The Endowment for the Study of Computational Thinking

Russ Abbott

The computer has changed the world. But there is an aspect of its influence that is only now beginning to be understood. In March 2006, Jeanette Wing (2006) published "Computational Thinking," a column whose title is coming to refer to one of the most important ways computer science has influenced society. The Center for Computational Thinking at Carnegie Mellon University (Wing's home Institution) describes Computational Thinking as "a way of solving problems, designing systems, and understanding human behavior that draws on concepts fundamental to computer science." The premise of computational thinking is that ideas developed and refined within computer science can provide significant intellectual leverage outside computer science.

The Endowment for the Study of Computational Thinking at CSULA has been created to support two primary activities.

(a) To develop computational thinking as a discipline in its own right development and
(b) To encourage the continued and extended application of computer science concepts to areas beyond computer science.

It's not surprising that computer science has developed ways of thinking that are applicable to many disciplines. After all, software is used in virtually all fields. For a computer program to be of any value it must embody concepts central to its area of application. In other words, to write a computer program one must understand the target discipline well enough to be able to stand back from it and analyze its central concepts. Analyzing concepts is not new. Much of contemporary philosophy is devoted to conceptual analysis. The philosophy of science, for example, asks such questions as: what do we mean by reductionism or causality and what do we require of something for it to count as a scientific explanation? Since both computer science and philosophy analyze ideas, Frederick B. Thompson, a former mentor, selected as his Caltech title to be Professor of Computer Science and Applied Philosophy.

Since there is no compromising with computers, computer science must go one step beyond conceptual analysis. Software can't implement poorly formulated ideas. To write a computer program not only must one analyze ideas, one must cast them in precise enough terms that they can be expressed as instructions for a computer.

Demanding as that task may be—and it is quite demanding—computer science is faced with yet an additional challenge. Software is written in a programming language. But it is not feasible to develop a new programming language for every discipline. Computer science is required to write software for virtually any field but to do so in a limited number of programming languages. How can that be done? Computer science has solved this problem by developing programming languages with two important features.

- Today's programming languages embody a repertoire of core knowledge representation mechanisms, i.e., mechanisms that are widely useful across a broad range of disciplines. These include the notions of conditional choice, repetition (and more generally recursion), state machines, databases (and the organization or information), networks (and communication across them), computational objects, autonomous agents, evolutionary computing, interaction through what appears to be the direct manipulation of virtual objects within a virtual world, and platforms as infrastructures within and upon which additional capabilities can be constructed.

- But a foundation of core knowledge representation concepts is not sufficient. When writing software for a particular discipline one must be able to extend a programming language with concepts for that discipline. So programming languages also have means to allow the addition of new concepts. One of the most significant developments in this area is the notion of a level of abstraction, a collection of categories along with the sorts of actions entities belonging to those categories are capable of performing. The ability to define a level of abstraction enables software developers to formulate new ideas explicitly, defining them and building them into software as the need arises.

The preceding explained why computational thinking has application beyond computer science. Here are some examples from my own work.

- In "The reductionist blind spot" (Abbott 2009) I showed that traditional reductionist thinking produces a blind spot in how we understand nature. The argument is based on the notion of levels of abstraction. A level of abstraction establishes entities (such as you and I) that are both ontologically real while at the same time being physically reducible (to elementary particles such as electrons). Standard reductionism has no way to explain the reality of higher level entities—even though it seems intuitively clear to us that macro-level objects do indeed exist as real entities. A level of abstraction establishes the reality of categories of entities through the independence of their specifications while simultaneously allowing those specifications to be implemented in any of a number of ways.
- In "Bits don't have error bars" (Abbott 2009) I discuss differences between how engineers think and how computer scientists think. Because computer scientists can count on the bit (virtual though it is) as being absolutely indestructible we are free to build arbitrarily complex and sophisticated levels of abstraction on top of it. The unbreakable bit serves as an absolutely reliable foundation for anything built on top of it. Engineers, in contrast, build artifacts from physical materials. Because one can at best approximate the strength of any material, engineers find themselves building models that approximate downward to be sure that the foundations upon which they are building are actually firm enough to support the superstructure built upon it. To simplify downward approximation engineers use functional decomposition. This downward-facing approach—often referred to as building stove pipes—is distinctly different from the computer science approach—which focuses on building horizontal integrative platforms upon which new constructions may be erected.
- In "If a tree casts a shadow is it telling the time?" (2008) I explain that computer software is the only symbolic language that can be interpreted outside the human brain. That means that a software developer is able to externalize his/her thoughts in such a way that those thoughts can be brought to life without having to be re-internalized by another human being. Human beings have been externalizing thought ever since there was language. But thoughts externalized in natural language are no more than marks on paper or noises in the air until a human being reads or hears them. Thought externalized as software is both symbolic—like natural language—and interpretable by an agent other than a human being, namely a computer.
- In "Unintended consequences," (2009) an MS student and I explained the mechanism behind unintended consequences. An example—that fortunately was caught in time—was a plan by the US Army in Iraq to pay a reward to anyone who brought in an unexploded IED—otherwise known as an Improvised Explosive Device. Instead of reducing the number of IEDs such a plan would have resulted in their increased production—since the Army would be providing a guaranteed market for them! This example illustrates the idea that once a mechanism is installed in the world, it will be used by anyone who finds it to his advantage to do so—and not necessarily for the purpose intended. The idea that unintended consequences can flow from installed mechanisms is vital when

enacting new legislation. For what is legislation, after all, but a mechanism installed in the world. People will understandably use any such mechanism to their advantage, no matter what the intent of the legislators.

Mathematics has shaped the sciences because it enables the clear and precise expression of so many of the abstractions important to science. Christos Papadimitriou, a professor of computer science at UC Berkley, in a talk given originally in 2007 and repeated many times since, described how computer science is developing a relationship to the sciences similar to that held by mathematics. Science increasingly builds its models in computational terms. And it is often the computational questions that provide the best ideas about how to approach open questions. As Papadimitriou put it, "Computational research transforms the sciences … by providing a novel and powerful perspective which often leads to unforeseen insights.

In summary, computer science has already transformed the world through the artifacts it has developed. It has and will continue to transform the world even more profoundly through its concepts.

References

Abbott, Russ (2009 to appear) "Bits don't have error bars," *Philosophy and Engineering: An Emerging Agenda*, Springer. Originally presented at the *Workshop on Philosophy and Engineering*, Delft, The Netherlands, 2007,

Abbott, Russ (2009) "The reductionist blind spot," *Complexity*, 14/5 (May 2009) pp 10-22.

Abbott, Russ (2008) "If a tree casts a shadow is it telling the time?" *International Journal of Unconventional Computing*. 4/3, pp 195-222. Best presentation award at the *Conference on Unconventional Computing*, York, UK, 2007.

Kim, Joan and Russ Abbott (to appear 2009 or 2010) "Unintended consequences," *Proceedings of 2007 Asian-Pacific Conference on Computing and Philosophy*, Bangkok, Thailand, 2007.

Papadimitriou, Christos (2007) "The Algorithmic Lens: How the Computational Perspective is Transforming the Sciences" *Federated Computer Research Conference*, June 11, 2007.

Wing, Jeanette (2006) "Computational Thinking," *CACM,* 49/3 (March 2006) pp 33 – 35.