Computational Thinking: A Definition

Janice E. Cuny
National Science Foundation
4201 Wilson Boulevard
Arlington, VA 22230
1 703-292-8489
jcuny@nsf.gov

Lawrence Snyder
University of Washington
Department of CSE
Box 352350, Seattle WA 98195
1 206 543 9265
snyder@cs.washington.edu

Jeannette M. Wing
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213
1 412 268 3068
wing@cs.cmu.edu

ABSTRACT

Computational Thinking has sparked considerable interest in the computing community. It calls for the thinking habits of computer scientists to be more widely practiced by other scientists and engineers, and somewhat less technically, by all members of society. Computer scientists understand concepts like abstraction and algorithmic analysis, but the public does not. To promulgate computational thinking widely, we must define what we mean in a way that is understandable by the public. This paper's intent is to provide a definition that computer scientists can use in presenting Computational Thinking to a wider audience. It is not meant to define what computer science is, but rather to define in general terms the kinds of thinking skills and approaches used by computer scientists.

Categories and Subject Descriptors

K.3.2 [**Computer and Information Science Education**]: Computer science education, curricula, literacy.

General Terms

Algorithms, Performance, Design, Experimentation, Languages

Keywords

Computer science education, computational thinking, abstraction, automation, analysis.

1. INTRODUCTION

Since the term was introduced in a Viewpoints article in CACM in 2006 [1], computational thinking has been the subject of much discussion in the computing community. The concept,

> Computational thinking involves solving problems, designing systems, and understanding human behavior, by drawing on the concepts fundamental to computer science,

got traction because the article advocated teaching computational thinking broadly. Scientists need such training because computation has become the "third pillar" of science [2]. More generally, college students need to acquire the thinking habits of computer scientists, because they are widely applicable in the information society in which those students will live and work, regardless of their eventual profession. And ultimately, K-12th graders should be introduced to computational thinking in order to set these fundamental thinking habits firmly in the minds of the next generation.

The interest and excitement surrounding computational thinking and the potential of spreading it broadly across the population has motivated several recent projects:

- The College Board is designing a new AP course that covers the fundamental concepts of computing and computational thinking.
- The National Academies' Computer Science and Telecommunications Board is holding a series of workshops on "Computational Thinking for Everyone" with a focus on identifying the fundamental concepts of computer science that can be taught to K-12 students.
- On May 29, 2009, an event on The Hill sponsored by ACM, CRA, CSTA, IEEE, Microsoft, NCWIT, and SWE called for putting the "C" (computer science) into "STEM."
- The NSF, through the CISE CPATH program, emphasizes computational thinking in efforts to revitalize undergraduate computer science curricula.
- Microsoft supports the Carnegie Mellon Center for Computational Thinking.
- CSTA has produced and disseminated *Computational Thinking Resource Set: A Problem-Solving Tool for Every Classroom.*
- The NSF CISE Directorate's BPC Program has launched the CS/10,000 Project that aims to catalyze a revision of high school curriculum, with the new AP course as a centerpiece, and to prepare 10,000 teachers to teach the new courses in 10,000 high schools by 2015

Additionally, panels and discussions on the topic have been plentiful at venues such as SIGCSE, ACM Educational Council, and the newly formed CRA-E.

Beyond the computer science community computational thinking has attracted some interest from professionals in other fields (e.g., see CDI above) as well as educators. They are curious about it. But not being schooled in computer science, they struggle to understand exactly what it is. "Does it mean we should all think like a computer? Does it mean everyone should be a programmer?" (Answers: No.) And this confusion is understandable. The rationale for wide adoption of computational thinking, which is so attractive to computer scientists, is not yet accessible to non-computer scientists. Computer scientists see the value of thinking abstractly, thinking at multiple levels of abstraction, abstracting to manage complexity, abstracting to deal with scale, etc. We know the value of these capabilities. Others do not know what we mean.

Because the goals of the computational thinking discussion – get it to other scientists, teach it to all college students, include it in K-12 curricula – involve populations outside of computer science, we cannot advance unless we can explain what it is all about. People in those communities must understand computational thinking enough to appreciate its importance.

The purpose of this paper is to define computational thinking in a way that is accessible to the lay population. This is not a trivial translation of computer jargon into English. Rather, we must consider exactly what it is we mean by computational thinking, and be explicit, translating it into easily understood terms. We must give everyday examples. We must say why it is so useful. In this way we will make it accessible to the public at large, and hopefully move the program forward. We might even find that not every computer scientist means the same thing despite understanding all the words [3].

2. ANTECEDENTS

The computational thinking effort is not the first attempt to bring knowledge about computing to broader audiences. In the 1970s and 1980s non-major "computer appreciation" classes were common; they covered few concepts and had no programming. In the 1990s "computer literacy" classes focused on computer applications and contained even less about computational thinking.

Motivated by goals similar to the computational thinking effort, Bill Wulf, then Assistant Director of CISE at NSF, commissioned an NRC study of "What everyone should know about IT." The 1999 report, *Fluency with Information Technology* [4] called for teaching "skills, concepts and capabilities," and gave ten sample topics for each. The report made the case that "some" programming knowledge is essential for the general public. "Skills" refers to day-to-day proficiency with applications and is basic literacy. However, the "concepts," such as algorithmic thinking, and "capabilities," such as logical reasoning, correlate closely with computational thinking. The Fluency recommendation, consistent with the study's charge, emphasized specific topics rather than seeking the holistic approach that computational thinking does. Today fluency is taught in many colleges, but only occasionally in high school. To remedy the slow transfer to high school, NRC conducted another study, which issued the report *ICT Fluency and High School Graduation Outcomes* [5].

3. PRELIMINARY TERMS

One advantage the cognoscenti have is they understand the terms of the field deeply. It's easy to understand one another. But the shared knowledge so thoroughly hides their thinking as to sound like jargon to everyone else. In order to make the concept of Computational Thinking more widely accessible, we will have to define our terms carefully. We start here, at the very basic level of data and processes.

Computing is fundamentally concerned with two phenomena: data and processes. Both exist in the physical world we can touch, and in the logical world of concepts. Roughly speaking, data are static entities, while processes are dynamic entities that operate on data.

Data is anything that can be observed or imagined in the physical or logical worlds. Everyday examples include numbers, images, songs, positions of the planets, subway maps, and medical records. Examples closer to computing: programs are data, the relationship among pages at a Web site (as in the site map) is data, a record of the path of a packet through the Internet is data, and so forth.

A *process* is a sequence of actions. The process of setting up the coffee maker in the morning starts with these individual actions: getting out the filter, placing it in the basket, filling the reservoir with water. Other real-world examples of processes include balancing a checkbook, monitoring a patient's temperature, searching the Web, guiding a rocket towards Jupiter, and natural processes like metabolism and protein folding. A process's sequence of actions can be finite or infinite, and a comprehensive definition would be broad enough to relax "sequence."

Data and processes are everywhere.

4. COMPUTATIONAL THINKING DEFINED

At this point, it is perhaps clearer to jump to our central definition, leaving key terms – abstraction, automation, and analysis – that it uses to be defined in the remainder of the section.

Computational Thinking is the use of *abstraction*, along with *automation* and *analysis*, in problem solving,

4.1 Abstraction

Abstraction is *the process of generalizing from specific instances*. As used in problem solving, it is intended to capture essential common characteristics, while discarding unessential characteristics. It is at the core of typical computing activities such as developing algorithms, identifying structural properties, patterns, and relationships in data, parameterization, creating new computational systems, and discovering emergent behaviors in complex systems. Abstraction is the defining characteristic of computational thinking.

There are many kinds of abstraction. Let's consider two: a data abstraction and a process abstraction. The DC metro map below is an example of a data abstraction. It represents information in a way that is useful for its intended purpose: to help people find their way from one part of DC to the other using the metro. It is a topological abstraction of the physical layout of the metro lines. The map gives exactly the relevant detail needed for someone to find his or her way from the Dupont Circle in downtown DC to Reagan National Airport in Virginia. It shows relevant details, like the places where we can transfer between the different lines (the transfer points shown as large, double circles) while it eliminates irrelevant detail like distances between each pair of stops or the street locations of each metro station.

The typical abstraction of a process that takes some input data, follows a sequence of actions, and produces some desired output data is called an *algorithm.* A recipe for cooking chocolate chip cookies is an algorithm. The raw ingredients, such as flour, sugar, and eggs, are the algorithm's inputs. The recipe's steps are the algorithm's sequence of actions; and the baked cookies are the algorithm's outputs. An algorithm focuses our attention on relevant detail and abstracts away from irrelevant detail. For example, a typical chocolate chip cookie recipe says exactly how much flour to use, and the order in which to add each ingredient and how. But, it eliminates irrelevant details such as what kind of oven or utensils to use.

Some abstractions *exist only in the mind, separate from any embodiment* or explicit representation. For example, computational thinkers use a design methodology called *divide and conquer*. The details are unimportant, but the idea is to solve a problem by dividing it into n subproblems, for concreteness, say n = 2. We solve both smaller problems and combine the two results. The two smaller problems are each solved the same way – dividing them into two, solving those two smaller problems and combining them, and so forth until we reach a base problem we know how to solve. It is an abstraction of how to solve problems; it is not actually a solution to anything. It is an abstraction to guide computational thinkers in finding good solutions.

Other abstractions – most in fact – can be expressed in symbolic form and thereby be made tangible. We can carry a copy of the DC metro map in our pocket; we can print a recipe on the back of the bag of chocolate chips.

For computing purposes, we often use a binary representation (based on the symbols 0 and 1) to express data in a tangible form. But, data abstractions usually impose structure and describe relationships that go considerably beyond the representation of the content. For example, a spreadsheet – the arrangement and organization of the cells, their properties and dependences – contains information beyond the bits that represent the values in each position. The spreadsheet makes this additional information about the data tangible. Likewise, for computing purposes, processes are expressed in symbolic forms, notably as programs written in a programming language, but also as algorithms in pseudo-code, flowcharts, state diagrams, database queries, spreadsheet formulas, HTML files, and on and on. These are the tangible and familiar forms of computation.

When we specify abstractions of data or processes in symbolic form using a formal, written language, the results are *expressions*. Motivated by a recent formulation of Denning and Freeman [6], we say that computational thinking concerns *expressions that describe data or processes*. Expressions can be manipulated with a computer. Expressions of algorithms are not just descriptive, but also *generative*; that is, they are capable of producing actions when interpreted by a suitable *agent*.

4.2 Automation

The agent that interprets an expression of an abstraction can be a person, a computer, a group of people or computers, and combinations thereof.

When a human interprets an expression, such as the instructions for setting up the coffee maker, the instructions can rely on his or her intelligence. An instruction like “fill the reservoir with water” is simple compared, for example, to describing to a robot what the reservoir is, what water is and how it is handled, and how to fill it from the sink. Thus, we can use higher-level instructions when instructing people. This is convenient. Unfortunately, humans are also unreliable agents. Perhaps the most compelling example of this frailty is the number of lives saved in hospitals simply by verifying (using a checklist) that the written-out steps in a medical procedure are actually performed by the medical personnel [7].

Computers on the other hand tirelessly and stolidly follow precise instructions. They are superbly engineered to check themselves continuously, producing a near perfect agent, a laborsaving device ready to perform any process for us. For many problems, they are lightning fast, compared to the processing power of humans. Their memory capacity is practically infinite and stored data persist forever, dwarfing what a human can remember in his or her lifetime.

Automation inspires computational thinkers to invent and create new abstractions. Because the tireless agent executes the abstractions perfectly, using a resource (computer) that is already available and largely underutilized, new “work” comes for free. Further, because executing abstractions is not significantly constrained by limitations of the physical world, the main limit to inventiveness is human imagination.

4.3 Analysis

The computer as a “perfect agent,” however, extracts a huge price from computational thinkers. First, it is “not very bright,” meaning that the instructions of the process must be extremely primitive, making the automation task challenging. And second, it is unforgiving, meaning the instructions must be exactly right. Thus the automation of abstractions is both challenging and exacting. The expressions produced by computational thinkers are not amorphous collections of instructions or other representations. Quite the opposite. They are carefully organized and structured to achieve the correct result when interpreted by an agent. Achieving this desired behavior is challenging. Accordingly, a central activity for computational thinkers is analyzing abstractions: What does this algorithm compute? Is it correct for all possible input data? How much of the available computational resources does it require? Does this data abstraction capture the relevant properties of interest? What algebraic properties does this data abstraction enjoy?

Computational thinkers must do certain types of analysis to ensure that their abstractions – expressed as algorithms, programs, databases, and systems of all sorts – achieve the goals of *efficient* and *correct* behavior. These analyses include: *algorithmic analysis, performance analysis, specification, verification, debugging, testing, and experimentation.*

More generally, however, computational thinkers spend considerable time designing new and better abstractions. The creation of any new abstraction requires some design decisions, and in this sense it shares with engineering the need to be inventive in the presence of constraints. As in engineering computational thinkers need to have good judgment: to be able to make design tradeoffs and to evaluate an abstraction by qualitative measures such as simplicity and elegance, Unlike engineers who are constrained by limits of the physical world, computational thinkers can design abstractions that exist in synthetic worlds, and thus their imagination is the only limit on their design creativity.

4.4 The 3As: Abstraction, Automation, Analysis

Abstraction is a near-universal tool. It is full-time logical reasoning. Abstractions can be interpreted (executed) by any agent with sufficient capability. Expressing an abstraction so that a computer can interpret it is both challenging and exacting—it takes human creativity and ingenuity. And, to get abstractions and their expressions right and to evaluate their goodness requires analysis, reasoning, and judgment. Computational thinking is a rich intellectual activity with wide applicability.

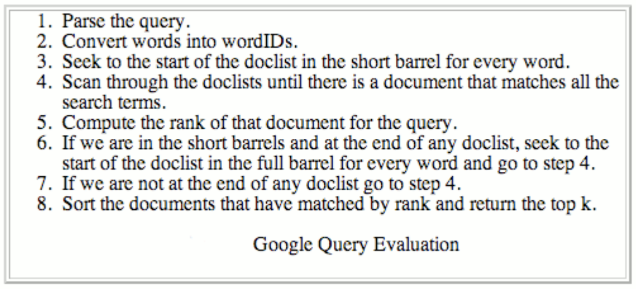
Though computational thinking is unique, it shares features with other forms of thinking. With respect to abstraction, it has closest ties to mathematics, and with respect to analysis it has closest ties to engineering. *Automation, which is exploration and discovery with computers, gives us the courage to solve problems at a scale and complexity beyond anything humans can do alone.* Computational thinking may be new collection of high-quality intellectual skills, but it extends a rigorous and respected foundation.

5. BENEFITS

5.1 Levels of Abstraction

By definition, abstraction involves two levels: the more general level to which we abstract and the more specific level from which we abstract. The true power of abstraction comes from layering, one level more abstract than the level below.

Let's consider the task of evaluating a search query. The figure below gives Brin and Page's original query processing algorithm [8] for a Google search. These eight instructions are "very large" in that each assumes a high capability by the interpreting agent, in this case a human reader. They are sufficient to explain (to those who have read the paper in which they appear) to a human how a computer implements the search. So they describe to computational thinkers how the software works. It is one level of abstraction – probably the highest level of abstraction – of the query processing software.


1. Parse the query.
2. Convert words into wordIDs.
3. Seek to the start of the doclist in the short barrel for every word.
4. Scan through the doclists until there is a document that matches all the search terms.
5. Compute the rank of that document for the query.
6. If we are in the short barrels and at the end of any doclist, seek to the start of the doclist in the full barrel for every word and go to step 4.
7. If we are not at the end of any doclist go to step 4.
8. Sort the documents that have matched by rank and return the top k.

Google Query Evaluation

No computer has hardware for the first action "Parse the query" or any of the other seven actions, so they must be simplified. Parsing the text that we type into Google's query window requires processing by a sequence of algorithms that might be expressed by the instructions

Group letters into words

Identify quoted phrases

Find minus words (NOT)

Group OR words

…

This description of how "Parse the query" is implemented is *another level of abstraction*. It is a lower level of abstraction because the operations are simpler, presuming less capability by the executing agent.

We can consider how each of these actions is implemented with simpler instructions, creating other still lower levels of abstraction. Eventually we will get to the program code for these algorithms written in a programming language such as C or JavaScript. It forms another level of abstraction – the *programming level* – with components such as

```
if (query[i] == QUOT || query[i] == APOST)
     phrase.length = count;
```

that express the computation simply enough that the actions can be translated into a form a computer can actually understand. This translation – performed by computers – goes first to a level of abstraction known as the *assembly code* level, and then to a lower level, called *machine code*, which is actually written out in binary. Assembly code is probably the lowest level for most computer scientists, though there are several lower levels used by computer architects, electrical engineers, and chip designers.

A key aspect of this discussion is that there is only one computation, but it is expressed in terms of multiple levels of abstraction. At each level instructions get simpler than those above it, and their total number increases. To continue the query-processing example, the eight instructions of the highest level of abstraction expand to 100,000s of thousands at the lower programming level, and possibly a million at the assembly language level.

The explanation makes clear why many levels of abstraction exist. But how do computational thinkers benefit from them? First, abstraction gives us the ability to focus our attention on one level at a time; ignoring all details of all the levels below. It allows computational thinkers to build large systems, one level of abstraction at a time, without having to think about all of the details of one monolithic system all at once. Second, the relationship between each pair of levels is just as important as the two levels themselves. When we compile a C program into assembly code, the compiler is transforming expressions at a high level of abstraction into expressions at a lower level; the compiler had better be correct so that the assembly code will have the same intended behavior as that expressed by the C program. The compiler formally defines a relationship between the two layers of abstraction. The compiler itself is an abstraction (i.e., a process that translates expressions from one language to another), and moreover, because it is itself a formal expression of a relationship between two levels of abstraction we can *automatically* build lots and lots of systems: given any C program, we can produce the corresponding assembly code.

When applying the idea of layers of abstraction throughout a computation or more generally in building large systems, computational thinkers end up moving up and down among the levels smoothly. It promotes a nimble mind, and is the sort of thinking students can be taught.

5.2 Building Large, Complex Systems

Computing has brought us many very complex systems: the Internet, the Web, and social networks, to name a few. These are complex because they permit a rich set of interactions. In fact, nearly all software of any sophistication is so complex as to be beyond *complete* understanding, even with all of the current tools of computer science.

Armed with the ability to automate and analyze abstractions—and layer abstractions—computational thinkers can build large, complex systems.

Two essential tools used by computational thinkers to manage the complexity of large systems are *decomposition/composition* and *stepwise refinement*. Both are applicable broadly. Decomposition is simply the idea of breaking a task into parts and working on them separately; if they are still too complex to deal with, decompose each, and so forth. (Divide-and-conquer of Section 4.1 is an example of decomposition.) *Composition* is the opposite: starting with pieces and composing them to build larger and larger systems.

Stepwise refinement, which is closely related, solves a problem using (possibly imagined) operations that are sufficient to produce the result, but probably not within the agent's "ability," i.e. they are not actual operations the agent can perform. With the problem solved in terms of those operations, return to implement them using stepwise refinement. Repeating until all operations are lare legitimate for the agent, produces a working result. More generally, refinement, by going from the general to more specific, is the opposite of abstraction, which goes from the specific to the more general.

Such techniques allow computational thinkers to design and build systems of astonishing complexity, but they can be used in nearly any problem-solving situation.

6. EDUCATIONAL CONSIDERATIONS

Computational thinking is a rich and valuable intellectual capability with value to students. *Improved ability to abstract.* Abstraction is widely used outside of computing, and computational thinking will make students more sophicated in its use, better, for example, at separating the relevant from the irrelevant aspects of a problem.

Ability to automate an abstraction. Expressing a process well enough that a computer can execute it is challenging, requiring students to be logical, thorough, clear, and precise in their thinking.

Ability to assess the goodness of an abstraction. Analyzing an abstraction and assessing how well it meets its requirements – How efficiently does it encode data? How long does the process execute? etc. – is a valuable skill that easily translates to other settings.

Greater facility with scale and complexity. Computational thinking skills allow us to deal with huge, real-world systems, can be taught and applied in a classroom setting.

Sustained logical reasoning. Case analysis, debugging, exception handling, invariants, type checking, correctness by construction, and many other aspects of computational thinking provide myriad of opportunities to enhance a student's reasoning abilities.

Improved facility with parallelism. Computers work together to solve problems; so do people, but in both cases synchronization, coordination, and communication are fraught with complications; computational thinking clarifies the opportunities and challenges of parallelism guiding students to avoid difficulties.

Improved persistence in problem solving. It is possible to engage students in very compelling applications of computational thinking that may increase their interest and enthusiasm for problem solving in general.

Learning design skills. Design of abstractions requires the ability to satisfy logical constraints and make engineering tradeoffs, but also requires good taste and understanding of the end user.

Opportunities to create. Computational thinking applies to the synthetic world of information where a student can realize creative ideas through automation. Dreams and ideals can be fulfilled.

7. Future Work

Computational thinking must be brought in to the K-20 curriculum, both in classes devoted to computing and in classes for other disciplines. It is important for our science students to understand and be facile with the use of computation in their fields, it is important for our journalism, art, and music students as well. Computational thinking is a basic 21st century skill. As perhaps one of the many next steps in achieving this goal, we would like to write a primer full of real-world example abstractions suitable for teaching computational thinking to children.

8. REFERENCES

[1] Jeannette M. Wing, "Computational Thinking," CACM 49.03:33-35, 2006.

[2] PITAC Report: "Computational Science: Ensuring America's Competitiveness," National Coordination Office for Information Technology Research and Development, 2005.

[3] British Computing Society Debate on Computational Thinking, 2007, `www.bcs.org/server.php?show=ConWebDoc.11837`

[4] National Research Council, *Fluency with Information Technology: Skills, Concepts and Capabilities*, NAP 1999.

[5] National Research Council, *Information and Communication Technology Fluency and High School Graduation Outcomes,* NAP 2006.

[6] Peter J. Denning and Peter A. Freeman. Computing's Paradigms. CACM, to appear.

[7] Atul Gawande, "The Checklist," *New Yorker*, Dec. 10, 2007.

[8] [SB94] Sergei Brin and Lawrence Page, "The Anatomy of a Large-Scale Hypertextual Web Search Engine," `infolab.stanford.edu/~backrub/google.html`