Editor in Chief: **Sigrid Eldh**
Ericsson AB, Mälardalen University,
Carleton University
sigrid.eldh@ieee.org

A 40-Year Impact Perspective

Meet Your New Editor in Chief

Sigrid Eldh

WITH THIS ISSUE, I will take over as the new Editor in Chief (EIC) for *IEEE Software*, a magazine I have followed most of my working life. It will not be easy as I am following the formidable Ipek Ozkaya, who served here as EIC for the past five years. I am jumping in when all of next year's issues are already planned. What a fantastic setup to ease myself into the agenda. It is easy to forget that this magazine is built by so many skilled and wonderful coeditors, guest editors, reviewers, and authors, not to mention boards and committees. I might need to remind you—everyone is a volunteer. Very impressive!

In addition, we have professional and dedicated staff to keep it all aligned, to make systems work, and to ensure that we adhere to a large set of policies. Together, there is a strong will from all of us to keep the magazine timely and interesting for this grand community of smart and curious readers. I feel blessed to have this opportunity, and I hope that the magazine will continue to be a guide and an inspiration.

Together, there is a strong will from all of us to keep the magazine timely and interesting for this grand community of smart and curious readers.

Being in touch with research helps us to understand the frontiers, and being in touch with industry makes us aware of opportunities and challenges. We will address issues close to us who realize that software runs the world. The magazine is and will remain a beacon.

Are We Building the "Product Right"? Or the "Right Product"?

Forty years have passed since *IEEE Software* came to life. In the first issue of *IEEE Software*, I cannot help but smile when reading Prof. Barry Boehm's article "Verifying and Validating Software Requirements and Design Specifications."[1] In this article, there are two statements expressed that are probably repeated at every software testing course:

> "Verification. "Am I building the product right?"
> Validation. "Am I building the right product?""

By including such an article, *IEEE Software* has contributed to forming the field of software testing. The statement is profound, yet much has changed since then—most of all, the scale of things, but also the number of graduating computer scientists

Digital Object Identifier 10.1109/MS.2023.3328199
Date of current version: 20 December 2023

 0740-7459/23©2023IEEE

and software engineers. In 1984, I was in the first batch of computer science students in Sweden. We were then promised virtual keyboards and screens instead of the clumsy boxes. These virtual interactions were to be like moving paper on a desk, creating a keyboard by resizing a dot (like widgets) on the entire screen that was supposed to be like a cloth to put on any desk.

Today, we still use boxes of hardware, screens, keyboards, and cords, even if the technology for virtual keyboards and screens is here. The reason is more about business than technology, paired with a better understanding of ergonomics. The driver of software evolution is business. We simply like boxes. Boxes are tangible. And virtual reality software is much more difficult when it comes to getting it just right. It will still take time before we can tap on our virtual keyboards in the air. But why have keyboards at all when we can simply use eye tracking and voice recognition instead? We instead buy more things. Things are easier to sell. We must hope that the virtual software experience will be more sustainable—fewer boxes, more software. Obviously, the "right" in "right product" is not so simple. Today, our validation exercise is replaced with what closes the Dev Ops loop: automatic data collected from the user.

The development of "building the product right" has also changed over these 40 years. Verification has become testing. Testing has become diagnosing. We prefer tests to be generated, and if not, to be expressed directly in code to be repeated. Automated tests are far more efficient and effective. The other part of this statement is: What are effective ways to *build* software? How can organizations, with the right tools and ways of working, create the software reflecting the goal for the "product" or software system? How can we break down our ideas?

Today in agile organizations, requirements are often translated into textual "user stories." Is the architecture organizing these concepts enough to build software in the right way? In fact, in agile design, many organizations are no longer documenting design specifications at the level of detail sufficient for verification. It is too costly. One solution is to automatically capture the text and turn it into a model that can generate the code we want—or everyone must learn how to express ideas in a more software-friendly format. Making text into models and/or code is often the core of what drives "digitalization." Another solution to look at this is to say that we transform or curate the data to the right format for a model. My hope is that this transformation also includes a more verification-friendly format to be able to "build it right" automatically as an extra benefit.

Ongoing Digitalization

Today, system documentation has become one important data source serving localized large language models (LLMs) to query for a new generation of developers. Digitalization is ongoing in both industries and governments. And digitalization is especially important for those who have binders of documentation for the legacy code.

Software systems are seldom thrown out but have had to adapt. It takes a lot of effort to retrain staff to express themselves more formally in a model instead of writing a text, sometimes presented as slide text. It is costly for most industries to transform manual work to automation and prepare the software to be tuned by AI models for which customers may not be

CONTACT US

AUTHORS

For detailed information on submitting articles, visit the "Write for Us" section at www.computer.org/software

LETTERS TO THE EDITOR

Send letters to software@computer.org

ON THE WEB

www.computer.org/software

SUBSCRIBE

www.computer.org/subscribe

SUBSCRIPTION CHANGE OF ADDRESS

address.change@ieee.org
(please specify *IEEE Software*.)

MEMBERSHIP CHANGE OF ADDRESS

member.services@ieee.org

MISSING OR DAMAGED COPIES

contactcenter@ieee.org

REPRINT PERMISSION

IEEE utilizes Rightslink for permissions requests. For more information, visit www.ieee.org/publications/rights/rights-link.html

Digital Object Identifier 10.1109/MS.2023.3332788

willing to pay. This is the dilemma of software development. Who will pay for upgrades, better developer tools and infrastructure, tests, and migrating to the latest platform? It is still very hard to make clear business cases for maintenance costs and to truly honor backward compatibility in upgrades.

Today, the integration of software into ecosystems makes it even more costly to untangle software, processes, and tools. Yet, untangling is what most of us need and want. We need tools to aid us in automatic refactoring, something that is rather difficult, but often necessary, to be able to keep up with new demands.

Code Cloning and the Need for Automatic Refactoring

By looking at code cloning research, it is relatively easy to identify clones, but unfortunately, much of this research stops at this first step. So now what? Knowing that your code is filled with clones is like a measurable lack of communication (and the lack of reusable libraries). The solution should of course be to automatically refactor and remove these clones from the code base. We still have a long way to go to make sure the refactored code is equivalent to the old code, with the same or improved performance, and with fewer faults. This is too time consuming today, except maybe for type 1 clones (the duplications)—if they are big enough.

In addition, most of us rely on unit tests for verification, which would be rather useless after such a transformation. No customer wants to pay for this, even if it increases effectiveness, lowers the code footprint, and saves energy. You still need to simultaneously create new sellable functionality.

Can Large Languages Models Solve Our Refactoring and Verification Needs?

It is no wonder why we investigate having LLMs aid us in generating more effective and better code. Hopefully, these models can be trained to do this more accurately. LLMs are notorious for giving us something that looks correct but is, in fact, wrong. These are known as *hallucinations*.

For smaller players and systems, refactoring is less of an issue. For large organizations and systems, it is a necessity to invest in the management of the constantly increasing complexity and the system evolution. The "keep it simple" mantra does sometimes require us to completely start over, but that investment is often too steep. Instead, most companies work hard to get their own LLM to learn their patterns and make sure to limit bugs. Can the developers generate the code and then continue to happily generate the tests with these LLMs? How do you know what is right? The code or the test? Is there only one answer to this question? Is this ground truth, or is this maybe a perspective in a specific context that something is right?

What Is Right for Whom?

What is right and for whom can be both an ethical and a scientific question. It is easy to create a belief system where you think you know the truth. Critical thinking is a skill we must improve as software engineers. Improvement should be both in the scientific method and in understanding the context. Where does "right" come from? The requirements? The design documentation?

As a tester, critical thinking is essential, but does that alone

automatically make you the best test oracle? Can you really know that what you are deciding is always right? I hear people discuss the sequence of the multiple trained AI models that are used to fix every step in the process. This ranges from cleaning the data and transforming it to analyzing it and creating an action. Both reinforcement learning and supervised variants are interesting techniques in the sense that it is important *who* provides the ground truths for these models. With what knowledge do you tell what is right? Is it always obvious? Probabilities might not be right either—as the "right answer" really could be in the outlier.

As a result, it is not hard to understand the fear some humans have of AI. Imagine having to explain to a future automatic bot that the issues and contexts for your case are unique. Have not most of us already had that experience in some horrible phone guidance system? As software engineers, we must take this into account. We must be thinking about how to allow for more "diverse" models by thinking more holistically.

One new approach is that we aim to train domain experts to write software with automatic code bots instead of having a wall between users (domain experts) and developers. Are LLMs not just exactly that—developer support? We just need to practice "how to ask the questions right" and express ourselves more clearly. The ground truth of what is right might often come from users. If the user is "another bot" or a machine interface, what then? Learning to program on a higher abstraction level will be like learning a foreign language. Our brain processes it similarly, as this study[2] partly indicates.

We need to question ourselves about what we observe and if that fits "for whom" it is intended. Is what we observe "right" and in what context? One example is if we are looking for a bug in our AI model, this will look completely different from someone trying to figure out the ecosystem of an AI model. This is why abstraction levels make sense.

Making any knowledge or process more observable, possible to debug, and possible to explain should be a quest for truth.

A Quest for Truth!

Making any knowledge or process more observable, possible to debug, and possible to explain should be a quest for truth. But are the data sufficient? We humans are not as self-aware as we think we are, so maybe handing some of the smaller decisions over to the computer would not be so bad after all. Having the human in the loop is essential for evolution. Otherwise, we will miss both innovation and new insights, even if some deep learning models can show us new patterns. Do not forget the unique creativity needed and (sorry for repeating myself) the quest for truth. Our human perspective is unique, and it matters. One single person's truth also matters. You are important.

We should be satisfied with the things that are getting better. Software engineering is key in this transformation as well as in making it easier to both build and test. Still, we must ask ourselves: What is better? More effective? More sustainable? More trustworthy? More fun? More challenging? Or maybe more secure? Or are you simply holding on to what is known? Change will happen, that is for certain.

Therefore, it is with pleasure that I present an issue that has the theme of observability and explainability of system decisions. It will be as valuable as it has always been to dive into the articles.

I can only conclude with the hope to fulfill the first as well as the last EIC's wish to address more industry-relevant aspects in this magazine. Remaining open to new technology shifts, which the industry needs to respond to, is important. These technology shifts are happening now.

References

1. B. W. Boehm, "Verifying and validating software requirements and design specifications," *IEEE Softw.*, vol. 1, no. 1, pp. 75–88, Jan. 1984, doi: 10.1109/MS.1984.233702.
2. B. Floyd, T. Santander, and W. Weimer, "Decoding the representation of code in the brain: An fMRI study of code review and expertise," in *Proc. IEEE/ACM 39th Int. Conf. Softw. Eng. (ICSE)*, 2017, pp. 175–186, doi: 10.1109/ICSE.2017.24.