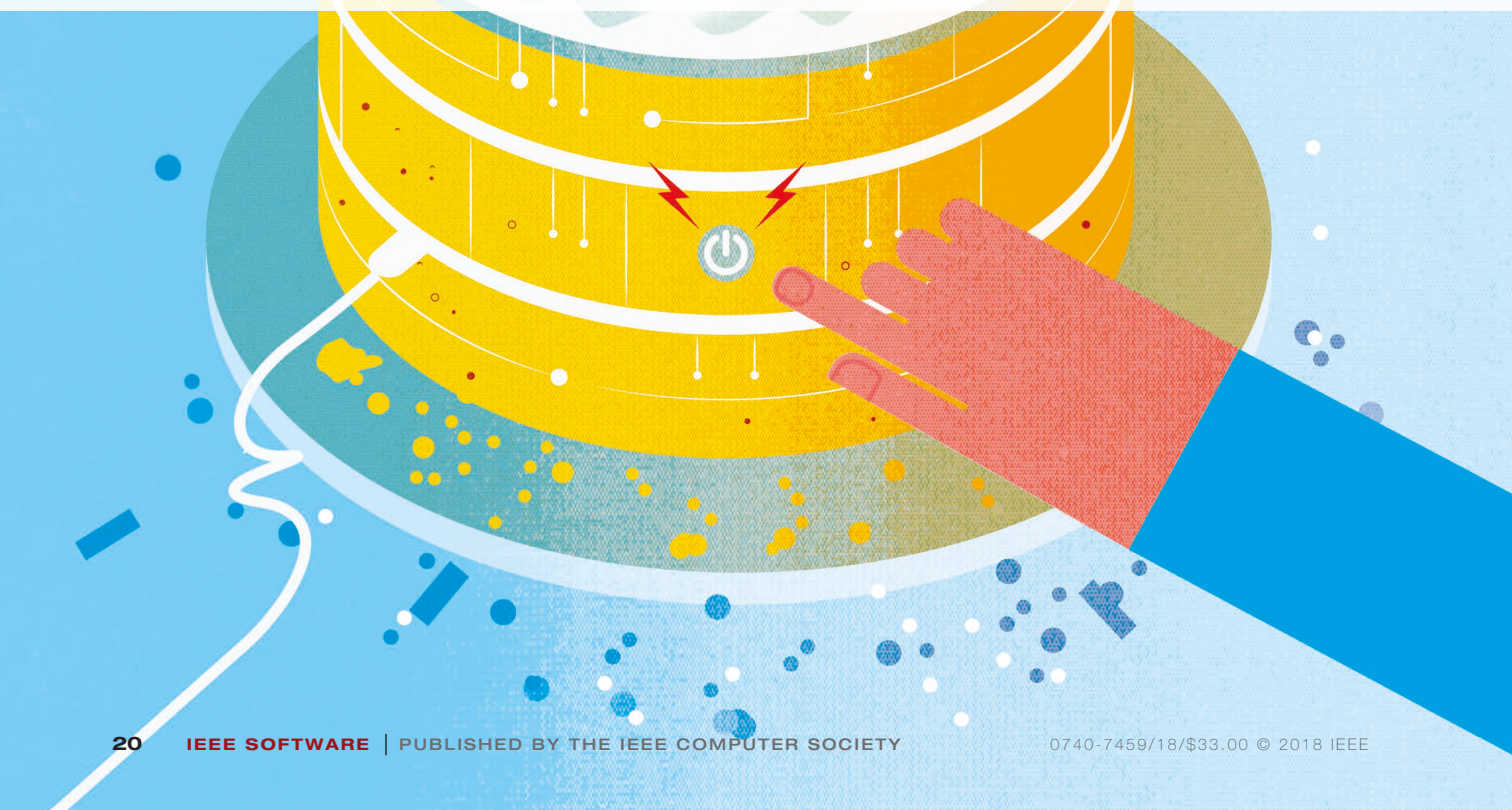
50 Years of Software Engineering

Hakan Erdogmus, Carnegie Mellon University

Nenad Medvidović, University of Southern California

Frances Paulisch, Siemens Healthineers



0740-7459/18/$33.00 © 2018 IEEE

DEFINED PLAINLY, SOFTWARE engineering is the building of nontrivial software-intensive systems in disciplined ways subject to a set of constraints. It's widely recognized as a sociotechnical process. On the social side, it involves a significant people and team component. On the technical side, its theoretical underpinnings are computer science and, in particular, the science of programming. However, it's much more than just programming.

The Uniqueness of Software Engineering

Software engineering is a unique line of endeavor in a number of ways. Some of these ways naturally stem from the uniqueness of software itself. Its relatively few technical principles—separation of concerns, isolation of change, abstraction, modularity and reuse, to name some—have been applied to result in thousands of languages, methods, techniques, and tools. More than three decades ago, Fred Brooks argued eloquently that none of our solutions, no matter how ingenious, will ever be able to fully eliminate the four fundamental difficulties—corresponding to the four fundamental sources of uniqueness—that software engineers face: software's complexity, conformity, changeability, and invisibility.[1] Nevertheless, we're still able to build systems of startling sophistication that, despite their imperfections, power the modern world.

A central aspect that makes software engineering unique is software's invisibility; it's not subject to the laws of nature. You can build a software system with infinitely intricate and detailed connections. This is unheard of in the physical world, which has hard constraints. For example, you can put only so many transistors on a silicon wafer that's a fraction of a square inch. To further complicate things, a tiny, seemingly local error—a single bit—can potentially have huge consequences for the physical system the software controls.

Software engineering is also a unique brand of engineering because, as a discipline, we're not always … disciplined. Our principles, methods, techniques, and tools are applied with various degrees of diligence. Our systems are notoriously unevenly documented. Case studies, anecdotes, and memes about software bugs and failures abound. Some are amusing because they're caricatures of true events whose consequences aren't that severe. Others are not because they're true and put living beings and assets of considerable economic or intangible value in peril.

Unlike many other engineering disciplines, software engineering is extremely broad. It's the foundation of a huge part of our everyday lives. Its construction is perhaps more like creating and evolving an architectural marvel than building a bridge in civil engineering, constructing a motor in mechanical engineering, or creating new molecules in chemical engineering.

Software is prone to being tampered with from thousands of miles away. This can cause the systems it controls to be prone to hacking in uniquely dangerous, devastating, and undetectable ways that wouldn't be possible in its absence.

Software is amenable to automated analysis because software itself is data, and all the artifacts generated during its production are also software. This bootstrapping quality creates unique opportunities to study and understand both software's behavior and the nature of its underlying engineering process in ways not readily possible with physical artifacts.

Finally, the software engineering community's nature and diversity—or, you might argue, the lack of a coherent community—make the field unique. Many persons developing software are self-taught and didn't learn their craft "by the book." Unlike in other engineering disciplines, the term "software engineer" doesn't signify a widely accepted set of qualifications backed up by standards, proper apprenticeship, accreditation, or certification. Significant gaps exist among the academics who teach software engineering, the researchers who study it, and the professionals who practice it. Although all this leads to a diversity of opinions, ideas, and perspectives that fuel software innovation, it creates a multitude of irreconcilable disagreements that sometimes block the progress achievable through common understanding.

A Uniquely Diverse Community

A sign of a well-established, mature discipline—and community—is agreement on how to solve common problems. In software engineering, however, solutions tend to depend highly on the background and experience of the person solving the problem, rather than on a common understanding across the community. At times this makes the field more craft-like than engineering-like. This fuzziness sometimes stems from the inherent multiplicity of dimensions you must consider in a solution and, in each dimension, a whole spectrum of choices that appear to work to widely varying degrees depending on

the problem's context and what's accepted as "good enough."

An infinite number of combinations of unpredictable viability and effectiveness arise from such choices—for example,

- the degree of consideration of underlying business models, from one-time licensing to subscription services;
- the type of delivery model, from discrete delivery of a single product to a continuous value flow of features in an evolving product;
- the abstraction level, from low-level coding to model-driven approaches;
- the management of intellectual property, from proprietary approaches to a high degree of openness and transparency;
- the architectural choices, from monolithic systems to a decentralized set of systems or microservices; and
- the level of security, from none to approaches more oriented toward prevention or detection.

Even when fads become clear trends that start to establish themselves and push systems toward particular regions of the solution space, the best solutions' context dependence still prevents straightforward generalizability.

Extreme points in the solution space give rise to dogma and dichotomies. Not surprisingly, the truth often lies somewhere in the middle, based on balancing a large number of trade-offs. The solution also depends on the nature of the problem: whether it's *simple*, *complicated*, *complex*, or *chaotic*, to use the Cynefin framework's terminology.[2] Solutions that are appropriate for complicated problems are typically inappropriate for complex or chaotic problems. The first step for us as software engineers is to be aware of such differences.

A clear manifestation of diverse perspectives is the existence of different camps that still disagree on fundamental issues and questions, such as these:

- Is agile software development better than carefully planned development, and if so, under what circumstances?
- Are formal methods essential, or even useful, or are they just an intellectual exercise that gets in the way of building real-world systems?
- Should a system's architecture be designed, or does it emerge organically during development? If the former is the case, how, when, and to what extent is architectural design appropriate?
- Are standards critical to software engineering's maturation as a discipline, or are they just an impediment to pragmatic development?
- Should software development rely on up-front planning or on a more adaptive approach in which you strive to decide at the last responsible moment?
- Does success depend primarily on technical skills or peopleware?
- What constitutes good software: software that appears to work most of the time and does the job? Or delights users in special ways? Or never crashes? Or is easy to understand and maintain? Or has an elegant design? Or is thoroughly tested? Or is proven to be correct? Or is a combination of some or all of these criteria?
- Should systems be decomposed with respect to functionality or dataflow?

Software engineering provides some foundations for answering these questions and many more like them. However, an ever-increasing number of moving parts and considerations are layered on top of those foundations. Both contemporary software engineering research and novel field practices address the themes underlying these questions. Alas, the answers are challenged by the fact that each of us has his or her own biases regarding these themes, and the lion's share of self-taught practitioners are often outside the academic and research community's reach. Without tight integration between research and practice, how can we evaluate the associated trade-offs in a systematic and widely agreed-upon manner, as we would do in a mature engineering discipline?

A Uniquely Distinct Beginning to Celebrate

Another unique aspect of software engineering is that it has a birth date of sorts: Monday, 7 October 1968. On that day, the first software engineering conference, sponsored by NATO, opened in Garmisch, Germany.[3] This isn't to say that software engineering didn't exist before Garmisch. Many well-documented developments predate it. Complex software was certainly being written—and, in fact, engineered—well before 1968. For example, one of the foundational readings in software engineering—Fred Brooks' *The Mythical Man-Month*—describes a large software engineering effort at IBM from the mid 1960s.[4] But the Garmisch conference was instrumental in introducing

software engineering as a discipline and indirectly kicking off the massive pedagogical, research, governmental, and commercial enterprise that software engineering is today.

And that, of course, inspired this theme issue marking software engineering's 50th birthday. We collected a range of contributions—from pioneers and well-established software engineers, to younger contributors whose imprint on the field is perhaps yet to come. These contributions come in a variety of formats that provide a balanced look at our field's past, present, and likely future. The topics include both timeless ideas that appeared to fade for a while, only to pop up again in a new incarnation, and entirely new paradigms that have disrupted the field.

According to many folks, the term "software engineering" appeared before 1968 courtesy of Margaret Hamilton, a software pioneer who made pivotal contributions to putting a human on the moon. As Ken Power tweeted during Hamilton's keynote address at the 2018 International Conference on Software Engineering, "Neil Armstrong may have been the first to walk on the moon, but Margaret Hamilton's software was the first to run on the moon."[5] So, it's only apt that we include a contribution by this celebrated pioneer. Hamilton makes the case for a return to a preventive approach for high-reliability software systems.

Other topics in the theme issue include

- fresh perspectives on the history and state of software engineering, by Grady Booch and Manfred Broy, expanding on and complementing similar accounts from one and two decades ago;[6,7]
- bridging the gap between research and practice, by Victor Basili, Lionel Briand, Domenico Bianculli, Shiva Nejati, Fabrizio Pastore, and Mehrdad Sabetzadeh, and by Claire Le Goues, Ciera Jaspan, Ipek Ozkaya, Mary Shaw, and Kathryn Stolee;
- the state of the practice in engineering secure software systems, by Laurie Williams, Gary McGraw, and Sammy Migues;
- software analytics' role in enhancing our understanding of software engineering, by Tim Menzies and Thomas Zimmermann;
- the mainstreaming and evolution of modern software development methods, by Rashina Hoda, Norsaremah Salleh, and John Grundy, and these methods' emerging extensions that connect the software development process with information technology governance, by Erik Dörnenburg; and
- the history and changing needs of software engineering education, by Nancy Mead, David Garlan, and Mary Shaw.

But that's not all. Our interview piece with another pioneer, Barry Boehm, will take you down memory lane for a firsthand account of lesser-known milestones and their influences. Look also for complementary contributions in *IEEE Software*'s On DevOps, Practitioners' Digest, Redirections, Reliable Code, and Software Technology departments. And, as a special treat, Željko Obrenović has paired quotes from *IEEE Software*'s early days with quotes from more recent issues to highlight how things have changed, or how they've remained the same the more they've changed.

The world of computing is changing at an accelerated pace. The relatively recent developments in data-driven computing (big data), powerful commodity platforms (the cloud), the Internet of Things, cyber-physical systems, AI and machine or deep learning, and ever-shorter feedback loops and continuous learning will likely persist for some time. Once the hype that has accompanied each of these inevitably dies down, the opportunities for real science and engineering will still remain. For example, although many successful examples of data-driven and intelligent systems exist, and despite the fact that they're predominantly software, we still don't know the best way to engineer them from first principles. Common patterns will emerge from successful examples and get codified.

As software engineers, we need to seize these opportunities and figure out the role these developments might play in advancing software engineering. This has already been happening—a look at the recent software engineering research literature will confirm as much. Conversely, we also need to figure out how software engineering can help bring out the full potential of the advancements in these areas of computing. This task is much more difficult but will pay off handsomely in the many—today perhaps even unimaginable—advances that are ahead of us.

We hope that these theme articles illuminate, encourage reflection and debate, and spawn new ideas. Enjoy!

References

1. F.P. Brooks, "No Silver Bullet: Essence and Accidents of Software Engineering," *Computer*, vol. 20, no. 4, 1987, pp. 10–19.

ABOUT THE AUTHORS

HAKAN ERDOGMUS is a teaching professor of electrical and computer engineering at Carnegie Mellon University's Silicon Valley campus. His areas of specialization include the economics of software engineering, value-based software engineering, empirical studies of software engineering practices, and software engineering education. Erdogmus received a PhD in telecommunications from INRS-Université du Québec. He's a Senior Member of IEEE, a Golden Core member of the IEEE Computer Society, and a member of ACM. Contact him at hakan.erdogmus@sv.cmu.edu.

NENAD MEDVIDOVIĆ is a professor in the University of Southern California's Computer Science Department. His research interests are in architecture-based software development. Medvidović received a PhD from the Department of Information and Computer Science at the University of California, Irvine. He's an ACM Distinguished Scientist and an IEEE Fellow. Contact him at neno@usc.edu.

FRANCES PAULISCH is responsible for Siemens Healthineers' company-wide Software Initiative. The related activities include best-practice sharing, training, and leveraging internal and external networks on software-related topics. Paulisch received a PhD in software engineering from the University of Karlsruhe. She's a member of ACM, IEEE, and Gesellschaft für Informatik. Contact her at frances.paulisch@siemens-healthineers.com.

2. D.J. Snowden and M.E. Boone, "A Leader's Framework for Decision Making," *Harvard Business Rev.*, Nov. 2007, pp. 69–76.
3. P. Naur and B. Randell, eds., *Software Engineering: Report on a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968*, NATO, 1968; http://homepages.cs.ncl.ac.uk/brian.randell/NATO/index.html.
4. F.P. Brooks, *The Mythical Man-Month*, Addison-Wesley, 1975.
5. K. Power, tweet, 31 May 2018; https://twitter.com/ken_power/status/1002132724583469056.
6. A. Brennecke and R. Keil-Slawik, eds., *Position Papers for Dagstuhl Seminar 9635 on History of Software Engineering*, 1996; https://www.dagstuhl.de/Reports/96/9635.pdf.
7. N. Wirth, "A Brief History of Software Engineering," *IEEE Annals History of Computing*, vol. 30, no. 3, 2008, pp. 32–39; https://www.inf.ethz.ch/personal/wirth/Miscellaneous/IEEE-Annals.pdf.