The Success of a Heavenly Marriage

Diomidis Spinellis

This year marks the 50th anniversary of the first conference on software engineering, which was organized in response to a perceived software crisis. It was sponsored by the NATO Science Committee and took place in Garmisch, Germany. Its chair was Friedrich Bauer; participants included Edsger Dijkstra, Gerhard Goos, David Gries, Doug McIlroy, Peter Naur, and Alan Perlis.

For a field that sprang out of a so-called software crisis, software engineering has done rather well over the past half-century. By riding on the coattails of Moore’s law, it has progressed phenomenally. The field’s achievements are visible through the large, complex, yet effective software systems that power our everyday lives. By looking at the drivers of the field’s progress and taking stock of its achievements, we can appreciate the challenges in front of us and confidently plan for the future.

Progress

Software engineering has coevolved with computer hardware and benefited mightily from its advances. It’s tempting to crow about software engineering’s astonishing progress over the past half-century. It’s also easy to look down on early programs, which offered primitive user interfaces, were written in low-level languages, lacked a modular structure, overused global variables, and utilized self-modifying code. However, those archaic programs were written in austere environments to run efficiently on puny-powered hardware.

A more reasonable view is that software engineering has successfully exploited tremendous computer hardware advances: orders-of-magnitude increases in computing power and storage capacity, standardization, sophisticated peripherals, advanced system architectures, and ubiquitous networking. By basing its evolution on that of general-purpose computing devices, software engineering advanced with the same speed as other application areas ranging from scientific and business computing to consumer electronics and gaming.

Increased computing power and storage capacities have allowed software engineers to invent, develop, and use powerful abstractions that increase software construction’s efficiency and the resulting code’s maintainability and reliability. Early programs had to use all the tricks of the trade to squeeze the most out of every one of the (few) CPU cycles and memory bits that were available at the time. As these resources became more abundant, some of them could be devoted to abstractions that resulted in better, more powerful code. Examples include dynamic memory allocation, software libraries, dynamic method dispatch, garbage collection, relational databases, regular expressions, associative arrays, scripting and domain-specific languages, generic data structures and algorithms, and virtualized containers.

In parallel, computing power’s increased availability made some of it directly available to software developers. They quickly took advantage of the offering by developing high-level languages, interactive (rather than batch-oriented) programming environments, full-screen editors, symbolic debuggers, configuration management systems, program generators, IDEs, modeling tools, development and testing frameworks, Q&A forums, and social-coding sites. These developments in turn drove software development process...
improvements, covering both tightly managed waterfall and modern agile methods.

Over the years, hardware has also become more standardized. An early zoo of incompatible word sizes, character sets, peripheral access methods, memory models, floating-point representations, and physical-network protocols has now (mostly) settled down to 8-bit bytes, Unicode, SCSI, flat memory, IEEE 754 floating-point arithmetic, and Ethernet. A stable foundation allowed the development of software standards for programming languages, APIs, data formats, and networking. All these have aided the portability of software code and the interoperability of programs. Thanks to this progress, programmers now can easily port many applications and libraries between Linux, macOS, and Windows, while users can share complex data between diverse applications.

Hardware advances have also aided software usability. There’s only so much you can do to improve the interface of a program operated through a teletypewriter. Thankfully, over the years software engineers were given addressable cursor terminals, high-resolution color graphics displays, mice and touchscreens, diverse sensors, and ubiquitous devices. The corresponding usability improvements have lowered the bar of computer use from dedicated skilled mainframe operators to toddlers pinching on tablet screens.

At a lower level, software engineers were able to exploit ever-more-sophisticated hardware architectures. Memory management hardware has allowed the development of virtual memory, OS-maintained buffer caches, and the enforcement of process separation. So, most modern programs can be designed around a high-level abstract model featuring abundant high-speed memory, fast access to reliable secondary storage, and sophisticated, secure multitasking. On the numerical-computation front, the floating-point-arithmetic standard and its high-precision formats let university graduates write accurate number-crunching code that might in the past have required a numerical analyst’s handholding.

The cherry on the top of hardware advancements that have fed into software engineering is networking. Although its initial impetus was remote access to computing facilities, software engineers quickly realized they could use networking’s abstractions and facilities to architect sophisticated planet-wide software systems. Nowadays, networking is routinely used to share computational loads between and within datacenters, to scale processing capacity, to provide fault tolerance, and to organize large systems into smaller independent modules through designs such as microarchitectures.

Achievements
What software engineering has brought through its 50 years of progress is nothing short of astounding.

First, consider modern software’s size and complexity. The software development industry regularly delivers systems comprising millions to hundreds of millions of code lines. Through aggressive layering and modularizing at the levels of the OS, middleware, and applications, well-designed systems successfully conquer complexity that dwarfs anything else humans have constructed. This feat has also been made possible through the achievement of the software reuse Holy Grail. Nowadays,
both software products and entire organizations are mostly utilizing reused software, with only a small percentage of the software being developed expressly for the particular task.

These achievements are also related to tremendous increases in the development process’s efficiency and effectiveness. Software construction tasks that were challenging at the time of the Garmisch conference are now trivial; projects that used to be impossible, such as those requiring thousands of globally distributed developers to work productively on the same code base, are now common.

Then, think of software reliability. A few decades ago it would have been politically incorrect hubris to claim reliability as an achievement of software engineering. And yet, through the extensive application of testing, through process improvements such as code reviews, continuous integration, and DevOps, and through the liberal reuse of mature components, modern software systems don’t collapse like a house of cards but mostly work correctly most of the time.

Even more impressively, software is also now frequently used in safety-critical applications—once an outlandish goal. From pacemakers, to fly-by-wire planes, to robots, to industrial automation, to driver-assisting or autonomous vehicles, we now trust software with our lives day in, day out.

Finally, consider the scope of what runs on software today—namely, everything. This is certainly a remarkable achievement for a field born out of a crisis.

Challenges
Despite the impressive progress and achievements, the road ahead for software engineering looks challenging and bumpy. There are still many problems whose solution has eluded the software engineering community.

Some aren’t technical, and perhaps this explains the lackluster progress on them. Effective collaboration with software users, for eliciting requirements or delivering software that satisfies their goals, is still an open problem, especially for large bureaucratic organizations. The challenge of attracting, training, and advancing a populous and diverse workforce remains unsolved. On a wider scale, software engineering must address its sociopathic traits (commonly defined as antisocial behavior, impaired empathy, and bold, disinhibited egotism), while dealing systematically with the ethical and social issues associated with the development of modern software-intensive systems.

On the technical front, the field must deal with its past and set a roadmap for the future. Safeguarding knowledge accumulated over decades in what is rapidly becoming legacy software is a daunting task. It challenges managers around the world, because every new line of code is one more line that must be expensive maintained against technological progress. Regarding the future, the emergence of machine learning (and deep learning in particular) based on the processing of big data is changing what it means to program. Many software engineering practices will need to adapt to this new reality.

Finally, because of energy and physical-size constraints, progress by riding on the coattails of Moore’s law will probably soon become impossible. The world where it’s reasonable for a single process’s average memory size to dwarf that of the
entire available memory in a PDP-11 Unix system is ending. The field’s scientists and practitioners will have to find ways to do more with fewer resources.

I feel extremely fortunate to have been able to participate as a practitioner and scientist in a small part of this amazing 50-year journey. I know that many in the software engineering community share this feeling. We certainly didn’t let a good crisis go to waste. We now owe future generations an even more fruitful second half-century. 😊

WELCOME NEW BOARD MEMBERS

James Herbsleb is a professor at the Institute for Software Research in the School of Computer Science at Carnegie Mellon University, where he’s the director of the PhD program in societal computing. His research interests lie primarily in the intersection of software engineering, computer-supported cooperative work, and sociotechnical systems, focusing on such areas as geographically distributed development teams and large-scale open source development. Herbsleb received a PhD in psychology and a JD from the University of Nebraska, and an MS in computer science from the University of Michigan. He has won an ACM Outstanding Research Award (2016), an Alan Newell Award for Research Excellence (2014), a Most Influential Paper Award (ICSE 2010), an Honorable Mention for Most Influential Paper Award (ICSE 2011), ACM Distinguished Paper Awards (ESEM 2008 and ICSE 2011), and Best Paper Awards (CSCW 2006 and Academy of Management 2010). Herbsleb is the editor of IEEE Software’s upcoming Engineering the Human Condition department. Contact him at jim.herbsleb@gmail.com.

Rashina Hoda is a senior lecturer (associate professor) in software engineering and the founder of the SEPTA (Software Engineering Processes Tools and Applications) research group at the University of Auckland. Her research focuses on the human and social aspects of software engineering, including agile transitions, teams, practices, self-organization, and project management in agile and lean contexts, and on human-centered design, including serious-game design, child–computer interaction, and user-centered design for smart applications. She specializes in using grounded theory and case studies to study these areas in industrial settings. Hoda received a Distinguished Paper Award at the 2017 IEEE International Conference on Software Engineering (ICSE) for her “theory of becoming agile.” She leads the game design theme of the Developing in Digital Worlds project, which is funded by the New Zealand Ministry of Business Innovation and Employment. She is on the ICSE program committee and is an associate editor for the Journal of Systems and Software and the International Conference on Information Systems. She served as the research chair of the Agile India 2012 conference, has co-organized several CHASE (Cooperative and Human Aspects of Software Engineering) workshops at ICSE, and chaired the research workshops and impact-to-industry track at XP 2018 and EASE (Evaluation and Assessment in Software Engineering) 2018. Rashina received a PhD in computer science from Victoria University of Wellington. Contact her at r.hoda@auckland.ac.nz; www.rashina.com.