else sees it the same way.

The Algorithm Problem

The two interpretations of CT have generated confusion around the definition of algorithm. This confusion clouds what is expected of computational designers and thinkers.

At the start of his monumental work, *The Art of Computer Programming*, Don Knuth discusses the meaning of algorithms. He says there are five requirements for a procedure to be an algorithm: finiteness, input, output, effectiveness, and definiteness. The last has been mangled in the muddles of computational thinking.

Definiteness means each step has a definite and unambiguous effect.

Algorithms expressed in English (or any other human language) are subject to different interpretations by different listeners, according to their language and cultural backgrounds. Knuth that to avoid this difficulty, we invented formally defined computer languages, whose statements are rigorously unambiguous. When we use a computer language to express an algorithm, we avoid the risk that users will misinterpret our intentions. Algorithms expressed in a computer language are programs. There is thus a tight coupling between algorithms and programs.

Some promoters of new CT have claimed a step-by-step procedure, such as a kitchen recipe, is an algorithm, and a person following a recipe is a computational thinker. Knuth says kitchen recipes are notorious offenders of the definiteness requirement: "Instructions like 'toss lightly until mixture is crumbly' are quite adequate as explanations to a trained chef, but an algorithm must be specified to such a degree that even a computer can follow the directions."

Thus, the promoters of new CT have espoused a definition of algorithm that conflicts with long traditions in mathematics and computer science. This is a serious error. Because the error is not obvious to many teachers and students, students are learning fallacies about algorithm design algorithms and the limitations of computing machines.

You might ask whether the insistence on definiteness can be maintained for new technologies whose outputs can vary with probabilities, such as quantum computing and white noise generators powering randomized algorithms. Definiteness can be maintained in these cases. Definiteness does *not* mean that an operation always produces the same answer with the same inputs. It means each operation has a well-defined effect that can be carried out by a machine. Getting a random number from a white noise generator is an example. Feeding parameters to a quantum computer that approximately solves an optimization problem is another example. We routinely accept machines can produce different outputs with different probabilities for the same input. We can have definiteness even with randomized algorithms.

This is not the end of it. Some promoters of new CT have argued an algorithm is an expression but not a program. (For example, computingatschool.org.uk.) The intention of this distinction is to divorce the notion of algorithms from a computing machine, because the promoters believe that computational thinking can be used in everyday life without programming. Moreover, they see "expression" as a hallmark of individual creativity and do not want children to feel constrained from being creative while inventing algorithms. (I wonder if they believe programming constrains creativity.) This attempt at separation is another fallacy and ignores long traditions of mathematics and computing.

Al Aho pointed out it is impossible to design computations without having a computational model in mind [8]. The computational model is a conceptual machine controlled by the algorithm. It is the reference point for "precision" in the meaning of computational steps. His insight is lost in the fallacious notion that algorithms are merely expressions with no necessary connection to machines [9]. Algorithms cannot be cleanly separated from programs as long as we insist the steps be so precise that a computer could do them.

Conclusion

The term computational doing was offered as an antidote to the possibility that new computational thinkers will live in their own worlds of thought and produce nothing useful for people in the real world. The proposed antidote, computational doing, is even fuzzier than new computational thinking. Fortunately, it is possible to eliminate the fuzziness by focusing on computational design, which is at the intersection between thinking and doing. Computational design is where the power of the computing revolution is

showing up. Computational design is what we are really after and would be a good substitute for computational thinking and doing.

I wish we could restore rigor to the definition of algorithm. Misconceptions allowed by the claim that step-by-step procedures are algorithms will lead students to believe they are creating computational entities when they are not.

References

[1] Denning, Peter. Beyond computational thinking. *Communications of ACM* 52, 6 (June 2009), 28-30.

[2] Thiruvathukal, George K. Computational Thinking … and Doing. *Computing in Science and Engineering* 11, 6 (November-December, 2009).

[3] Barr, Valerie. Disciplinary thinking, computational doing: promoting interdisciplinary computing while transforming computer science enrollments. *ACM Inroads* 7, 2 (May 2016), 48-57.

[4] Tedre, Matti and Denning, Peter J. (2016) The Long Quest for Computational Thinking. In *Proceedings of the 16th Koli Calling Conference on Computing Education Research* (Nov. 24-27, Koli, Finland). ACM, New York, 2016, 120-129.

[5] Denning, Peter. Remaining trouble spots with computational thinking. *Communications of ACM* 60, 6 (June 2017).

[6] Wing, Jeannette. Computational thinking. *Communications of ACM* 49, 3 (March 2006), 33-35.

[7] Denning, Peter, and Fernando Flores. Emergent innovation. *Communications of ACM* 58, 6 (June 2015), 28-31.

[8] Aho, Al. 2011. **Computation and computational thinking**. ACM *Ubiquity* (January 2011).

[9] Denning, Peter. Computational thinking in science. *American Scientist* 105 (Jan-Feb 2017), 13-17.

Author

Peter J. Denning 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-in-Chief 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.

Footnotes

1. Curiously, computational doing is not mentioned in the text.

2. From sidebar in "Remaining Trouble Spots with Computational Thinking" [5]

©2017 ACM $15.00

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

The Digital Library is published by the Association for Computing Machinery. Copyright © 2017 ACM, Inc.

COMMENTS

POST A COMMENT

Your Name (Required)

Your E-Mail address (Required)

Comment (Required - HTML syntax is not allowed and will be removed)

Post