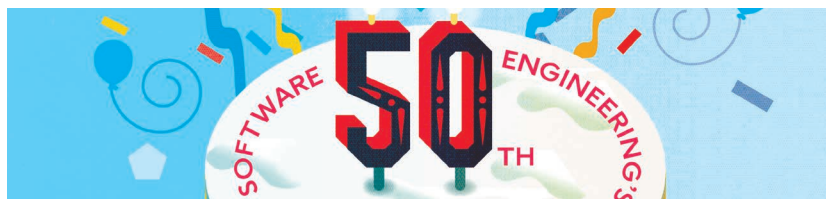


Yesterday, Today, and Tomorrow

50 Years of Software Engineering

Manfred Broy, Technische Universität München and Zentrum Digitalisierung.Bayern

// It's now 50 years after the groundbreaking conference that introduced the notion and discipline of software engineering. This is a moment to look at what we've achieved, what we haven't achieved, where we are today, and what challenges lie ahead. //



IN OCTOBER 1968, the NATO Science Committee sponsored a meeting in Garmisch, Germany, chaired by Friedrich Bauer, to respond to what was called the “software crisis.” The result of the meeting was documented by a report concerned with “a problem crucial to the use of computers, viz. the so-called software, or programs, developed to control

their action.”¹ The conference was attended by more than 50 people from 11 countries, all concerned professionally with software, either as users, manufacturers, or teachers and researchers at universities.

The discussions covered all aspects of software as seen at that time, including the relation of software to computer hardware, the

design of software, the production and implementation of software, the distribution of software, and the servicing of software. This meeting was very influential, covering topics that paved the way for research in the next 50 and maybe more years. (For photos from the conference, see <http://homepages.cs.ncl.ac.uk/brian.randell/NATO/N1968/index.html>.)

Yesterday

Looking at the list of topics discussed 50 years ago, a lot seems remarkably familiar. In particular, two fabulous working papers by Doug McIlroy on mass-produced software components² and by Edsger Dijkstra on the complexity controlled by hierarchical ordering of functional variability³ point to still-active areas of research. Brian Randell contributed the working paper “Towards a Methodology of Computing System Design,”⁴ and a lot of what he wrote there still seems highly relevant to us today.

However, the major contribution of the Garmisch conference was not only what was presented but also what was initiated. People became more aware of the challenges of software engineering, and they agreed that software engineering is and should be an engineering discipline. One of the characteristics of an engineering discipline is a scientific foundation—methods and techniques that are based on scientific theories to quantify the correctness, effectiveness, and validity of engineering methods.

A lot of the work that started, motivated by the conference, was creating step by step scientific foundations for software engineering. For instance, the most remarkable step into structured programming was made, leading more



or less finally into the assertion calculus of Tony Hoare⁵ and Edsger W. Dijkstra⁶ and opening the stage for a number of fundamental papers explaining some of the foundations of our field. Moreover, questions of the software development process were investigated, as were methods of software development management. Soon, a lot of different scientific methods were applied in software engineering, working out theories, statistical methods, analytical methods, and much more.

What is interesting, however, is what was not discussed at the 1968 meeting. Because of the speedy development of the technology, quite a number of challenges became visible only after the Garmisch conference. One example was embedded systems, which started only around 1968 with microprocessors that included embedded software. Another example was distributed systems, networks, and concurrency—and, in turn, cybersecurity topics that are of still-growing interest today.

The major topic of programming languages, although not a focus of the Garmisch conference, got a lot of attention in the '60s and beyond. Niklaus Wirth was working toward Pascal, a milestone in programming languages. More or less at the same time, Simula 67 was started, leading into object-oriented programming. However, all object-oriented programming languages that have been proposed up to today follow more or less the classic von Neumann-style sequential-execution model. All attempts to widen the object-oriented paradigm toward concurrency, interaction, and real time so far have led only to complicated concepts too difficult to understand and to apply.

Actually, the Garmisch conference showed just how many topics

in software development were still open issues—challenges with which we still deal today, such as

- getting requirements right,
- designing adequate architectures,
- doing implementation effectively and correctly,
- verifying the quality of the result, and
- maintaining software systems with targeted functionality and high code quality over long time periods and to further develop them.

We have seen enormous progress in our field, with a lot of work on requirements engineering and on data structures, and with the rise of databases, networks, and distributed systems all influencing software engineering. We have seen a number of schools and lines of thought in software engineering, such as

- structured programming;
- formal methods, with the idea that software engineering should be based completely on a mathematical calculus;
- model-oriented software development; and
- code-centric agile programming, which puts all the responsibility onto the people who do the job and places trust in their motivation and competency—in particular, in the application domain.

All schools have contributed, but none has delivered the silver bullet.

Even for a software engineer, it is amazing to see how our field has developed over the past 50 years. My PhD supervisor, predecessor, and paternal scientific friend, Friedrich Bauer, told me that he never expected

such a significance and range of application of software engineering. When dealing with the first programs, when writing the first software in the '50s and even in the '60s, no one could have expected the role software plays today. Cyber-physical systems are everywhere—and are operated by everyone.

This brings in challenges to software systems that we did not face before. Software has become part of physical reality; it changes and enhances physical reality. We have to keep in mind that in 1968, most of the software was running on mainframes, not on a network and not directly connected to the processes of the real world, and was therefore quite separated from those. Nowadays, software is tightly coupled with physical processes and controls a lot of the technical and social activities of the world.

Today

Our discipline has exploded since 1968. In 1968, there were a number of key people who seemed to understand the whole discipline in all its facets. Since then, our field has come up with so many different subdisciplines and areas of application that no one can grasp the whole field in all its details any more. At the same time, it is becoming pervasive and mission critical, developing with a speed that is breathtaking.

Perhaps most remarkable was the enormous momentum with which networks developed. With the start of the Internet at DARPA, all computers became connected sooner or later, and all of the software was running on connected distributed hardware. Computers developed from mainframes to workstations, desktops, laptops, tablets, and smartphones. Client-server hardware and

software architecture became quasi standard and developed into general networked computer systems with distributed software. Finally, computers are everywhere; software is used by everyone and is connected to everything, which has an enormous impact on software engineering. A contemporaneous development has been that of embedded software, finally leading to cyber-physical systems—with grand challenges today like autonomous driving.

On the basis of the invention of the web, search engines, and connected devices, the whole discipline is entering a new era. With the arrival of the smartphone, computing devices are now practically in the hands of everyone, every day, all the time. Software has to serve billions of devices and billions of users with millions of applications, placing high demands on software developers. Software engineering has grown from a small discipline in 1968, with a restricted number of experts, into a mass industry. That industry, due to mechanisms like “business of scale” and “winner takes all,” is becoming more and more the predominant technical discipline today.

Today, software is eating the world.⁷ There are literally no technical or organizational processes and no services not related to software systems. The communication and lifestyle of people depend to a large extent on software systems. The security and critical infrastructure of whole nations depend on software. The largest and most successful companies around the world are software companies.

Through social networks, big data, and machine learning, new applications are created based not only on the enormous mass of data from embedded systems but also on user

behavior on the Internet, exploiting increases in machine power in accordance with Moore’s law.

Particularly remarkable is the success of AI. The past 50 years saw several trends that made big promises and set high expectations but ended in disappointment, often so deep that progress in AI was nearly overlooked. Today, AI systems offer functionality we hardly expected. Due to the enormous power of the hardware and big data available today, learning algorithms are being used successfully for specific applications. Natural-language processing is only one example. AI methods now have to be considered a branch of software engineering. This brings new challenges for software engineers to develop hybrid systems with both conventional and learning-based software parts and to specify and verify these systems.

Best Practice versus Everyday Practice

However, in the everyday state of the practice of software engineering, there is still a crucial discrepancy between the enormous success of software used everywhere and the much-too-weak abilities of many—even large—companies around the world to produce high-quality software. It is depressing to see companies that do not apply the best-understood and most effective techniques and methods in software engineering for capturing their requirements, for designing their architecture, and for adequate coding and quality control, including continuous integration and systematic automatic testing for achieving and keeping high code quality. A major and ever-growing problem still is legacy software, software that is badly needed and used every day but

is of low quality, hard to maintain, and hopeless with respect to incorporating additional functionality.

Improving Our Abilities

It is amazing that important areas of software engineering have not developed further, as is necessary considering the quick evolution of technology and applications. One example is programming languages that do not reflect the needs of today’s programming of cyber-physical systems. Our programming languages are no good for handling real-time interaction and distribution. Coding is error prone, not sufficiently abstract, and still done too close to the machine level or at least to the OS level, mixing application logic with low-level technical systems.

Our software tools do not reflect all the possibilities for improving and supporting software development processes. We are a discipline where the power of the technology and the demands of applications are growing faster than our ability to maintain this pace using our methods. The incredible economic success of software makes it even harder to convince people to base their approach on scientific foundations—agility, as well as speed and application and business success, seems all that counts. Finally, established companies, which have to find their way into the new areas of software, are run by management that often has no clue on how to organize software development effectively.

Getting Requirements Right

One of the perhaps most difficult challenges of building software systems is to understand the requirements.⁸ Even today, we see many examples of software that is not

properly specified, does not meet the users' expectations, is not correct and reliable due to insufficient requirements, and finally fails to meet the quality requirements owing to missing specifications. Agile methods can help only to a certain extent because it depends very much on the product owner whether expectations are met.

Apart from that, the existing standards for requirements engineering are out of date and do not address the current situation of interactive real-time systems related to the real world, with all the requirements dealing with probabilities and all kinds of quality demands for system functionality. Approaches like design thinking, with their prototyping ideas, might help to clarify requirements. However, a prototype is not a requirements specification at all. A second step is needed to identify which properties of the prototype are definitely requirements and which aspects are not.

Software Development—Agile or Not Agile

Software engineering has always suffered from trends. Each decade over the past 50 years has had its own trends, starting with structured programming, then client-server architectures, then object-oriented programming, and now web programming and web services, which are very much influenced by cloud technologies, software platforms, and agile methods. But as always, the truth is never in just one method. The old wisdom of Fred Brooks, "there is no silver bullet," proves still valid over and over again.⁸

In fact, agile development⁹ is also not the silver bullet. Its advantage is fast feedback to the developer. We have to work hard to make sure

that agile programming and continuous integration find a synthesis with model-oriented techniques and systematic approaches¹⁰ for documentation, systematic analysis, and long-term development.


The enormous economic relevance of software must be reflected in the way we understand the role of software engineers. They have to become entrepreneurs and leaders defining the software strategies of companies and organizations. This has to be reflected in software engineering education.¹¹

Tomorrow

How will software engineering look tomorrow? Software engineering will be part of systems engineering

programming languages and in architectural ideas based on them. We need to define architectures in terms of interfaces including all kinds of real-world aspects such as time, concurrency, topology, place, probability, and much more.

Another big challenge is maintenance and resilience. We have to understand how to design and implement reliable software such that maintenance, changeability, and variability are easier to handle. A key is to work on software that is adaptive and to a large extent autonomous, prepared for all the different use cases and their special demands that may come up. Changing business models with an emphasis on a sharing economy and service orientation ask for concepts



Our software tools do not reflect all the possibilities for improving and supporting software development processes.

of cyber-physical systems closely connected to the real world. Programming has to be much closer to the real world; this is particularly true when defining requirements. Getting the requirements right, making systems attractive to the users, and making systems right and easy to use are still ongoing challenges.

A further challenge is architecture. We need more abstract views into architecture. We have to make sure that architectures provide flexibility and methodological structure. We have to go away from the old von Neumann-style architectural ideas as we find them in today's

supported by software. Technical challenges such as autonomous driving pose questions related to societal, ethical, and legal issues.

Tooling

In software engineering, we have worked on building tools for more than 40 years now. However, the tools we have are still insufficient. Tools based on UML and SysML are of interest to some practitioners because they believe that graphical representations seem easier to understand. However, we need an expressive power in tools that goes far beyond simple diagrams. We need

comprehensive artifact models that capture all the documentation produced in the development process and that show the proper relationships of that documentation—not just for tracing but also for being able to exploit data already captured for the next step in the development process.

Teaching

Our attitude toward teaching software engineering must also change. The demand exists for more than just the classic teaching of the basics of software engineering. Today, for a growing number of companies, software is of high strategic relevance. Therefore, they need software engineers who, on

as AI, blockchains, and big data are still to come, and developments will occur in other areas. Today we are virtually cyborgs; perhaps tomorrow we will literally be cyborgs. Software systems will be part of everything, including human bodies, not only through DNA programming. Software support will be everywhere.

Software engineering has to incorporate methods from cloud technology, microservices, and AI, building software systems comprising both conventional parts and machine-learning parts. Software systems have to be further developed while they are running, and they have to be adaptive and interoperable to a much larger extent than we can imagine

most significant economic impact in engineering.

There are many techniques and methods that are relevant for software engineers, such as databases, cloud technologies, distributed systems, and AI. In addition, software engineers need to understand application-specific know-how from fields like medicine, avionics, and the automotive domain, when they're working in the those application areas. Our field is structured into core areas such as requirements engineering, architecture design, coding, and quality assurance.

Our discipline still is not and, due to the speed of technological development, cannot be in a mature state. A lot of software is full of faults, does not sufficiently address the need of the users, and shows too many difficulties and obstacles in use.

At the same time, software has deeply changed the world. The safety developed for air traffic, with its balance between the pilot's interaction through carefully designed human-machine interfaces and powerful support from embedded software and flight control systems, is to a large extent due to software. The effectiveness of our software-based processes is amazing. What we see going on in the Internet—be it search engines, social networks, or online shops—is all about to change our world.

All of these fields of application are using software and depend on professional software engineering. It is key to have software that is reliable, addresses the user's needs, and is future-proof. The quest goes on. Software and systems engineers will continue to change the world. Only if we manage to develop our methodologies as quickly as the field develops its application needs will we have the chance to keep the world from

It is key to have software that is reliable, addresses the user's needs, and is future-proof.

one hand, understand software engineering methods and technologies and, on the other hand, are competent making decisions at the strategic level. Only this way will we be sure that companies run with the right software technology for their demands, produce software-based products and processes that are the right ones for the future, and are able to stay in the market and grow on the basis of their software know-how. Typical examples are the platform-based companies of today and the data-driven companies of today and tomorrow.

Additional Developments

It is also clear that additional decades of development in areas such

today. This requires a systematic approach to software and systems engineering based on a conceptual backbone in which all the artifacts in software engineering are integrated and can be related, adapted, modified, and improved.

Looking back at the past 50 years and looking forward at what is to come, we can only be amazed at the speed of our field's progress. Today, software engineering is the most important engineering technique. It is about to become the key interdisciplinary approach for all the established engineering techniques, and it has the

running into the dead end of software that's much too complicated, much too difficult to maintain, lacking cybersecurity, and much too unreliable.

Having seen 50 years of software engineering with a clear look at where we are today, it is quite obvious that it is completely impossible to predict the future of the field in all its details. Software engineering is the future. It is very hard for the software engineering community to reach all those individuals who would benefit from knowing our basic techniques. This means we have to teach the world. Software engineering will influence and change every application area. It will be part of the curriculum of every other scientific discipline. That will require scientific understanding more than ever. 🌐

Acknowledgments

This article is dedicated to Friedrich L. Bauer, my predecessor in the chair for software and system engineering at Technische Universität München. It's a pleasure to thank Frances Paulish for stimulating discussions.

References

1. P. Naur and B. Randell, eds., "Highlights," *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>.
2. D. McIlroy, "'Mass Produced' Software Components," *Software Engineering: Report on a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968*, P. Naur and B. Randell, eds., NATO, 1968, pp. 138–156; <http://homepages.cs.ncl.ac.uk/brian.randell/NATO/index.html>.
3. E.W. Dijkstra, "Complexity Controlled by Hierarchical Ordering of Function and Variability," *Software Engineering: Report on a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968*, P. Naur and B. Randell, eds., NATO, 1968; <http://homepages.cs.ncl.ac.uk/brian.randell/NATO/index.html>.
4. B. Randell, "Towards a Methodology of Computing System Design," *Software Engineering: Report on a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968*, P. Naur and B. Randell, eds., NATO, 1968, pp. 204–208; <http://homepages.cs.ncl.ac.uk/brian.randell/NATO/index.html>.
5. C.A.R. Hoare, "An Axiomatic Basis for Computer Programming," *Comm. ACM*, vol. 12, no. 10, 1969, pp. 576–580.
6. E.W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, 1976.
7. M. Andreessen, "Why Software Is Eating the World," *Wall Street J.*, 20 Aug. 2011; <https://www.wsj.com/articles/SB10001424053111903480904576512250915629460>.
8. F.P. Brooks, *The Mythical Man Month*, Addison-Wesley, 1995.
9. K. Beck et al., "Manifesto for Agile Software Development," Agile

ABOUT THE AUTHOR



MANFRED BROY has been a professor at the Institute for Informatics at Technische Universität München and is the founding president of Zentrum Digitalisierung.Bayern. His research aims at the foundation of software and systems engineering, with emphasis on improving the engineering of software in cyber-physical systems. Broy received a habilitation from the Faculty for Mathematics and Informatics at Technische Universität München. Contact him at broy@in.tum.de.

- Alliance, 2001; <http://agilemanifesto.org>.
10. CMMI Product Team, *CMMI for Software Engineering, Version 1.1, Staged Representation (CMMI-SW, V1.1, Staged)*, tech. report CMU/SEI-2002-TR-029, Software Eng. Inst., Carnegie Mellon Univ., 2002.
11. M. Broy, W. Brenner, and M. Leimeister, "Auf dem Weg zu einer Informatik neuer Prägung in Wissenschaft, Studium und Wirtschaft" [On the Way to a Computer Science of New Character in Science, Study, and Economics], *Informatik-Spektrum*, vol. 40, no. 6, 2017, pp. 1–5 (in German).

myCS

Read your subscriptions through the myCS publications portal at

<http://mycs.computer.org>