Quotes from *IEEE Software* History

Željko Obrenović

JUST LIKE SOFTWARE engineering, *IEEE Software* has a rich history. Since 1984, many leading software engineering professionals have contributed ideas and lessons they've learned to the magazine.

In my role as an informal curator of the *IEEE Software* history website (https://obren.info/ieeesw),[1] I've read quite a few of the early *IEEE Software* articles. Although many of these contributions are now obsolete, I was surprised to find out how much of the early work is still valid.

To call attention to the relevance of such often-forgotten articles, I created an alternative view of *IEEE Software* history, extracting quotes organized in "conversations." Each conversation pairs a quote from the magazine's early days (1984–1990) with a more contemporary quote, with at least 20 years between the two. In this way, I hope to illustrate that some key ideas and topics are classic and have value even decades later.

My selection of quotes isn't an attempt to create a static, systematic overview of all software engineering trends. It only scratches the surface. The main goal is to create an interesting, inspirational presentation of software engineering history, at least as captured by *IEEE Software*. I hope to pique your curiosity so that you study this history and engage in such conversations with it yourself.

So, why are many of the old software engineering articles still important? Figure 1 shows the progress of two sides of software engineering: technological and human. On the one hand, computing technology has been progressing in a superlinear fashion for years. And software engineering has been closely related to this trend. Moreover, software has been a main driver behind most of the recent technological advances.

For instance, over the past 10 years, *IEEE Software* has covered mobile computing, cloud computing, big data and analytics, automotive software, the Internet of Things, social media and crowdsourcing, cyber-physical systems, and bitcoins and cryptocurrency. These are largely new phenomena whose size, complexity, and novelty have no direct parallels with the early years of

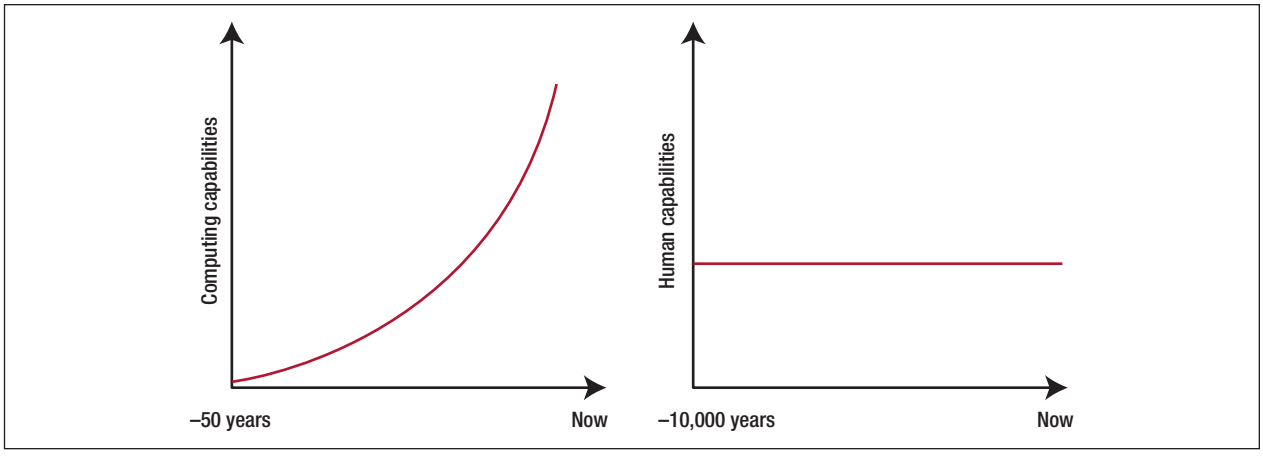

FIGURE 1. Two sides of software engineering: technological and human. Software engineering has progressed quickly, but human nature and behavior haven't. That's why old software engineering articles are still relevant.

 0740-7459/18/$33.00 © 2018 IEEE

software engineering and *IEEE Software*. Lessons learned about some technology trend 20 years ago (or in some cases only a few years ago) tend to have limited value today. Although such technology-centric contributions are highly relevant at the moment of their publication, they're normally only a stepping stone in the development of technology, with little value for the next technology generation.

And then there's the human side. Human nature and cognitive capabilities haven't advanced with technology. That's the main reason why old software engineering contributions are still important. Software engineering is more about humans than about computers. It's concerned primarily with techniques that help people deal with complexity, ambiguity, and each other as they build complex software systems. Or, as James Coplien so nicely expressed, the core principles of software architecture, such as coupling and cohesion, aren't about the code.[2] The code doesn't "care" about how cohesive or decoupled it is. But people do care about their coupling to other team members. And about these and many other human issues, we can still learn much from our past. The challenge is to extract and keep these lessons.

For a selection of quotes, see the sidebar. For the complete collection, see the Web Extra at https://extras.computer.org/extra/mso2018050010s1.pdf.

ABOUT THE AUTHOR

ŽELJKO OBRENOVIĆ is a consultant at the Software Improvement Group and is on *IEEE Software*'s advisory board. Contact him at z.obrenovic@sig.eu.

References

1. Z. Obrenović, "Insights from the Past: The *IEEE Software* History Experiment," *IEEE Software*, vol. 34, no. 4, 2017, pp. 71–78.
2. J.O. Coplien, "Reevaluating the Architectural Metaphor: Toward Piecemeal Growth," *IEEE Software*, vol. 16, no. 5, 1999, pp. 40–44.

SAMPLE QUOTES

1984	2009
Many of the challenges facing the software industry today are a direct result of our insatiable appetite for new computer-based systems applications. Others confront us simply because we have not managed to successfully solve a large number of problems that we ourselves created many years ago.	Our aspirations grow faster than our capabilities, so I don't expect software development to "get solved."
B.D. Shriver, "From the Editor-in-Chief," *IEEE Software*, vol. 1, no 1, pp. 4–5.	M. Shaw, "Continuing Prospects for an Engineering Discipline of Software," *IEEE Software*, vol. 26, no. 6, 2009, pp. 64–67.

1984	2009
I believe that in our branch of engineering, above all others, the academic ideals of rigor and elegance will pay the highest dividends in practical terms of reducing costs, increasing performance, and in directing the great sources of computational power on the surface of a silicon chip to the use and convenience of man.	It's possible to combine rigor and relevance in computing research in a fairly simple manner. Will (at least some) journals require researchers to pursue this approach? Will researchers begin to employ it? Will practitioners, once relevant work starts pouring forth from research journals, pay attention? Our field's future relevance is at stake. That communication chasm that has for so long separated our research and practice communities might at last begin to go away.
C.A.R. Hoare, "Programming: Sorcery or Science?," *IEEE Software*, vol. 1, no. 2, 1984, pp. 5–16.	R.L. Glass, "Making Research More Relevant While Not Diminishing Its Rigor," *IEEE Software*, vol. 26, no. 2, 2009, pp. 96, 95.

1984	2016
Periods of rapid technological change require more innovation and greater risks than periods of stability.	The fast-changing nature of our field is one of the things that make working in software so much fun—and so challenging.
P. Wegner, "Capital-Intensive Software Technology," *IEEE Software*, vol. 1, no. 3, 1984, pp. 7–45.	M. Vierhauser, R. Rabiser, and P. Granbacher, "Monitoring Requirements in Systems of Systems," *IEEE Software*, vol. 33, no. 5, 2016, pp. 22–24.

1985	2011
The use of formal notation does not, however, preclude that of natural language. In fact, mathematical specification of a problem usually leads to a better natural-language description. This is because formal notations naturally lead the specifier to raise some questions that might have remained unasked, and thus unanswered, in an informal approach.	Research has shown that formal specifications and methods help improve the clarity and precision of requirements specifications.
B. Meyer, "On Formalism in Specifications," *IEEE Software*, vol. 2, no. 1, 1985, pp. 6–26.	D. Drusinsky et al., "Verification and Validation for Trustworthy Software Systems," *IEEE Software*, vol. 28, no. 6, 2011, pp. 86–92.

SAMPLE QUOTES (cont.)

1984	2008
An abstraction is a simplified description, or specification, of a system that emphasizes some of the system's details or properties while suppressing others. A good abstraction is one that emphasizes details that are significant to the reader or user and suppresses details that are, at least for the moment, immaterial or diversionary.	Determining the appropriate level of abstraction is an old debate in the patterns community—authors are always asking, "Where should abstraction end?"
M. Shaw, "Abstraction Techniques in Modern Programming Languages," *IEEE Software*, vol. 1, no. 4, 1984, pp. 10–26.	L. Rising, "Understanding the Power of Abstraction in Patterns," *IEEE Software*, vol. 24, no. 4, 2007, pp. 46–51.

1985	2014
The lack of a complete theoretical basis for distributed computing systems need not inhibit the development of useful systems. Even without such a basis, many technical advances have been made by individuals, who then share them with others, who in turn accept useful concepts and add further innovations.	The capacity to reflect on past practice is important for continuous learning in software development. Reflection often takes place in cycles of experience followed by conscious application of learning from that experience, during which a software developer might explore comparisons, ponder alternatives, take diverse perspectives, and draw inferences, especially in new and/or complex situations.
S.F. Lundstrom and D.H. Lawrie, "Experiences with Distributed Systems," *IEEE Software*, vol. 2, no. 3, 1985, pp. 5–6.	T. Dybå, N. Maiden, and R.L. Glass. "The Reflective Software Engineer: Reflective Practice," *IEEE Software*, vol. 31, no. 4, 2014, pp. 32–36.

1985	2017
Today we tend to go on for years, with tremendous effort to find that the system, which was not well understood to start with, does not work as anticipated. We build systems like the Wright brothers built airplanes—build the whole thing, push it off the cliff, let it crash, and start over again.	39 percent even used the production system as a testing environment
W.E. Howden, "The Theory and Practice of Foundation Testing," *IEEE Software*, vol. 2, no. 5, 1985, pp. 6–17.	M. Kassab, J.F. DeFranco, and P.A. Laplante, "Software Testing: The State of the Practice," *IEEE Software*, vol. 34, no. 5, 2017, pp. 46–52.

1986	2016
One of the major challenges facing project software system managers and maintainers in the 1980's is how to upgrade large, complex, embedded systems, written a decade or more ago in unstructured languages according to designs that make modification difficult.	It's also important to understand the difference between what a single programmer can do and what large teams of programmers can do. Even the best practices of refactoring are really a joke in the context of a large legacy application. Refactoring tools really don't help you with large legacies.
R.N. Britcher and J.J. Craig, "Using Modem Design Practices to Upgrade Aging Software Systems," *IEEE Software*, vol. 3, no. 3, 1986, pp. 16–24.	D. Thomas quoted in S. Johann, "Dave Thomas on Innovating Legacy Systems," *IEEE Software*, vol. 33, no. 2, 2016, pp. 105–108.