Software and Systems Modeling (2023) 22:1969–1976
https://doi.org/10.1007/s10270-023-01136-y

EXPERT VOICE

On the persistent rumors of the programmer's imminent demise

Hessam Mohammadi[1] · **Wided Ghardallou**[2] · **Elijah Brick**[1] · **Ali Mili**[1]

Received: 28 July 2023 / Revised: 3 August 2023 / Accepted: 25 September 2023 / Published online: 15 November 2023

Abstract
Since the dawn of programming, several developments in programming language design and programming methodology have been hailed as the end of the profession of programmer; they have all proven to be exaggerated rumors, to echo the words attributed to Mark Twain. In this short paper, we ponder the question of whether the emergence of large language models finally realizes these prophecies? Also, we discuss why even if this prophecy is finally realized, it does not change the job of the researcher in programming.

Keywords Programming languages · Automatic programming · Programming profession · Large language models

1 The recurring obituary of programming

In May 1897, while he was on a speaking tour in London, American author Samuel Clemens, known by his pen name *Mark Twain*, was contacted by an English journalist of the *New York Journal*, who wanted to confirm rumors that were circulating in New York to the effect that he was gravely ill or even dead. Mark Twain wrote a response, part of which was published in the *New York Journal* of June 2, 1897: *"I can understand perfectly how the report of my illness got about, I have even heard on good authority that I was dead.... The report of my death was an exaggeration"*. Mark Twain is the only person known to history who had the privilege of commenting on the announcement of his death.

Whereas Mark Twain survived rumors of his death only once, the professional programmer keeps dealing with such rumors on a regular basis, as we discuss in this paper.

✉ Hessam Mohammadi
hm385@njit.edu

Wided Ghardallou
wided.ghardallou@gmail.com

Elijah Brick
eb275@njit.edu

Ali Mili
mili@njit.edu

[1] CS Department, NJIT, University Heights, Newark, NJ 07102, USA

[2] Department of Computer Science, University of Sousse, Rue Khalifa Karoui, 4000 Sousse, Tunisia

2 Automatic programming

As preposterous as it may sound today, the emergence of *high-level* programming languages such as *Cobol*, *Fortran* and *Algol* in the late fifties of the twentieth century was hailed as the advent of automatic programming and the end of the need for programmers. Of course, for programmers who were accustomed to writing software in the form of instruction codes and binary addresses, programming in high-level languages does sound like writing specifications; the rudimentary compilers of the time did sound like they were generating code automatically. Also the development of compilers for these high-level languages prior to the emergence of compiler design theory, syntax directed translation, and modern compiler generation tools did represent a significant technical achievement.

But the era of spectacular software failures of the sixties and seventies was a sobering experience: It quickly dispelled any fantasy anyone had about automatic programming, and led to the realization that programmers, analysts and designers, far from being an endangered species, are actually needed more than ever [1]. It also led to the realization that the very idea of automatic programming is a fleeting concept, that is relative to current technology and current market demands; as both of these evolve, so does the characterization of automatic programming. This is best articulated by David Parnas in [2]: *"In short, automatic programming always has been a euphemism for programming with a higher-level language than was available to the programmer. Research in auto-*

matic programming is simply research in the implementation of higher-level programming languages".

3 Knowledge based software engineering

The decade of the eighties brought a fresh, multi-pronged assault to the programmer's job security, through several distinct but interdependent initiatives:

- The emergence of logic programming as a viable programming paradigm and its implication that logic programming languages such as *Prolog* (*Pro*gramming in *Log*ic) enable us to formulate specifications as relationships between inputs and outputs, and let the interpreter figure out how to derive the output from the input. With logic programming, we no longer need programmers to map inputs to output, and Prolog-like interpreters can do it automatically, using SLD (Selective Linear Definite clause) resolution.
- The emergence, with great fanfare, of the initiative of *Fifth Generation Computing* [3]. This initiative was launched by the Japanese government, with the aim of revolutionizing the field of computing, and caused other jurisdictions (The USA, Canada, Europe) to scramble to catch up with similar initiatives, whose focus was artificial intelligence, knowledge-based systems, and expert systems. Many of these jurisdictions were sensitive to the possibility that after dominating the auto industry then the electronics industry, Japan was poised to dominate the computing industry.
- Much of the focus of the fifth generation computing initiative is on knowledge-based engineering, including knowledge-based software engineering, whose premise is that we can support software engineering by capturing, storing and deploying programming knowledge and domain knowledge in knowledge bases.
- The advent of the personal computer, along with the emergence of the *Microsoft DOS* operating system, led to the democratization of programming, in the sense that anyone who can afford a personal computer can be a programmer; implicit in this premise is the subtle suggestion that we no longer need professional programmers.

By the end of the decade, it was becoming clear that logic programming is no match for the requirements of the software industry: Prolog has limited means for data modeling and representation, so that most problems cannot be modeled as logic programs, and most of those that could were beyond the capability of Prolog interpreters. Also, most of the claims and hype of fifth generation computing did not materialize, leading to a loss of interest and a decline in funding. Furthermore, while there were some research advances in artificial intelligence, they did not translate into significant advances in the state of the practice. Finally, the expectations and claims of knowledge-based software engineering were getting dramatically scaled down: far from replacing programmers and software engineers, the focus was becoming to offer them apprenticeship support [4, 5].

In the meantime, as the demand for software products continued unabated, notwithstanding all the hype around logic programming, most software was developed in C and C-like languages; when the US DoD launched an international competition for the design of a programming language to use across its wide range of platforms, it is reported that virtually all the competing proposals were variations on a C-like language (Pascal), as is the selected language Ada [6].

4 Reuse based software engineering

The decade of the 1990s brought about another set of threats to the profession of the programmer, and associated claims that we would soon be able to do without programmers.

4.1 Software reuse as a storage and retrieval paradigm

Whereas the 1970s was the decade of the *structured* disciplines (structured programming, structured design, structured analysis, etc) and the 1980s was the decade of the *knowledge-based* disciplines (knowledge-based programming, knowledge-based design, knowledge-based software engineering, etc), the 1990s was the decade of *reuse-based* disciplines: this includes primarily reusing source code; but it also includes specification reuse, design reuse, cost estimation reuse, etc. There is a sound rationale for evolving and refining a discipline of software reuse: reuse is an integral part of all engineering disciplines, and no engineering discipline needs the gains in productivity and quality that stem from reuse as much as software engineering does; indeed, software engineering is typically characterized by poor product quality and low productivity.

Software reuse was all the rage during the 1990s, with the proliferation of specialized conferences, workshops, conference tracks, special issues, etc. Wild claims were made about the impact that software reuse would have on programmer productivity and program quality, and how we would need much less code to be written from scratch. Much of the research on software reuse was focused on the question of storage and retrieval of software components, a question that turned out to be of limited significance, in practice. But when governmental agencies created repositories of free software, set up sophisticated retrieval procedures of software components, and encouraged software engineers to avail themselves

of this free resource, they were utterly disappointed by the outcome: there were no takers.

Throughout the decade, a number of premises were emerging to explain the lack of reuse in software engineering practice:

- *The Unlikelihood of Functional Match*. The functional specification of a software component is typically a very detailed artifact, and so is the specification of a user query/requirement. As a result, the likelihood of a match is very low.
- *The Not Invented Here Syndrome*. Even if a functional match succeeds, programmers may be reluctant to integrate a component in their code if they do not entirely trust how it was developed and by who.
- *Architectural Match*. Functional match is a necessary condition for software reuse, but not a sufficient condition: Architectural match is also necessary. In other words, in order for a software component to be reused, the architecture within which the component can be reused must be compatible with the architecture within which it was developed. Hence not only is functional match rare, but even when it occurs, it does not ensure success; we also need architectural match.
- *The Need for Planning*. The idea that one can develop a software product by scavenging through a repository of free software has proven to be unrealistic. The most realistic scenario for reusing a software component is a scenario where the developer of the component intends for it to be reused, prepares it for reuse, and specifies the terms and conditions of its reuse.

As a consequence, a consensus started to emerge around the recognition that the idea of *opportunistic reuse*, whereby one picks software components from random repositories and integrates them into a cohesive software product, is a naive pipe dream. Instead, it was becoming increasingly clear that in order for software reuse to happen, a number of conditions must be met: the person producing reusable software and the person reusing it must have a shared software architecture in mind; the jointly adopted software architecture must stem from a thorough effort in *domain engineering*; in addition to defining a reference architecture, domain engineering must also define the *Application Engineering* protocol, which is the carefully choreographed process of selecting, adapting, and composing reusable components from specific application requirements. These premises are in effect the tenets of the discipline of *Software Product Lines*, which emerged as the sole form of viable/practical software reuse. Software product lines made their way into several sectors of the software industry, and rather than put an end to the profession of programming, they redefined it. Much of the domain engineering activity of product line engineering revolves around data modeling and data representation, hence benefits from advances in model engineering and data engineering.

4.2 Design for reuse: object oriented programming

With its emphasis on encapsulation and genericity, object oriented programming is a natural fit for software reuse: encapsulation supports software storage and retrieval and genericity supports software adaptation in the context of whitebox reuse. Hence, object oriented programming prospered alongside software reuse during the 1990s, initially driven by the interest in Smalltalk [7], and subsequently leading to the development of object oriented languages or the expansion of existing languages with object oriented features [8].

At its core, object oriented programming is a discipline of modular programming, and object oriented languages are designed to support modularity. But modular programming is 90% programming discipline and 10% programming language use; so that it is possible to write modular programs in non-object oriented languages and non-modular programs in object oriented languages. Be that as it may, the focus on object oriented languages pervaded academia and industry; a consequence of the pervasiveness of object oriented practice in academia is that generations of students were trained to be fluent in all the concepts (and buzzwords) of object oriented programming (inheritance, genericity, polymorphism, encapsulation, etc) but were not adequately trained in the simple skill of writing a straight domain-to-range function or designing a simple algorithm. Also, the obsession of modeling everything as objects (when simple data types are adequate) often leads to unnecessarily complex and opaque code.

4.3 Design with reuse: component-based software engineering

Concurrent with the interest in reusing software components was the interest in building systems from reusable components. This discipline was focused on systems integration and systems validation issues, and is technically indistinguishable from application engineering. But in practice component-based software engineering (CBSE) is most effective when it is carried out according to the application engineering protocol of a product line initiative.

5 Product line engineering

By the first decade of the millennium, a consensus emerged about the premise that product line engineering is a (the only?) viable technology that synthesizes much of the research of the previous decade and much of its best prac-

tices. The practice of product line engineering became the standard modus operandi of sophisticated/specialized software development organizations that develop applications within a limited application domain for special market segments, by investing a significant amount of resources into developing corporate expertise in the domain knowledge of the market they are serving. Rather than mark the end of the programming profession, software reuse/product line engineering marked its evolution into an efficient, streamlined process that makes optimal use of programming talent and domain-specific expertise.

6 Large language models

By the end of the 1980s, artificial intelligence was thoroughly discredited, as much of its promises, which were over-hyped and over-sold through the decade, failed to be fulfilled. Yet some hardcore believers pursued work in neural nets and machine learning, and produced increasingly sophisticated machine learning algorithms; as training data became increasingly abundant, the performance of these algorithms became increasingly convincing, culminating recently in Large Language Models, and increasingly confident talk of AI-generated code. But opinion on the potential of this technology to alter the business of software development remains sharply divided.

6.1 The hopeful vision

Nowadays, the announcements of the programmer's demise are not even subtle. In an article titled *The End of Programming* [9], Matt Welsh declares colorfully: *"The end of classical computer science is coming, and most of us are dinosaurs waiting for the meteor to hit"*. Welsh presents a very attractive vision of a future where, rather than developing systems through *programming*, we will develop them through *training*. To this effect, we do not need an original algorithm for each new application, we just need one highly specialized learning algorithm, one that can analyze massive amounts of learning data and synthesize behavioral rules therefrom. Because of its critical dependency on data, generative AI depends critically on our ability to model and represent data. Welsh argues that the new atomic unit of computation is not a predictable, static process governed by instruction sets but rather massive, pre-trained, highly adaptive AI models. Welsh acknowledges that along with the massive potential of this new computation paradigm, comes an equally significant level of risk that stems from the reality that *nobody actually understands how large AI models work*. Welsh describes AI models, the new building blocks of future computation, as *temperamental, mysterious, and adaptive agents*.

6.2 The reasoned skepticism

In an equally self-assured paper titled *The Premature Obituary of Programming* [10], Daniel Yellin pours cold water on Welsh's prediction by pointing to several flaws in the *Deep Learning* approach to programming, which he calls *Deep Programming (DP)*:

- *Lack of Adaptability*. Because it is based on legacy code, AI-generated software suffers from intrinsic limitations, including:
  - Inadequacy with respect to new machine and network architectures.
  - Inadequacy with respect to new programming frameworks for solving problems.
  - Inadequacy with respect to new types of problems and concerns.
- *Operating Cost*. The models built by Deep Programming are huge, as they involve billions of parameters and require massive computational power.
- *Specifying Requirements*. Deep Programming is worthwhile only if it is much easier to write the requirements specification of a program than to write the program; writing precise, complete, minimal requirements specifications is an intractable problem.
- *Stalled Innovation*. If all new code is produced from synthesizing existing code bases, then the diversity of the global software repository will stall. What makes the current pool interesting is the diversity that comes from a vast set of programmers populating these repositories, each with its own unique problem-solving techniques and programming styles.
- *Programming as a Social Endeavor*. Large-scale software development is not a straightforward transformation from a specification to a finished software product; rather it involves complex social interactions within development teams, such that the final product specification may be significantly different from the original intent. Such an interaction, that often leads to a better product, is not possible with AI-driven development.

We recognize some further obstacles to the wide use of AI-driven software development methods:

- *Opacity*. The generation of software products from AI prompts bridges two massive gaps:
  - *Prompt Generation*: The gap between what the user means and what the user writes.
  - *Prompt Interpretation*: The gap between what the user writes and what the AI model understands.

Add to that the gap between what the user means and what the user actually needs, but that is not specific to AI-generated code.

- *Requirements Specification*. One of the most intractable issues in software engineering is the issue of requirements engineering and requirements specification, including the tasks of data modeling, data representation, and model validation. This question has mobilized researchers for decades, yielding the design of several specification models, theories, and languages, for little impact on the state of the practice. AI-based code generators give little thought to this massive problem, and assume that user prompts are sufficiently precise to capture requirements in a complete and minimal manner.
- *Scale*. Whether software is developed by human programmers or by AI models, the problem of specifying the requirements is the same. In practice, this is feasible under two possible circumstances:
  - Either the targeted program is small, so that it is possible to write short, simple specifications for it.
  - Or the targeted program is an instance of product line, where a large portion of the requirements is implicit, and the specification involves pinning down some minor options in the form of pre-codified variabilities.

  In either case, the contribution of AI is limited in scale: in the first case (cited above), it consists of generating a limited-size program; in the second case, it consists of a limited-scale adaptation and integration task.
- *Quality Concerns*. Because it is based on legacy code, AI-generated software is likely to be as flawed as the legacy code it is based on.

In summary, the role of *Deep Programming* is best articulated by Yellin [10]: *DP will not replace programming. Its aim should be to increase the productivity of software development and thereby make up for the significant shortage of programmers today*. This sounds like a verbatim echo of the spirit of the *Programmer's Apprentice* [4] articulated three decades earlier.

7 Conclusion

7.1 The grandmother's wisdom

In a painting published in 1947, titled *The Outing* (Fig. 1), Norman Rockwell depicts an American post-war family going to the lakeshore in the morning, and returning home in the afternoon. Rockwell emphasizes the contrast between the expression of excitement, enthusiasm and expectation that radiates on everyone's face in the morning, and the subdued expression of apathy, exhaustion and perhaps a tinge of disappointment (it was not such a big deal after all). The only person who displays the same facial expression on the way out and on the way back is the grandmother: Rockwell took her picture in the morning, turned it around and copied in the afternoon's version.

The contrast between the grandmother's detached/skeptical attitude and everyone else's is a good metaphor for the community's diversity of reactions to successive software engineering trends that we can observe through the decades: whereas many researchers and practitioners get carried away by the prospect of fundamental /radical breakthroughs that will change the state of the art and state of the practice in software engineering, not everyone subscribes to the hype. So far the grandmother's skepticism has proven to be the more rational attitude, as most past trends have led to modest incremental changes in the business of software engineering, rather than the radical transformations that were originally envisioned. The question that arises naturally is then: is AI-based code generation just another fad, or is it different this time? The fact that some top universities have already altered the way they teach programming may suggest that this time the hype of the new trend may be justified [11]; but to the extent that programming-in-the-small and programming-in-the-large are two distinct paradigms (rather than two sizes of the same paradigm) suggests that the success of AI-based code generation in a classroom setting does not necessarily translate into successful deployment in an industrial context.

7.2 Research implications

In the meantime, the practice of software engineering remains relatively unchanged. Two of the most intractable issues in software engineering are as follows:

- *Software Specification*. This includes stakeholder identification, requirements elicitation, collection and compilation, data modeling and validation, requirements specification and specification validation. Faulty requirements specification remains the cause of the vast majority of software failures [12].
- *Program Verification*. This involves ensuring that the program meets its specification, through a combination of methods: testing, static analysis, process controls, etc.

AI-based code generation has no impact on any of these two aspects of software engineering; if anything, it probably makes verification more complex because it precludes process controls, since AI operation is notoriously opaque. As far as the issue of software specification is concerned, the emergence of generative AI has given rise to a new engineering discipline, namely *prompt engineering*, which deals with formulating queries to generative AI tools. Until and unless

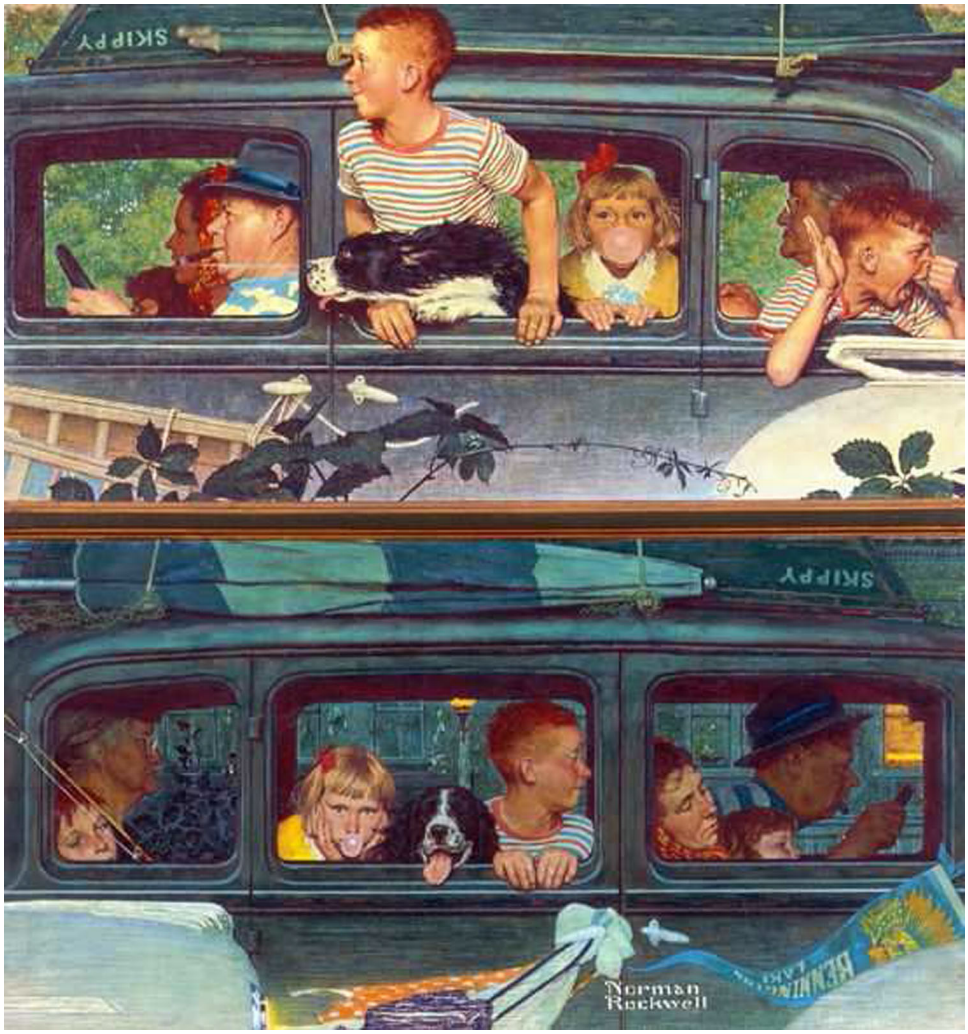


Fig. 1 *The Outing*, Norman Rockwell, 1947

prompt engineering reaches the scale and expressive power of requirements engineering (which has been the subject of research for decades), AI-based code generation may remain limited to small scale software development.

Be that as it may, much of the code developed, evolved and maintained nowadays is written in C-like languages. The five top-ranked languages in the April 2023 Tiobe classification of programming languages (https://www.tiobe.com/) are derived from C or inspired by it:

1. Python: 14.51%.
2. C: 14.4 %.
3. Java 13.23 %.
4. C++: 12.96 %.
5. C#: 8.21 %.

The Tiobe Index of a programming language measures the frequency of internet searches of the programming language in the main search engines, normalized to the total number of searches of programming languages; these indices are interpreted as proxies for the frequency of use of each language. These five languages account for a total of 63.31 % of programming language use worldwide, and all five are trending upward by comparison with April 2022.

Hence, the problem of ensuring the correctness, reliability, safety and security of software artifacts has changed little since the nineteen seventies: we are still considering programs written in C-like syntax and pondering the question of whether they are correct, reliable, safe or secure. One way to answer these questions is to capture the full domain-to-range function that programs define between their input space and their output space. But doing so may be too costly, and not very effective: programs are complex artifacts, whose function may be hard to express in closed form; also, not all the functional details of a program are worth extracting (what value a program assigns to auxiliary variables may be of little interest to the user, or may be irrelevant to the specification). Hence, we may leverage the ability to compute program function, not only to compute the function of the program in full detail, if needed, but also to answer queries

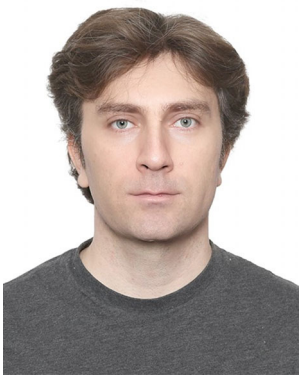
about the state of the program at selected labels, or about the function of program parts. To this effect, we propose a vocabulary of functions that enable a user to query a C-like program about its semantic properties [13]:

- *Assume()*. This function takes a condition as a parameter and declares an assumption about the state of the program at some label or the function of some program part. This function may be used, in particular, to specify the precondition of a program or routine.
- *Capture()*. This function refers to a program label or to a program part, and returns a characterization of the program state at the selected label or a characterization of the function of the selected program part.
- *Verify()*. This function takes a condition as a parameter and tells whether the condition holds at some state of the program, or about the function of a program part. This function may be used, in particular, to verify the postcondition of a program or routine.
- *Establish()*. This function takes a condition as a parameter and uses program repair technology [14] to modify the program so as to make the function *Verify()* return true, while originally it returned false.

A brief demo of a protype that implements these functions is available at http://web.njit.edu/~mili/acvedemo.mp4.

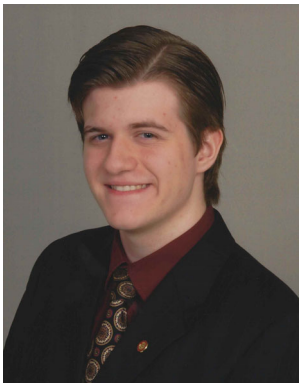
References

1. Brooks, F.P.: The Mythical Man-Month. Addison Wesley, Boston, MA (1975)
2. Parnas, D.L.: Software Aspects of Strategic Defense. Scientific American (1985)
3. Meyers, P.J., Yamakoshi, K.: The Japanese fifth generation computing project: A brief overview. J. Comput. Sci. Coll. **36**(2), 53–60 (2020)
4. Rich, C., Waters, R.C.: The programmer's apprentice: A research overview. IEEE Comput. **21**(11), 10–25 (1988)
5. Reubenstein, H.B., Waters, R.C.: The requirements' apprentice: Automated assistance for requirements acquisition. IEEE Trans. Softw. Eng. **SE–17**(3), 226–240 (1991)
6. Ichbiah, J.D.: Preliminary Ada reference manual. ACM Sigplan Notices. **14**(6a), 1–45 (1979)
7. Goldberg, A.: Smalltalk-80: The Interactive Programming Environment. Addison Wesley, Boston, MA (1983)
8. Stroustrup, B.: The C++ Programming Language. Addison Wesley (1986)
9. Welsh, M.: Viewpoint: The end of programming. CACM **66**(1), 34–35 (2023)
10. Yellin, D.M.: Viewpoint: The premature obituary of programming. CACM **66**(2), 41–44 (2023)
11. Dreibelbis, E.: Harvard's new computer science teacher is a chatbot. PC Magazine (2023)
12. Leveson, N.C., Thomas, J.P.: Certification of safety critical systems. Commun. ACM **66**(10), 22–26 (2023)
13. Mohammadi, H., Ghardallou, W., Mili, A.: Assume(), capture(), verify(), establish(): Ingredients for scalable program analysis. In: 10th International Conference on Model and Data Engineering, Tallin, Estonia (2021)
14. Gazzola, L., Micucci, D., Mariani, L.: Automatic software repair: A survey. IEEE Trans. on Soft. Eng. **45**(1) (2019)



Hessam Mohammadi received his Ph.D. in computer science from the New Jersey Institute of Technology in 2023. His research interests include software engineering, software verification, and static analysis. He also received his MSc in software engineering from Amirkabir University of Tehran (Tehran Polytechnic).

Wided Ghardallou is an assistant professor in computer science at the National School of Engineering (ENISO) in Sousse, Tunisia. She holds a Ph.D. in computer science from the Faculty of Sciences of Tunis (FST), Tunisia. Her research interests are in software engineering.



Elijah Brick is a technical analyst and information technology specialist at the Joint Enabling Capabilities Command. He provides subject matter expertise and recommendations to enable process improvements, data transformation, and automation in the workplace. He coordinates and provides guidance for the usage of an Enterprise Data and Analytics Environment. He works with engineers in an agile team to deliver tools that streamline and standardize workflows to enable data-driven decision-making. He also prepares data for use in artificial intelligence and machine learning solutions. Elijah earned a Master's degree in Cybersecurity and Privacy and a Bachelor's degree in Computer Science at the New Jersey Institute of Technology. He is a member of the CyberCorps Scholarship for Service program.

Ali Mili holds a Ph.D. from the university of illinois at Urbana Champaign and a Doctorat es-Sciences d'Etat from the Joseph Fourier University of Grenoble. He is a corresponding member of Beit Al Hikma in Tunis, Tunisia. He is also a Professor of Computer Science and Interim Dean of the Ying Wu College of Computing at the New Jersey Institute of Technology, in Newark, NJ.