A reprint from
American Scientist
the magazine of Sigma Xi, The Scientific Research Society

Cultures of Code

Three communities in the world of computation are bound together by common interests but set apart by distinctly different aims and agendas.

Brian Hayes

Kim studies parallel algorithms, designed for computers with thousands of processors. Chris builds computer simulations of fluids in motion, such as ocean currents. Dana creates software for visualizing geographic data. These three people have much in common. Computing is an essential part of their professional lives; they all spend time writing, testing, and debugging computer programs. They probably rely on many of the same tools, such as software for editing program text. If you were to look over their shoulders as they worked on their code, you might not be able to tell who was who.

Despite the similarities, however, Kim, Chris, and Dana were trained in different disciplines, and they belong to different intellectual traditions and communities. Kim, the parallel algorithms specialist, is a professor in a university department of computer science. Chris, the fluids modeler, also lives in the academic world, but she is a physicist by training; sometimes she describes herself as a computational scientist (which is not the same thing as a computer scientist). Dana has been programming since junior high school but didn't study computing in college; at the startup company where he works, his title is software developer.

These factional divisions run deeper than mere specializations. Kim, Chris, and Dana belong to different professional societies, go to different conferences, read different publications; their paths seldom cross. They represent different cultures. The resulting Balkanization of computing seems unwise and unhealthy, a recipe for reinventing wheels and making the same mistake three times over. Calls for unification go back at least 45 years, but the estrangement continues. As a student and admirer of all three fields, I find the standoff deeply frustrating.

Certain areas of computation are going through a period of extraordinary vigor and innovation. Machine learning, data analysis, and programming for the web have all made huge strides. Problems that stumped earlier generations, such as image recognition, finally seem to be yielding to new efforts. The successes have drawn more young people into the field; suddenly, everyone is "learning to code." I am cheered by (and I cheer for) all these events, but I also want to whisper a question: Will the wave of excitement ever reach other corners of the computing universe?

Setting Agendas

What's the difference between computer science, computational science, and software development?

When Kim the computer scientist writes a program, her aim is to learn something about the underlying algorithm. The object of study in computer science is the computing process itself, detached from any particular hardware or software. When Kim publishes her conclusions, they will be formulated in terms of an idealized, abstract computing machine. Indeed, the more theoretical aspects of her work could be done without any access to actual computers.

When Chris the computational scientist writes a program, the goal is to simulate the behavior of some physical system. For her, the computer is not an object of study but a scientific instrument, a device for answering questions about the natural world. Running a program is directly analogous to conducting an experiment, and the output of the program is the result of the experiment.

When Dana the developer writes a program, the program itself is the product of his labors. The software he creates is meant to be a useful tool for colleagues or customers—an artifact of tangible value. Dana's programming is not science but art or craft or engineering. It is all about making things, not answering questions.

Should these three activities be treated as separate fields of endeavor, or are they really just subdivisions of a single computing enterprise? The historian Michael Mahoney, an astute observer of computing communities, suggested that a key concept for addressing such questions is the "agenda."

> The agenda of a field consists of what its practitioners agree ought to be done, a consensus concerning the problems of the field, their order of importance or priority, the means of solving them (the tools of the trade), and perhaps most importantly, what constitutes a solution…. The standing of the field may be measured by its capacity to set its own agenda. New disciplines emerge by acquiring that autonomy. Conflicts within a discipline often come down to disagreements over the agenda: what are the really important problems?

The issue, then, is whether Kim, Chris, and Dana set their own agendas, or whether each of them has merely chosen to concentrate on se-

Brian Hayes is senior writer for American Scientist. *Additional material related to the* Computing Science *column can be found online at http://bit-player.org. E-mail: brian@bit-player.org*

computer science understand the nature of computation	computational science use computation to understand nature	software development write useful programs
Determine what can be computed.	Map natural processes onto computational ones.	Learn to manage complexity.
Determine what can be computed with finite resources.	Simulate the behavior of physical, biological, and social systems.	Map abstract concepts onto concrete program structures.
Determine what can be computed efficiently.	Capture and manage large volumes of data.	Provide tools for debugging.
Compare models of computation (e.g., classical and quantum).	Minimize numerical errors.	Provide tools for collaborative work and code sharing.
Organize data for efficient storage and retrieval.	Solve large systems of linear equations.	Manage versions and variants of code.
Define the syntax and semantics of programming languages.	Approximate the solutions of differential equations.	Define mechanisms and standards for exchanging data between programs.
Improve the human interface with computers.	Encode continuous quantities in discrete form.	Learn what factors influence programmer productivity.
Ensure the correctness of concurrent operations.	Devise ways to visualize spatial and temporal patterns, such as vector fields.	Learn what features make programming languages more expressive.

Communities that share an interest in computing but have distinct goals can be distinguished by their *agendas*: the lists of problems to solve and tasks to accomplish that members of each community agree on. The idea of defining a community by its agenda was introduced by the historian Michael Mahoney. Shown here are some possible to-do items for computer science, computational science, and software development.

lected parts of a shared agenda. There are certainly questions that would interest all three of them. A prominent example is "What can be computed efficiently?" Theoretical computer science seizes on this question as one of its most central, existential concerns, but the answer also matters to those who write and run programs for practical purposes. Thus the three groups might seem to stand on common ground. The trouble is, a theorist's answer to the question may not be much use to a practical programmer. Knowing that the worst-case running time grows as some polynomial function of the problem size doesn't actually tell you whether a specific computation will take seconds or centuries.

The issue here is not that all computer scientists are otherworldly theorists. Sometimes the theoretical challenges arise elsewhere. As a fluid dynamicist, Chris has on her agenda the tricky theoretical problem of partitioning a continuous fluid into discrete parcels suitable for processing by a digital computer. Solutions to such problems have come mainly from mathematicians, engineers, and physicists rather than computer scientists.

One of the glories of computer science in its early years was a deep analysis of programming languages. Everyone who does computing would seem to have a stake in this work. In an interesting collaboration between computer scientists, mathematicians, and linguists, the languages were classified according to their expressive power. The next project was to devise algorithms for parsing programs—breaking statements down into their basic grammatical units—and then assigning meaning to the statements. Most of this work was completed by the 1970s.

Programmers today are intensely partisan in their choices of programming languages, yet interest in the underlying principles seems to have waned. Two years ago I attended a lunch-table talk by a young graduate student who had turned away from humanities and business studies to take up a new life designing software. She had fallen in love with coding, and she spoke eloquently of its attractions and rewards. But she also took a swipe at the traditional computer science curriculum. "No one cares much about LR(1) parsers anymore," she said, referring to one of the classic tools of language processing. The remark saddened me because the theory of parsing is a thing of beauty. At the very least it is a historical landmark that no one should pass by without stopping to read the plaque. But, as Edith Wharton wrote, "Life has a way of overgrowing its achievements as well as its ruins."

Roots of Computing

Schisms in the computing community can be traced back all the way to the beginning of the digital electronic era, circa 1950. The designers of the early machines, such as the ENIAC in the United States and the EDSAC in Britain, wove together ideas from sources that must have seemed unlikely bedfellows. Basic notions of how to "mechanize thought" came from mathematical logic, including the 19th-century work of George Boole on a form of algebra in which the elements are not numbers but the values *true* and *false.* Electrical engineering, and in particular switching theory, provided circuits that implement Boolean operations in hardware.

Mathematical logic became one of the seed pearls on which theoretical computer science grew. Circuit theory also remains a core component of computer science and engineering. Indeed, the design and manufacture of hardware represents yet another independent computing culture.

Alongside mathematical logic and electrical circuits, there was a third tradition present at the birth of modern computing. The users of those first high-speed computing machines came mainly from applied mathematics and closely allied areas such as physics. Prominent among the users were tablemakers, who compiled tables of logarithms, trigonometric functions, and all sorts of other quantitative information. (The ostensible reason for building the ENIAC was to compile ballistic tables for artillery.) Another important constituency among the users were the numerical analysts, who devise schemes for finding approximate solutions to equations that cannot be solved exactly. Most of the interesting problems in the sciences fit this description. For

the tablemakers and the numerical analysts, the electronic computer was a problem-solving or question-answering tool; the heirs of these pioneers are today's computational scientists.

Notably absent from the planning for early computer projects was any serious discussion of programming. For each problem to be solved, a mathematician or other professional was expected to design the scheme of computation, perhaps in the form of a flow chart annotated with equations. Translating this plan into instructions suitable for the machine was viewed as a routine clerical task, requiring no intellectual engagement with the underlying ideas. In the case of the ENIAC, six women were recruited as "coders" to do this work. Three of the six had majored in mathematics in college, and all of them were absurdly overqualified for clerical duties. As it turned out, their qualifications were put to the test, because the work of preparing programs for the machine was anything but routine.

The discovery that programming presents serious intellectual challenges apparently came as a surprise to the early leaders of the field. Maurice V. Wilkes, the principal architect of the EDSAC, had an epiphany while writing the first substantial program for that machine in 1949:

> The EDSAC was on the top floor of the building and the tape-punching and editing equipment one floor below.… It was on one of my journeys between the EDSAC room and the punching equipment that … the realization came over me with full force that a good part of the remainder of my life was going to be spent in finding errors in my own programs.

Recognizing that programming requires skill and ingenuity elevated the status of the occupation. Unfortunately, not everyone benefited from this upgrade. Women who had pioneered in the art of programming were largely displaced by men—an issue the profession is still dealing with 60 years later.

Silver Bullets

By the time computers were being manufactured for commercial use, programming was recognized as a costly bottleneck. The work was tedious and slow; people good at it were hard to find; even the most talented and dedicated programmers made mistakes. Programming projects became notorious for running over budget and behind schedule. Progress in computing was threatened by a "software crisis."

The subsequent history of programming methodology can be read as an extended campaign to slay this dragon. Higher-level programming languages—closer to the vocabulary of the problem domain, further from the minutiae of the hardware—were the first weapon, and the most effective one. A regimen called structured programming tried to untangle the logic of programs by allowing only a few kinds of loops and branches. Under the banner of *modularity*, programs were to be assembled out of pretested, reusable units. Another movement called for formal proofs of program correctness. More slogans paraded by: abstraction, encapsulation, declarative programming, functional programming, object-oriented programming, design patterns, test-driven development, agile development. The sheer variety of these remedies is a hint that no one of them was a cure-all. Even now the software crisis is still with us: Witness the debacle of the Healthcare.gov website in 2013.

The debate over software quality has included repeated calls to make programming a proper engineering discipline, with recognized standards of proficiency and perhaps requirements for certification or licensing. A related trend imposed more management structure on the programming process. In the software shops of the 1960s and '70s, the way to get ahead was to rise above the actual writing of code and become a system analyst or architect.

At the same time, however, another strand of computing culture—or counterculture—was moving in the opposite direction. The enthusiasts who called themselves hackers, most famously situated at MIT among members of the Tech Model Railroad Club, saw computer programming as a puzzle to be solved, a world to explore, a medium of self-expression. They saw it as fun. They resisted the idea that only an elite with engineering credentials would be allowed access to the machinery. The notion that programming could be regulated or restricted was further undermined when personal computers became widely available and affordable in the 1980s.

The investment of effort that once printed "Hello, world" on the computer screen now brings the world itself to that screen.

Coding Is Cool Again

Another wave of irrepressible hacker enthusiasm is washing over us now, as a new generation discovers that coding is cool. Introductory programming courses, which had disappeared from many college curricula, now attract hundreds of students. At Harvard, for example, a hands-on programming course called CS50 has an enrollment of almost 900, the largest in the entire university. Online courses engage millions more. And a group called code.org is working to revive the study of computing in elementary and secondary schools.

Why this sudden infatuation with the nerdy side of life? Fad and fashion doubtless play a part. So does the prospect of creating the next billion-dollar app. And there's always excitement in joining your generation's mission to change the world. Beyond all that, I would cite one more factor. Within the past five years, programming tools have crossed a threshold of accessibility and power. It's not that we have finally found the magic elixir that makes programming easy and error-free. The learning curve is still steep. But the view from the top of the hill is spectacular. The same investment of effort that once printed the words "Hello, world" on the computer screen now brings the world itself to that screen.

In 1984 I saw a demo of a mapping program created by Michael Lesk and his colleagues at AT&T Bell Labs. The

graphics were crude by modern standards, but the program could answer geographic queries and recommend routes from point to point in the New York area. I was wowed.

A key innovation in Lesk's program was storing the terrain map in small square tiles that could be loaded into memory as needed. Twenty years later, Google Maps employed the same principle (with better graphics) to create the illusion that the computer screen is a window onto a vast unfurling map of the whole planet. New tiles are fetched over the network whenever you move the window or zoom in and out. I was wowed again.

Google Maps was state-of-the-art wizardry in 2005; in 2015 anyone can do it. With a dozen lines of code—plus an open-source library called Leaflet and a free web service that supplies the map tiles—you can create your own mapping program, offering the viewer the same breathtaking window-on-the-world experience.

The grizzled curmudgeon in me wants to object that this instant cartography is not *real* programming, it's just a "mashup" of prefabricated program modules and Internet resources. But building atop the achievements of others is exactly how science and engineering are supposed to advance.

Still, a worry remains. How will the members of this exuberant new cohort distribute themselves over the three continents of computer science, computational science, and software development? What tasks will they put on their agendas? At the moment, most of the energy flows into the culture of software development or programming. The excitement is about *applying* computational methods, not inventing new ones or investigating their properties. In the long run, though, *someone* needs to care about LR(1) parsers.

Guy Lewis Steele, Jr., one of the original MIT hackers, worried in the 1980s that hackerdom might be killed off "as programming education became more formalized." The present predicament is just the opposite. Everyone wants to pick up the knack of coding, but the more abstract and mathematical concepts at the core of computer science attract a smaller audience. The big enrollments are in courses on Python, Ruby, and JavaScript, not automata theory or denotational semantics.

I would not contend that mastery of the more theoretical topics is a prerequisite to becoming a good programmer. There's abundant evidence to the contrary. But it *is* a necessary step in absorbing the culture of computer science. I am sentimental enough to believe that an interdisciplinary and intergenerational conversation would enrich both sides, and help in knitting together the communities.

Bibliography

Baldwin, Douglas. 2011. Is computer science a relevant academic discipline for the 21st century? *IEEE Computer* 44(12):81–83.

Denning, Peter. 1985. What is computer science? *American Scientist* 73:16–19.

Felleisen, Matthias, and Shriram Krishnamurthi. 2009. Viewpoint: Why computer science doesn't matter. *Communications of the ACM* 52(7):37–40.

Fritz, W. Barkley. 1996. The women of ENIAC. *IEEE Annals of the History of Computing* 18(3):13–28.

Gramelsberger, Gabriele (ed.). 2011. *From Science to Computational Sciences: Studies in the History of Computing and Its Influence on Today's Sciences*. Zürich: Diaphanes.

Mahoney, Michael Sean. 2011. *Histories of Computing*. Cambridge, MA: Harvard University Press.

Wegner, Peter. 1970. Three computer cultures: Computer technology, computer mathematics, and computer science. *Advances in Computers* 10:7–78.