BLOG@CACM

The *Communications* Web site, http://cacm.acm.org, features more than a dozen bloggers in the BLOG@CACM community. In each issue of *Communications*, we'll publish selected posts or excerpts.

twitter
Follow us on Twitter at http://twitter.com/blogCACM

DOI:10.1145/3037383 http://cacm.acm.org/blogs/blog-cacm

The Slow Evolution of CS for All, the Beauty of Programs

Mark Guzdial considers the steps needed to reach the goal of CS for All, while Robin K. Hill ponders the aesthetics of programming.

Mark Guzdial Taking Incremental Steps Toward CS for All

http://bit.ly/2gCFpSM
November 28, 2016

At the end of October, the Expanding Computing Education Pathways (ECEP) alliance organized a summit with the White House Office of Science and Technology Policy (OSTP) on state implementation of the President's CS for All initiative. You can see the agenda at http://bit.ly/2ifPVwY and a press release on the two days of meetings at http://bit.ly/2iMvyek. I learned a lot at those meetings; one insight I gained was that the CS for All initiative will succeed in increments. U.S. states are developing novel, incremental approaches to CS for All.

The event's second day was focused on teams from the 16 states and Puerto Rico in the ECEP Alliance. At a session on teacher certifications, some of the attendees were concerned with what they saw as lowering standards in order to get more certified teachers. "We have a shortage of doctors in rural areas. That doesn't mean we make it easier to become a doctor!" That made sense to me, but then I heard others push the metaphor a bit. Adding more nurses and more physician assistants does improve quality of care, and it is less expensive to have more of these health care providers than to produce enough doctors.

Only a few U.S. states offer CS teacher initial certification, which requires a choice to become a CS teacher while still an undergraduate and take years of classes. Georgia and California, like several other states, offer an add-on certification ("endorsement") teachers can earn after gaining a certification in something else. An endorsement typically still requires multiple semester-long courses. Utah has one of the most innovative CS teacher add-on certification schemes, with three levels: an initial level that requires only some summer professional development, and two further levels requiring post-secondary courses.

Leigh Ann DeLyser hosted a great session about CSNYC and the new CS for All Consortium. CSNYC is charged with implementing Mayor Bill de Blasio's initiative to make CS education available to all students in all grades in all New York City schools by 2025. DeLyser told us CSNYC is defining the Mayor's initiative as a school-based mandate. Even 10 years and $81 million isn't enough to provide certified, full-time CS teachers in every school so every student gets a CS course.

Rather, every school must offer to every student in every grade a high-quality CS learning experience. Maybe that's a full course, like the BJC CS Principles curriculum now in NYC schools. Alternatively, it might be a Bootstrap unit in an algebra class, or a CT STEM activity that uses StarLogo to achieve NGSS science learning goals. It's a reasonable incremental approach towards CS for All.

New Hampshire, one of the newest ECEP states, is exploring micro-certifications. Rather than getting a certification as a CS teacher, a mathematics or science teacher might get a micro-certification to demonstrate proficiency in using a computer science approach in their teaching. There might be micro-certificates in Bootstrap, CT STEM, or Project GUTS for middle school science.

We want a future where computer science is taught by certified teachers and is as universally available as mathematics and science classes are today in most U.S. high schools. That's the vision Briana Morrison and I wrote about in *CACM* (http://bit.ly/2iIFeEc). Along the way, we need ways of growing CS education where we develop teachers who know about and teach computer science, even if not full-time, certified CS teachers.

Robin K. Hill
What Makes a Program Elegant?

http://bit.ly/2e2U6yK
October 11, 2016

A subfield of philosophy is aesthetics, in which we attempt to understand beauty. Is beauty universal? Does it make us better people somehow? Why do we focus on beauty, not ugliness? A ready application of this question to computer science (CS) addresses program elegance. Most programmers, or so I believe, would agree some programs are elegant, and elegant programs are better than others, and experienced programmers, or so I believe, generally agree on which programs are elegant.

The criterion of efficiency looms large in production programming, and appears in comment on elegance on the Web, for instance by Perrin (http://bit.ly/2ih2IhR). A program should be brief, but not a slave to brevity. An elegant design artifact is sleek and spare in its utility. An elegant program is minimally gratuitous. Consider Binary Search (of an ordered sequence) as opposed to Sequential Search, or Quicksort as opposed to Insertion Sort (http://bit.ly/2j7ldcx). Sequential Search tediously examines each (ordered) item, but does not have to; Bubble Sort tediously exchanges many items that will have to be moved again. To find the first *n* prime numbers, we can tediously test each for divisors or we can deploy the Sieve of Eratosthenes. Efficiency helps make the Sieve, Binary Search, and Quicksort elegant. We have our first criterion for elegance, **(1) minimality**, encompassing both shortness and simplicity.

Let's avoid features of programs depending on source code syntax, or compilers, or I/O mechanisms, or memory handling. A program that minimizes temporary variables, directly evaluating expressions instead, is "better," but we do not address the question of aesthetics at that level, nor at the level of self-describing identifiers, nor documentation, nor modularity, nor design patterns. A program also becomes better as it includes more error-checking, which does not strengthen, and may weaken, its elegance even as it enhances its quality.

Simplicity by itself can't be enough; Bubblesort is a simple program. (I would count Boyer-Moore String Search as elegant, though it's complicated.) Brevity by itself can't be enough; the C loop control `while(i++ < 10)` is terse, excelling in brevity, but its elegance is debatable. I would call it, in the architectural sense, brutalism. Architecture provides nice analogues because it also strives to construct artifacts that meet specifications under material constraints, prizing especially those artifacts that manifest beauty as well (http://bit.ly/2j8AMkN).

A factor that looms larger in CS than in architecture or other disciplines is correctness. A building may be regarded as elegant even if marginal parts of it are uncomfortable, but no program that does not work is regarded as elegant. This gives us another criterion, **(2) accomplishment**—the program does what it is supposed to do. Though included in the list of desiderata here, failure on that criterion is fatal rather than detrimental.

Constraints under which programming is done impose a context without which the elegance cannot be appreciated. We must understand the problem, the tools, and materials, to appreciate the solution. Expertise is necessary. Examining many student programs over many years refines an appreciation ever more impressed by work that does it all with graceful assurance and economy. Elegance, therefore, is doubly relative—to the context of the work and to the background of the observer.

Bitmap Sort, as presented by Jon Bentley (http://bit.ly/2ikzqSE) in a classic "Programming Pearls" column, is still worth studying. To sort *n* unique integers in a fixed range 0 to *m*, we rearrange them through a comparison-based sort such as Quicksort, or we initialize a bit array, indexed by 0 to *m*, to *false*, and then for each integer input, flip its bit to *true*. A pass through the resulting array, during which the indices of the *true* bits are output, gives us the sorted list. This is nice, and elegant, even relative to Quicksort, but only works on a set of unique values (as described); recognition of situations that meet that restriction distinguishes the programmer of elegance.

We are ducking hard questions about implementations at various levels of translation, and whether they should count toward or against elegance, and we will continue to do so. In fact, what I have been describing is not programs in source code terms, but algorithms. Brevity, or minimality, is a salient feature of code, but a subtle feature of algorithms; what we want is minimality in terms of the solution, however that solution is expressed. Yet another more general concept of spareness is at play in elegance, something like restraint. This gives us a criterion of **(3) modesty**. An example that flouts it comes right off the very first page of another classic, Kernighan and Plauger's *Elements of Programming Style* (http://bit.ly/2ikHDq8):

```
DO 14 I=1,N     DO 14 J=1,N 14 V(I,J)
= (I/J)*(J/I)
```

This exploits the FORTRAN compiler's truncation of integer division results to populate a matrix *V* with zeroes everywhere except the diagonal, where the values are one; that is, it initializes *V* to the *NxN* identity matrix. This is clever and short, but oh, dear, it's implementation-dependent, therefore fragile; it's obscure and ostentatious. Such virtuosity is unfortunate, yet hard to resist. (Kernighan and Plauger propose the obvious initialization to zero throughout, followed by a loop that assigns the value one to each *V*(*N*,*N*).)

What else counts? An elegant program confers a sense of satisfaction, of enlightenment. Let's call this criterion, especially characteristic of program artifacts, **(4) revelation**—the program shows us something new about its task, or brings to the fore something we forgot. Eratosthenes' Sieve shows us, or reminds us, multiples are the "not-primes." Bitmap Sort shows us, or reminds us, the integers are already ordered; they come as a sequence, so sorting can be accomplished by an indication of presence only. Boyer-Moore String Search shows us strings are just as distinct backward as they are forward.

The criteria for program elegance suggested here are **(1) minimality**, **(2) accomplishment**, **(3) modesty**, and **(4) revelation**, all rooted in the particulars of the problem. Are these criteria necessary? Sufficient? Inadequate? Because of dependence on the problem at hand, sometimes with complex circumstances, a wide range of examples of elegant programs is difficult to come by. What exemplars stand out in your world? C

Mark Guzdial is a professor at the Georgia Institute of Technology. **Robin K. Hill** is an adjunct professor at the University of Wyoming.