2017-12

The Forgotten Engineer

Denning, Peter J.

ACM

Denning, Peter J. "The forgotten engineer." Communications of the ACM 60, no. 12 (2017): 20-23.
http://hdl.handle.net/10945/57259

DOI:10.1145/3152912

Peter J. Denning

The Profession of IT
The Forgotten Engineer

Engineering has been marginalized by the unhealthy belief that engineering is the application of science.

We live in a time that reveres science. It was not always this way: in much of the previous centuries, engineers were heroes. In the late 20th century, however, the engineer's image eroded because science seemed to offer more hope with difficult problems and because technology seemed to inflict collateral damage through such issues as pollution, exploitation of nature, weapons of mass destruction, and massive surveillance.

Our modern fascination for science is marginalizing engineering. This is especially bad for computer science and engineering. For instance, we routinely teach programming as a set of abstractions to be applied rather than a skill of design to satisfy customers. We routinely make claims about what computing can theoretically accomplish without knowing that we can deliver.

Not long ago, *Science* magazine distributed a subscription solicitation that offered a T-shirt bearing on the front the image of a Leonardo da Vinci flying machine and on the back the inscription "Aviation. Brought to you by science." This slogan was an XL misrepresentation of how aviation came to be.[8]

The linear model of research and development behind this slogan has been repeatedly challenged and disproved. *Science* advertised this popular fallacy on a T-shirt—with the worst example imaginable.

The Wright brothers did look to science for help in answering fundamental questions about wings and propellers.[8] The premier institution of the day, the Smithsonian, was unable to help them. These bicycle mechanics and self-taught engineers conducted their own experiments, spending many hours studying the flights of birds to understand what enabled them to soar, eventually concluding that wing warping would be a key to controlling gliders and powered aircraft. Only when they had a sketch of a mechanical concept could they begin to apply science to its development. *Science* did not bring us aviation. Rather, the Wright brothers built working flying machines that opened the possibility of aviation and gave birth to a new science, aeronautics.

Much the same happened with computing technology. The first digital electronic computer was built in Germany in 1938 by Konrad Zuse, who was educated as a civil engineer. John Atanasoff and Clifford Berry built a digital computer that solved linear equations and demonstrated it in 1942. The first digital computer capable of any computable function—ENIAC (1945)—was built by electrical engineers J. Presper Eckert and John Mauchly under a U.S. Army contract. They spent many hours tinkering to find reliable electronic logic circuits. In 1945, they joined with Herman Goldstine, Arthur Burks, and John von Neumann to design a stored-program machine, which they demonstrated would be more powerful and significantly less complex than ENIAC.

Although Alan Turing, whom many computer scientists revere, proposed his Turing machine model of computation in 1936, his work was known only to a handful of mathematical logicians, and completely unknown to the engineers who built the first electronic computers.[2,5] It was not until the 1950s,

IMAGE BY ANDRIJ BORYS ASSOCIATES

when the first academic programs were being born, that Turing's work offered the theoretical basis to make computer science credible as a new department in universities. In other words, as important as Turing's work is, it did not inform or inspire the first electronic computers or the stored program concept. Instead, the success of the first stored-program electronic computers created the opening for Turing's work to become important. Yet we have our own T-shirts proclaiming that Turing was the father of digital computing.

Incongruities

Most engineering associations and the accrediting body ABET define engineering as "application of science and mathematics to finding practical solutions to problems," a definition that makes engineering an application of science or a branch of science. Not that long ago, some engineering associations defined engineering as the *science and art* of designing and making structures. Petroski comments, "Once that design is articulated by the engineer as artist, it must be analyzed by the engineer as scientist in as rigorous an application of the scientific method that any scientist must make."[7] What happened to the idea that engineering is *both* science and art?

When I recently reviewed the status of computational thinking in education,[3] I noticed that the recommendations for curricula focus almost exclusively about the science-math side of computational thinking and have little or nothing to say about architecture or design side. This seemed odd because programs are designed to control machines, and moreover most of the new jobs in computing are in architecture and design. Why are architecture and design not showing on the computational thinking radars?

Software pioneer David Parnas has long been a critic of the adopted approach to teaching software engineering, which seemed to him to downplay an engineering view in favor of a math-theory view. He wants students to get plenty of practice programming and designing programs to meet customer requirements.[6] He notes that the crux of software development is "multiperson development of multiversion programs"—in other words, teams and organizations building families of software. Curricula that lack a strong emphasis on design cannot prepare their graduates for this. What happened to the engineering in software engineering?

Engineering has helped all the social, life, and physical sciences advance by providing tools and instruments. Engineering has helped civilization advance by providing reliable infrastructures such as electricity, transportation, water, and food. Since the 1980s, the engineering of supercomputers and networks has given birth to a raft of new branches of science, mostly called "computational X" where "X" names a traditional science. Most computing professionals today are heavily engaged in engineering—they design systems for customers and experiment to find out what works. Why has the engineering outlook lost favor in many CS departments?

Much engineering is oriented around design of systems and structures that will be reliable, safe, and secure. Aside from "design of experiments and models," scientists hardly ever discuss design. How could engineering be a subset of science when its main concern is not a concern of science?

Distinctions

It is clear that science and engineering are distinct enterprises with different ways of looking at the world. Yet they cannot advance without interacting with each other. Historians Bowler and Morus trace the evolution of the steam engine and the telegraph in the 1700s and 1800s.[1] They debunked the modern myths of one-directional flow from science to technology. In those days, there was no practical difference between science and technology. It seems that the distinction between science and engineering is recent—introduced in the late 1940s when Vannevar Bush advocated the establishment of the U.S. National Science Foundation for government support of basic research.

Given the contemporary definitions, I have found three distinctions between engineering and science particularly helpful. The first concerns the nature of their work. Engineers design and build technologies that serve useful purposes, whereas scientists search for laws explaining phenomena. Design is among the most common words of engineering, whereas it is uncommon in science. Design in engineering is a process of finding practical, safe, cost-effective implementations. Whereas scientists have a knack for finding recurrences, engineers have a knack for listening to clients and proposing technologies of value to them.

Design in computing is fundamentally an engineering practice.

The second main distinction is how scientists and engineers regard knowledge. Scientists treat knowledge as data and information that have been organized into a "body of knowledge" that is then available for anyone to use. The scientific method is a process of standard, outside observers gathering and weighing evidence in support of claims that might be added to the body. Engineers treat knowledge as skillful practices that enable design and building of tools and technologies. Engineers are not disinterested outside observers; they are immersed in the communities of use. They embody practices for building, maintaining, and repairing technologies; attending to reliability, dependability, and safety in the context of use; and following engineering standards and codes of ethics.

The third main distinction concerns the role of abstractions and models. Science emphasizes models, and engineering machines. There is a fundamental distinction between modeling machines and building them. Abstractions are useful for what they leave out. Machines are useful for what they leave in. Hardware and software are interchangeable to the theorist, but not to the engineer.

The familiar phrase "devil is in the details" is an engineer's motto. Engineers must get the details right for systems to work. Scientists want to eliminate the details so that the recurrences stand out.

The accompanying table summarizes, compares, and contrasts how computer scientists and engineers tend to view design. These are dispositions and tendencies, not formal definitions. Computer scientists need to function with both worldviews.

Dispositions toward design in computing.

Science	Engineering
A design is a plan or a blueprint for a model or an experiment	Design is a process of proposing systems that meet customer concerns
Designs aim to reveal causes	Designers aim to harness naturally occurring effects
Designers find and validate models	Designers align software with user practices
Designers work with proven abstractions and models that omit inessential details	Designers know that every detail counts for a reliable and safe product or system
Designers are ultimately concerned with whether claims are true	Designers are ultimately concerned with whether products or systems work
Designers are objective observers detached from communities	Designers are immersed in their communities
Designers aim to understand the world	Designers aim for working implementations that can change the world
Correctness and validation measure success	Client satisfaction measures success
Mistakes can be eliminated with formal verification	Mistakes and defects are inherent, the system must tolerate them
Good designs can be formally verified so that they will work the first time	Good designs are fault tolerant so that they continue to be reliable and safe even when faults and defects appear
Good designs rule out contingencies or surprises	Designers work with contingencies and surprises
Experiments validate hypotheses	Tinkering is experimenting to find what works
What we know is expressed as our body of knowledge	What we know is expressed in our practices, standards, and lore about what works
Engineering and technology will apply the science	We build technologies to have something to apply science to

Engineering and Science in Computing

As we noted in 1989,[4] science, engineering, and mathematics are irrevocably interwoven in the fabric of computing. Every computing technology has a science, an engineering, and a mathematics aspect. Computing cannot be dissected into the three components. It is not a branch of science, engineering, or mathematics. In computing, design means developing practical systems with the aid of mathematical tools such as program verifiers, practices from science such as taxonomies of design patterns, and validation methods such as careful statistical testing. Design in computing is fundamentally an engineering practice.

In computing, we work closely with the notion that programs can be expressed as structures of abstract objects, and the useful work happens when those abstract objects control machines that affect the world. The science-math mind plays a strong role with structuring the abstractions; the engineering mind plays a strong role with bringing the effects into the world. The field cannot survive if these two aspects do not maintain a synergized balance.

These arguments are not new. In his 1968 ACM A.M. Turing lecture, award recipient Richard Hamming argued that the computer is at the heart of computing; without it, almost everything computing professionals do would be idle speculation. In the past two decades, we have added natural information processes, such as DNA transcription, to what we study, but the computer remains the heart. Every programming language is a means for designers to control an abstract machine that when simulated produces useful and practical results. Computer science graduates, Hamming argued, must learn design in the context of bringing value to users.

This is why I am concerned that our academic departments embody too strong an emphasis on the theoretical side of computing. The engineering side has been diminished in the process. Recent reforms to computing curricula have introduced a new first course "CS principles." Most of the content of these courses is concepts relating to programming and algorithm organization. A few departments, more so in engineering, use design courses and Raspberry Pi or Arduino labs to introduce students to the field. The teachers are always surprised by how much the students accomplish in the role of designers without much grounding in the science of the field. More departments ought to consider starting students with a design course.

The result is curricula that encapsulate computing inside a boundary of math-science-theory and diminish crucial engineering aspects in architecture and design. This is unhealthy because most of the jobs for which our graduates are aiming are much more strongly oriented around engineering than science. It is no wonder that employers complain that CS graduates do not fit and need extensive training and hand holding to become profitable employees.

It bothers me that all the modern advances—in AI, machine learning, big data, cloud computing, and computer security—are touted as triumphs of science rather than what they really are, achievements of engineering and science working together.

Conclusion

Science and engineering need each other. Neither is the application or fulfillment of the other. Science emphasizes the discovery of recurrences. Engineering seeks to harness effects before the recurrences are fully known. Science moves in when the effect has proved useful and we seek to understand it better, optimize it, make it more reliable, and exploit its recurrences for prediction. Science takes care of abstractions, engineering the details that enable abstractions to work. The marriage of science and engineering in computing is critical for the continued health of the field. C

References

1. Bowler, P.J. and Morus, I. *Making Modern Science: An Historical Survey.* University of Chicago Press, 2010.
2. Daylight, E. A Turing tale. *Commun. ACM 57*, 10 (Sept. 2014), 36–38.
3. Denning, P. Remaining trouble spots with computational thinking. *Commun. ACM 60*, 6 (June 2017), 33–39.
4. Denning, P. et al. Computing as a discipline. *Commun. ACM 32*, 1 (Jan. 1989), 9–23.
5. Haigh, T. Actually, Turing did not invent the computer. *Commun. ACM 57*, 1 (Jan. 2014), 36–41.
6. Parnas, D. David Parnas speaks of software engineering. CCSL Centro de Competéncia em Software Livre, 2014; http://ccsl.ime.usp.br/en/news/14/09/17/david-parnas-speaks-software-engineering-ccsl
7. Petroski, H. *To Engineer is Human: The Role of Failure in Successful Design.* Vintage, 1992.
8. Petroski, H. and Denning, P. Your science T-shirt doesn't fly. ACM *Ubiquity* (Dec. 2016); http://ubiquity.acm.org/blog/your-science-t-shirt-doesnt-fly/

Peter J. Denning (pjd@nps.edu) 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 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.

The author thanks Fernando Flores and Henry Petroski for conversations with insights about science and engineering.

Calendar of Events

December 4–6
K-CAP 2017: Knowledge Capture Conference, Austin, TX, Sponsored: ACM/SIG, Contact: Oscar Corcho, Email: ocorcho@gmail.com

December 5–8
UCC '17: 10th International Conference on Utility and Cloud Computing, Austin, TX, Co-Sponsored: Other Societies, Contact: Alan Fraser Sill, Email: Alan.Sill@ttu.edu

December 11–15
Middleware '17: 18th International Middleware Conference. Las Vegas, NV, Sponsored: ACM/SIG, Contact: Anshul Gandhi, Email: anshul@cs.stonybrook.edu

December 12–15
CoNEXT '17: The 13th International Conference on Emerging Networking Experiments and Technologies, Incheon, Republic of Korea, Contact: Taekyoung Kwon, Email: tkkwon98@gmail.com

2018

January

January 7–10
GROUP '18: 2018 ACM Conference on Supporting Groupwork, Sanibel Island, FL, Sponsored: ACM/SIG, Contact: Michael Prilla, Email: prilla.michael@googlemail.com

February

February 21–24
SIGCSE '18: The 49th ACM Technical Symposium on Computing Science Education, Baltimore, MD, Sponsored: ACM/SIG, Contact: Tiffany Barnes, Email: tiffany.barnes@gmail.com