Conversations with the Past:

50 Quotes from *IEEE Software* History

ŽELJKO OBRENOVIĆ

Just like Software Engineering, *IEEE Software* has a very rich history. Since 1984, many of the leading software engineering professionals have contributed ideas and lessons they learned to *IEEE Software*.

In my role as an informal "curator" of *IEEE Software* history site [1], I read for the first time many of the largely forgotten early *IEEE Software* articles. While lots of these contributions are obsolete nowadays, I was surprised to find out how much of early work is still actual.

To call attention to relevance of such often forgotten software engineering articles, I created an alternative view on the *IEEE Software* history, extracting 50 quotes organized in 25 "conversations". Each conversation features two quotes, one from the early days of *IEEE Software* (1984–1990), and another one more contemporary, with the threshold of at least 20 years in between. In this way, I want to illustrate that some key ideas and topics are "classical" and have value even decades later.

My selection of quotes is not an attempt to create a systematic overview of all trends in software engineering. It only scratches the surface. The main goal of this conversations is to create an interesting and inspirational presentation of software engineering history, at least as captured by *IEEE Software*. I want to trigger the curiosity of the reader to study and engage in such "conversations" with our history themselves. In other words, instead of presenting a static view on trends, I wanted to create a more dynamic medium, hopefully stimulating readers to read again old software engineering contributions.

Before presenting the quotes, I would also like to briefly reflect on my view on why many of the "old" software engineering articles are important today. Figure 1 illustrates my view on this issue, showing the progress of two sides of software engineering: technological and human. On the one hand, computing technology has been progressing in a superliner 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. In past ten years, for instance, *IEEE Software* has covered topics including mobile computing, cloud computing, big data and analytics, automotive software, internet-of-things, social media and crowd sourcing, cyber-physical systems, bitcoin and cryptocurrency. These are largely new phenomena that in their size, complexity and novelty do not have direct parallels with early years of software engineering and *IEEE Software*. Lessons learned about some technology trend 20 years (or in some cases only few years ago), tend to have limited value today. While such technology-centric contributions are highly relevant at the moment of their publication, they are normally only a stepping stone in development of technology, with little value for the next technology generation.

Software Engineering: Computing and Human Sides

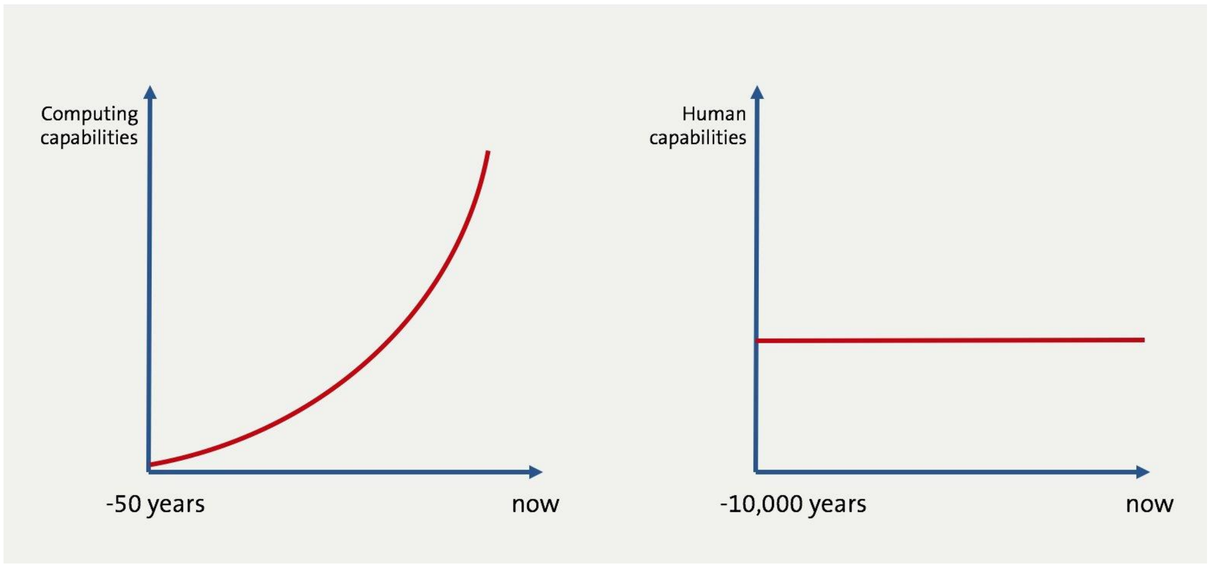


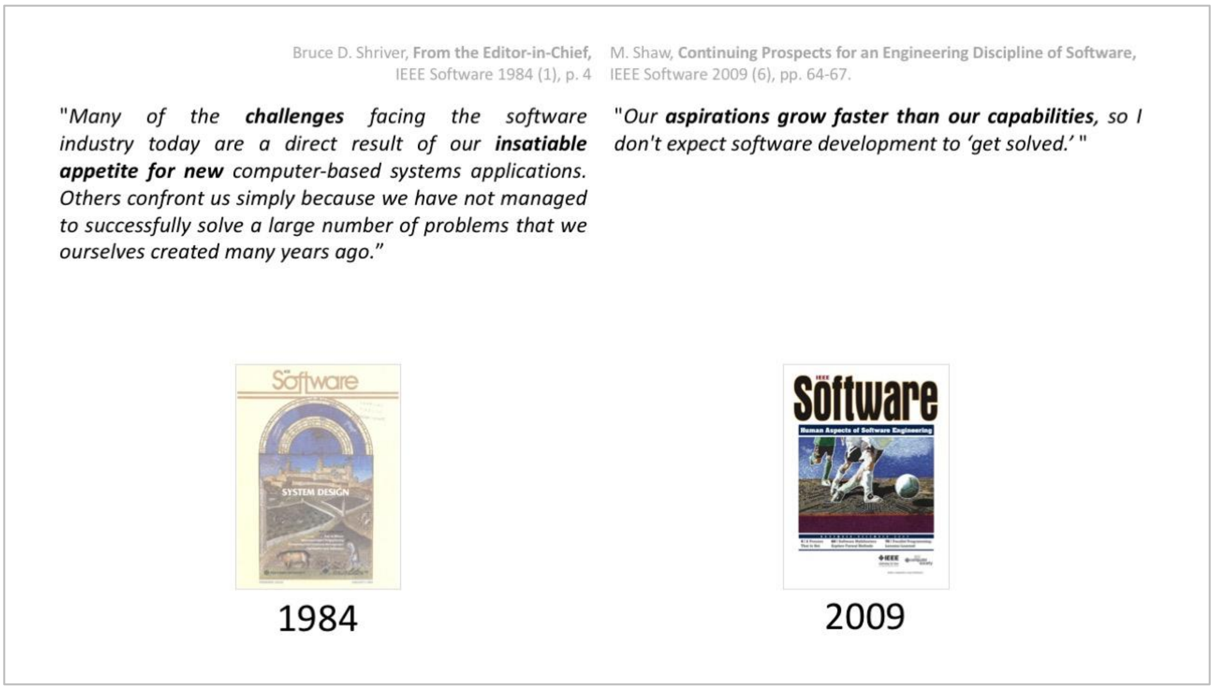
Figure 1: *Two sides of software engineering: technological and human. Software engineering is more about people than about computers. A brief remainder on why "old" software engineering articles are still relevant.*

And then there is the human side. Human nature and cognitive capabilities have not advanced with technology. That is, in my view, the main reason why old software engineering contributions are still important. Software engineering is more about humans than about computers. It is primarily concerned with techniques that help people to deal with complexity, ambiguity, and each other as they build complex software systems. Or, as nicely expressed by James Coplien [2], the core principles of software architecture, such as coupling and cohesion, aren't about the code. 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 a lot from our past. The challenge is to extract and keep these lessons.

References

1. Z. Obrenovic, "Insights from the Past: The *IEEE Software* History Experiment," in *IEEE Software*, vol. 34, no. 4, pp. 71-78, 2017.
2. J. O. Coplien, "Guest Editor's Introduction: Reevaluating the Architectural Metaphor-Toward Piecemeal Growth," in *IEEE Software*, vol. 16, no. , pp. 40-44, 1999.

25 Conversations with the Past

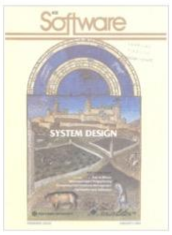


Bruce D. Shriver, **From the Editor-in-Chief,** IEEE Software 1984 (1), p. 4

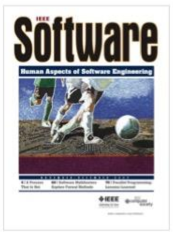
"*Many of the* ***challenges*** *facing the software industry today are a direct result of our* ***insatiable appetite for new*** *computer-based systems applications. Others confront us simply because we have not managed to successfully solve a large number of problems that we ourselves created many years ago.*"

M. Shaw, **Continuing Prospects for an Engineering Discipline of Software,** IEEE Software 2009 (6), pp. 64-67.

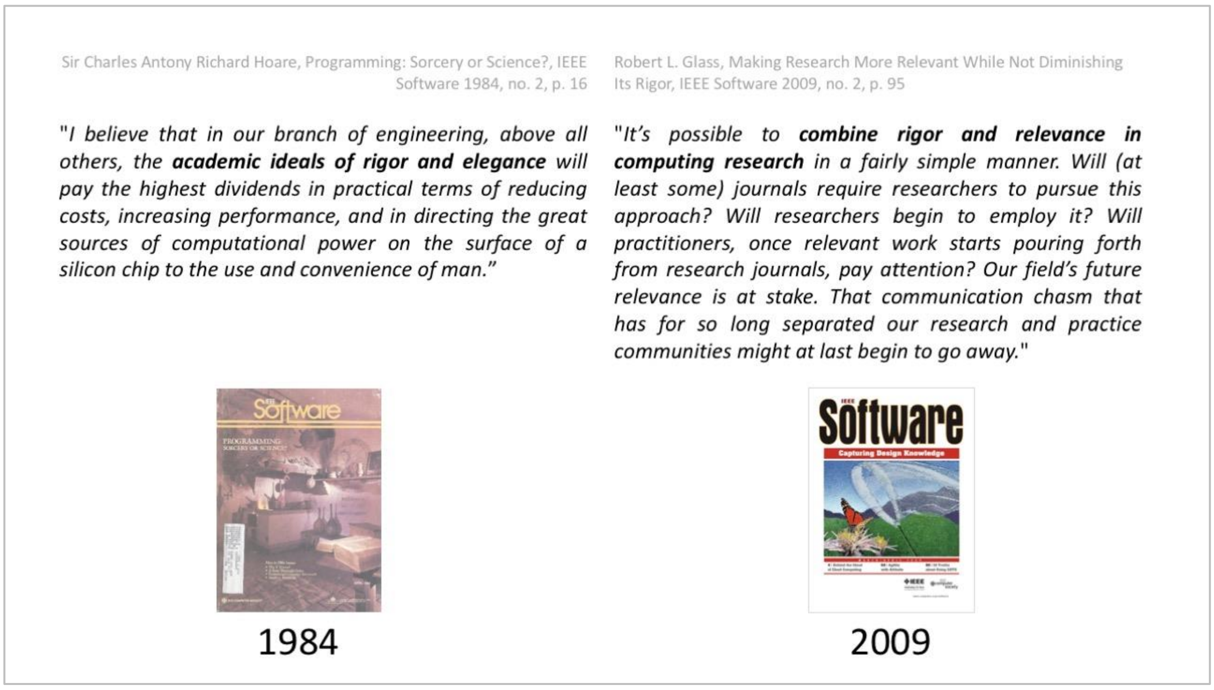
"*Our* ***aspirations grow faster than our capabilities****, so I don't expect software development to 'get solved.'* "



1984



2009



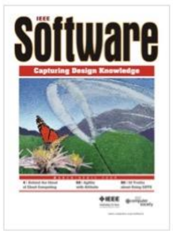
Sir Charles Antony Richard Hoare, Programming: Sorcery or Science?, IEEE Software 1984, no. 2, p. 16

"*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.*"

Robert L. Glass, Making Research More Relevant While Not Diminishing Its Rigor, IEEE Software 2009, no. 2, p. 95

"*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.*"

1984



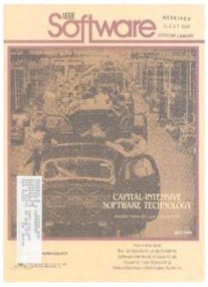
2009

Peter Wegner, Capital-Intensive Software Technology Conclusion, IEEE Software 1984, no. 3, p. 43

"*Periods of **rapid** technological **change** require more **innovation** and **greater risks** than periods of stability.*"

M. Vierhauser, R. Rabiser, P. Granbacher, "Monitoring Requirements in Systems of Systems", IEEE Software 2016 (issue 5), p. 22

"*The **fast-changing nature of our field** is one of the things that make working in software so much fun—and so challenging.*"



1984

2016

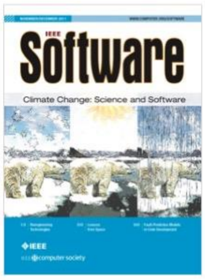
Bertrand Meyer, On Formalism in Specifications, IEEE Software 1985, no. 1, p. 22

"*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.*"

D. Drusinsky, J. B. Michael, T. W. Otani and M. Shing, "Verification and Validation for Trustworthy Software Systems," in IEEE Software, vol. 28, no. , pp. 86-92, 2011.

"*Research has shown that **formal specifications** and methods help **improve the clarity and precision** of **requirements** specifications.*"

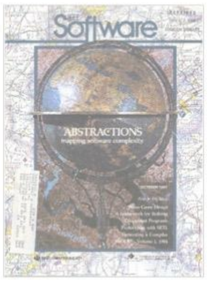
1985



2011

Mary Shaw, Abstraction Techniques in Modern Programming Languages, IEEE Software 1984, no. 4, p. 10

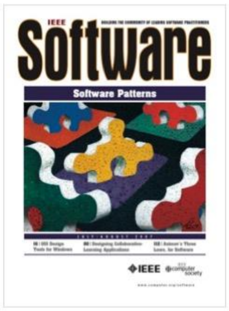
"*An **abstraction** is a simplified description, or specification, of a system that **emphasizes some** of the system's details or properties while **suppressing others**. A good abstraction is one that emphasizes **details that are significant** to the reader or user and suppresses details that are, at least for the moment immaterial or diversionary.*"



1984

L. Rising, "Understanding the Power of Abstraction in Patterns," in IEEE Software, vol. 24, no. , pp. 46-51, 2007.

"*Determining the appropriate **level of abstraction** is an old debate in the patterns community—authors are always asking, '**Where should abstraction end**?'*"



2007

Stephen F. Lundstrom, Duncan H. Lawrie, **Experiences** with Distributed Systems, IEEE Software 1985, no. 3, p. 5

"*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.*"

1985

The Reflective Software Engineer: Reflective Practice (FREE) T. Dybea, N. Maiden, R. L. Glass - IEEE Software 2014 (issue 4), p. 32

"*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.* "

2014

William E. Howden, The Theory and Practice of Foundation Testing, IEEE Software 1985, no. 5, p. 8

"*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**.*" *

1985

** NOTE: This is not an accurate description of how the Wright brothers worked. They have an advanced testing approach, using wind tunnels tests extensively.*

M. Kassab, J. F. DeFranco, P. A. Laplante, Software Testing: The State of the Practice - IEEE Software 2017 (issue 5), p. 46

"39 percent even used the production system as a testing environment"

2017

Robert N. Britcher, James J. Craig, Using Modem Design Practices to Upgrade Aging Software Systems, IEEE Software 1986, no. 3, p. 16

"*One of the major challenges facing project software system managers and maintainers in the 1980's is how to **upgrade large**, **complex**, embedded system, written a decade or more ago in unstructured languages according to **designs that make modification difficult**.*""

1986

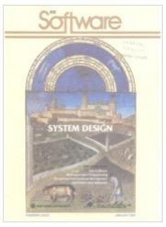
Dave Thomas on Innovating Legacy Systems (FREE) S. Johann - IEEE Software 2016 (issue 2), p. 105

"*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.*"

2016

Butler W. Lampson, Hints for Computer System Design, IEEE Software 1984, no. 1, p. 11

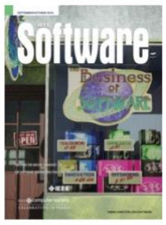
"***Designing a computer system*** *is very different from designing an algorithm: ... the requirement - is* ***less precisely*** *defined* ***more complex,*** *and* ***more subject to change****; the system has much more internal structure - hence, many internal interfaces; and the measure of success is much less clear. The designer usually finds himself floundering in a sea of possibilities, unclear about how one choice will limit his freedom to take other choices or affect the size and performance of the entire-system."*



1984

M. Vierhauser, R. Rabiser, P. Granbacher, "Monitoring Requirements in Systems of Systems", IEEE Software 2016 (issue 5), p. 22

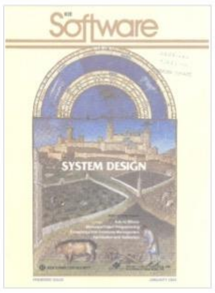
"***Developers of systems of systems*** *face challenges such as heterogeneous, inconsistent, and changing elements; continuous evolution and deployment; decentralized control; and inherently* ***conflicting and often unknowable requirements****.*"



2016

Butler W. Lampson, Hints for Computer System Design, IEEE Software 1984, no. 1, p. 11

"*There probably isn't a best way to build the system or even a major part of it. Much more important is to* ***avoid choosing a terrible way*** *and to have a clear division of responsibilities among the parts."*



1984

The Accidental Architecture G. Booch - IEEE Software 2006 (issue 3), p. 9

"*Philippe Kruchten has observed that 'the life of a software architect is a long and rapid succession of* ***suboptimal design decisions*** *taken partly* ***in the dark****.'* "

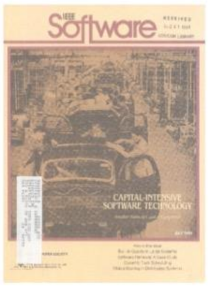
2006

Peter Wegner, Capital-Intensive Software Technology Conclusion, IEEE Software 1984, no. 3, p. 43

"*Periods of **rapid** technological **change** require more **innovation** and **greater risks** than periods of stability.*"

M. Vierhauser, R. Rabiser, P. Granbacher, "Monitoring Requirements in Systems of Systems", IEEE Software 2016 (issue 5), p. 22

"*The **fast-changing nature of our field** is one of the things that make working in software so much fun—and so challenging.*"


1984

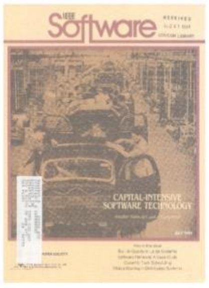
2016

E. Prell and A. Sheng, "Building Quality and Productivity into a Large Software System," in IEEE Software, vol. 1, no. 2, , pp. 47-54, 1984.

"*The key to developing larger and more complex software systems is to improve the **way we manage people** and information.*"

A. B. Pyster and R. H. Thayer, "Guest Editors' Introduction: Software Engineering Project Management 20 Years Later," in IEEE Software, vol. 22, no. 5, pp. 18-21, Sept.-Oct. 2005.

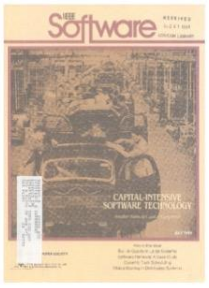
"*Twenty years is a very long time in the computing field. Yet, SEPM([Software Engineering Projects Management])'s progress has been agonizingly slow in many ways, probably because it's driven more by human behavior than by technology. **People change their behavior much more slowly than technology advances**.*"


1984

2005

Peter Wegner, Capital-Intensive Software Technology, IEEE Software 1984, no. 3, p. 9

"The greater speed of technical change means that capital investment must be recovered more quickly and that enhancement and evolution consume proportionately more resources than in a slowly changing technology. This contributes to the fact that ***maintenance and enhancement are the dominant costs in the software life cycle today****."*



1984

H. M. Sneed and C. Verhoef, "From Software Development to Software Assembly," in IEEE Software, vol. 33, no. 5, pp. 80-85, 2016

"You could view ***maintenance as an impending operational cost tsunami****, owing to seismic development activities. It's no longer tenable to keep creating new individual solutions to the same basic problems because those solutions must be maintained as long as they live, binding expensive human resources that are constantly declining."*

2009

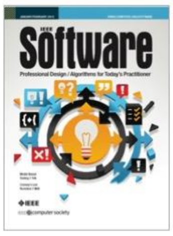
Alexander L. Wolf, Lori A. Clarke, Jack C. Wileden, Ada-Based support for programming-in-the-Large, IEEE Software 1985, no. 2, p. 58

"In essence, programming-in-the large involves the two complementary activities of modularization and ***interface control****. Modularization is the identification of the major system modules and the entities those modules contain, where entities are language elements that are given names, such as subprograms, data objects, and types. Interface control is the specification and control of the* ***interactions among entities*** *in different modules."*

1985

Frank Buschmann, To Boldly Go Where No One Has Gone Before, IEEE Software 2012, no. 1, p. 23

"The ***'things between things'*** *require the architect's full attention: domain concepts hidden between the lines of code;* ***interactions and interfaces*** *residing between components; and even choices between design options. This is the architect's territory, and successful architecture uncovers the things 'in-between' as early as possible, make them explicit, and decide about them!"*



2012

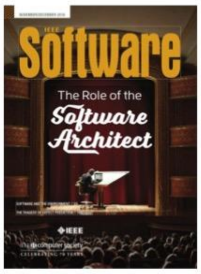
Robert S. Arnold, Roger J. Martin, Software Maintenance, IEEE Software 1986, no. 3, p. 16

"*All too often* ***maintainers*** *are faced with* ***hard-to-understand software*** *for which documentation is missing or out of date. Re-creating the documentation is typically out of the question in deadline-driven maintenance shops because of the time required and the difficulty in understanding software previously maintained without programming style standards.*"

1986

G. J. Holzmann, Hi Maintenance - IEEE Software 2016 (issue 6), p. 98

"*High-maintenance code not only is verbose but also tends to rely on* ***unstated, poorly stated, or incompletely stated assumptions****. If you want to understand that type of code, you need long chains of reasoning to figure out how and why it works, and under which conditions it could start failing when other parts of the system are updated. The reliance on hidden assumptions is probably the most telling feature of high-maintenance code.*"



2016

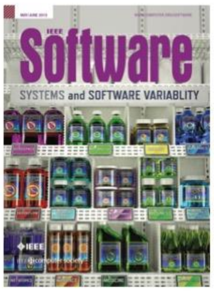
Ben Shneiderman, Philip Shafer, Roland Simon, Linda Weldon, Display Strategies for Program Browsing: Concepts and Experiment, IEEE Software 1986, no. 3, p. 16

"***User interfaces*** *in the software environment are much like* ***spices in good recipes****; the right arrangement must be found or the food will not show its full flavor. Factors such as data availability and complexity and the size of the display must be* ***carefully weighed*** *and accounted for in the design of any software environment.*"

1986

H. Khalid, E. Shihab, M. Nagappan, A. E. Hassan, What Do Mobile App Users Complain About? - IEEE Software 2015 (issue 3), p. 70

"*Interface Design: The user complained about the* ***design****,* ***controls****, or* ***visuals****. 'The design isn't sleek and isn't very intuitive.'*"



2015

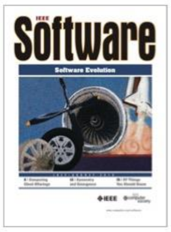
R.C. Linger, M. Dyer, H.D. Mills, Cleanroom Software Engineering, IEEE Software 1987, no. 5, p. 19

"*Software quality can be engineered under **statistical quality control** and delivered with better quality.*"

1987

H. Zhang and S. Kim, "Monitoring Software Quality Evolution for Defects," in IEEE Software, vol. 27, no. , pp. 58-64, 2010.

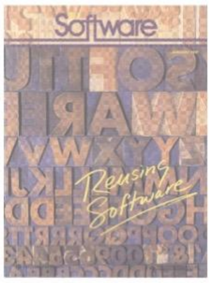
"*We used the c-chart, **a quality control chart** that's widely adopted in statistical process control (SPC) to study the quality evolution of two well-known, large-scale open source software systems … c-charts and patterns can help QA teams **better monitor quality** evolution over a long period of time… The quality evolution patterns in c-charts are **useful to understand the overall quality history** and thus to prioritize QA efforts efficiently.*"



2010

Ruber Prieto-Diaz, Peter Freeman, Classifying Software for Reusability, IEEE Software 1987, no. 1, p. 6

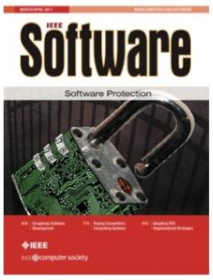
"*To **reuse** a software component, you **first have to find it**.*"



1987

G. Kotonya, S. Lock, J. Mariani, Scrapheap Software Development: Lessons from an Experiment on Opportunistic Reuse - IEEE Software 2011 (issue 2), p. 68

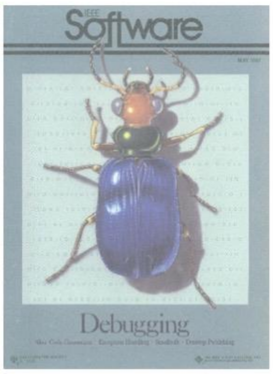
"*If we can **find efficient ways** to salvage and **reuse these components**, we might also recover some of the original investment and provide a rapid, low-cost means to develop new products.*"



2011

R.E. Seviora, Knowledge-Based Program Debugging Systems, IEEE Software 1987, no. 3, p. 20

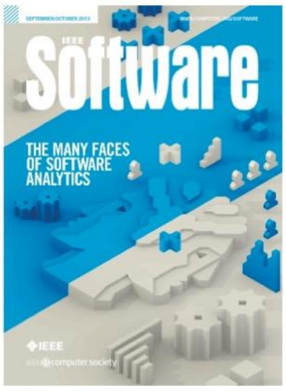
*"**No one likes to debug programs**, and there is no way to automate the task."*



1987

D. Spinellis, Differential Debugging - IEEE Software 2013 (issue 5), p. 19

*"If estimating the time needed for implementing some software is difficult, coming up with a **figure** for the **time required to debug** it is nigh on **impossible**"*



2013

Charles F. Martin, Second-Generation CASE Tools: A Challenge to Vendors, IEEE Software 1988, no. 2, p. 46

*"Today tools help systems analysts, so **why aren't they widely used**?"*

1988

D. Spinellis, Differential Debugging - IEEE Software 2013 (issue 5), p. 19

*"The main purposes of the study were to understand which and how architectural languages (ALs) are used in the software industry, **why some ALs aren't used in practice**, and what AL features are lacking according to practitioners' needs."*

2013

B. Zhou, **Software research and development in China,** IEEE Software 1989 (issue 2), p. 53

"*The Cultural Revolution that started in 1966 halted all computer-science activity until 1976, letting* ***China fall far behind the rest of the world****. Three more years passed until equipment was installed and trained educators could begin teaching programming classes. With this base finally established, the Chinese government in 1979 developed a national plan for software research and development.* "

1989

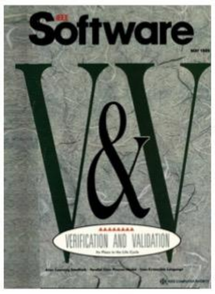
Z. Tang, M. Yang, J. Xiang, J. Liu, The Future of Chinese Software Development - IEEE Software 2016 (issue 1), p. 40

"*The Chinese software industry is* ***experiencing exciting, tremendous growth****. … the Chinese software industry's annual revenue grew from 1.3 trillion yuan in 2010 to 3.7 trillion yuan in 2014, a compound annual growth rate of 30 percent. This trend will likely continue into the next decade thanks to favorable government policies, growing demand for information infrastructure and services, fast-paced innovations from aspiring high-tech startups and industry heavyweights, and a steady supply of software engineering professionals.*"

2016

A. F. Ackerman, Lynne S. Buchwald, Frank H. Lewski, **Software Inspections: An Effective Verification Process**, IEEE Software 1989, no. 3, p. 31

"***Inspections*** *can detect and eliminate faults more* ***cheaply than testing****.*"



1989

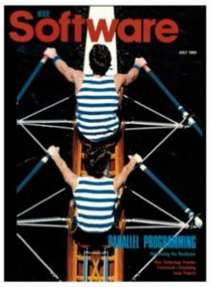
P. Rigby, B. Cleary, F. Painchaud, M. A. Storey and D. German, **Contemporary Peer Review in Action: Lessons from Open Source Development**, IEEE Software, 2012 (issue 6).

"*Software* ***inspection*** *is a form of formal peer review that has long been recognized as a software engineering 'best practice.'*"

2012

E. Chikofsky, **Looking for the best software engineers,** IEEE Software 1989 (issue 4), p. 107

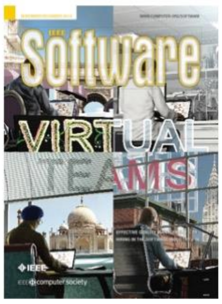
"*A strong common theme among the managers interviewed was the person's ability to* ***communicate*** *with both peers and managers … Most of the managers are looking for someone who will be a good* ***team player*** *- someone who can work for the good of the group and apply his skills and talents to assist collective goals. … I look for someone I can* ***motivate*** *and who wants to be motivated…*"



1989

T. Kaatz, **Hiring in the Software Industry**, IEEE Software 2014 (issue 6), p. 96

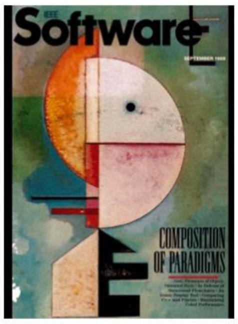
"*Do I think that there are some universals? Absolutely I do. And looking for a* ***team or cultural fit****, looking for people who are* ***motivated*** *and have* ***good communication*** *and good* ***collaboration****, my suspicion is that those are universal qualities that make people successful.*"



2014

Pamela Zave, **A Compositional Approach to Multiparadigm Programming**, IEEE Software 1989, no. 5, p. 15

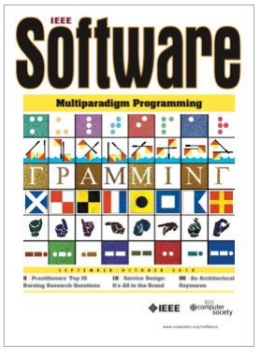
"***Multiparadigm programming*** *makes it possible to match the paradigm to the problem.*"



1989

D. Wampler, T. Clark, **Guest Editors' Introduction: Multiparadigm Programming** - IEEE Software 2010 (issue 5), p. 20

"***Combining paradigms*** *offers important benefits—for example, OOP minimizes the conceptual* ***gap between the problem domain and the implementation*** *in software, and functional programming (FP) brings mathematical rigor and robustness to computing, especially for concurrent applications.*"



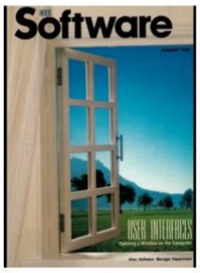
2010

Deborah Hix, **Guest Editor's Introduction: User Interfaces—Opening a Window on the Computer**, IEEE Software 1989, no. 1, p. 8

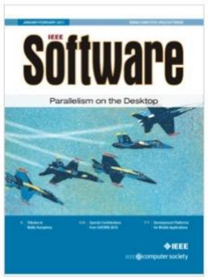
*"An **interactive system** - one with a human-computer interface - is not judged solely on its ability to compute. It is also judged on **its ability to communicate**. In fact, if users cannot communicate effectively with an interactive system, its computational ability may be inaccessible."*

Frank Buschmann, **Unusable Software Is Useless, Part 1**, IEEE Software 2011, no. 1, p. 92

*"Usability has a significant impact on the success of software-centric systems and products. It relates to the actual usage of a system, but also to its effective design and development. Ultimately, **failing to build usable software** may degrade a project's **ability to deliver** in time, budget, functionality, and quality."*



1989



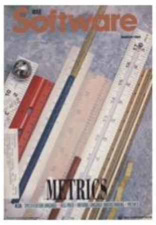
2011

Peter B. Dyson, Harlan D. Mills, **Guest Editors' Introduction: Using Metrics to Quantify Development**, IEEE Software 1990, no. 2, p. 14

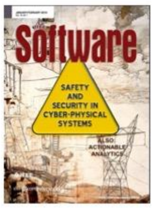
*"**You can't control what you can't measure**. That fundamental reality underlies the importance of software metrics, despite the controversy that has surrounded them since Maurice Halstead put forth his idea of software science. Skeptics claim metrics are **useless and expensive** exercise in **pointless data collection**, while proponents argue they are **valuable management and engineering tools**."*

Y. Yang, D. Falessi, T. Menzies and J. Hihn, **Actionable Analytics for Software Engineering**, in IEEE Software, vol. 35, no. 1, pp. 51-53, January/February 2018.

*"Although intensive research on **software analytics** has been going on for nearly a decade, a repeated complaint in software analytics is that industrial practitioners find it **hard to apply the results** generated from data science."*



1990


Software
Safety and Security in Cyber-Physical Systems
Also: Actionable Analytics

2018