Four Thought Leaders on Where the Industry Is Headed

Andrew Moore

ON THE BASIS of what’s going on at large Internet companies—plus what I see my star faculty working on—it’s clear that many of the new capabilities you’ll be seeing in software systems in 10 or 20 years have to do with the ability to adapt to information that’s streaming in all the time. In the future, software will much less frequently perform clear transactions that always return the same answer; instead, software processing will be about piecing together many disparate bits of information. Some of this is already happening in the ranking systems that companies such as Facebook and Google use, which govern how they present you personalized results. These systems are already ingesting tens of thousands of pieces of information every second, 24/7, day after day.

These systems provide all kinds of challenges for software engineers. How do you test and provide quality assurance on systems that are adapting all the time and will be running very differently over time on the basis of accumulated data? Both industry and academia are developing such methods, but this field is only in its infancy. If you’re a software engineer, it’s a crime for you not to be an expert in probability. You’ll be dealing with fuzzy information in many places, and you have to be very comfortable with how to combine it.

Similar complexities also apply to the verification of physical systems as they work in the real world, another emerging discipline about which I’m passionate. Here at Carnegie Mellon University, Andre Platzer is applying the formal methods that have proven useful for verifying system behavior to systems at scale such as those for air traffic control and self-driving cars. One way to visualize where this research might be headed: when two ultra-intelligent cars know they’ll collide, and they have a few seconds left before the collision, the negotiation between them must be about how they save as many lives as possible. Verifying that these complex systems can reach optimal answers regarding life-or-death decisions is what’s needed—not so much an “Asimov’s 3 Laws of Robotics approach” based on general principles...
for minimizing harm. Another of our CMU faculty, Aaron Steinfeld, is a pioneer in self-driving cars and has turned his attention to policy issues related to crashes.

Formal methods will be a key enabling technology. Over the years, many of our hearts have repeatedly been broken by the potential for formal verification methods and how they haven’t been able to reach the larger population of software engineers. Formal methods will continue to be important and will be used effectively as the underpinning of these new capabilities, although every software developer won’t be using them.

**Tim O’Reilly**

CODE FOR AMERICA is changing government and changing how we think about our role as software professionals. Its work is deeply rooted in the notion that you can no longer govern without using digital technology. Technology is central to how we deliver services today and how people access them.

The lessons we’ve learned from software development can be applied to make government work better for everyone. Right now, government still operates in ways used early in software development, when methods to manage software projects were more similar to those used to build a bridge. In describing the healthcare.gov site’s failure, Internet pundit Clay Shirky remarked that the waterfall method, in which everything is spelled out in excruciating detail beforehand, is essentially a promise “not to learn anything while doing the actual work.”

Today’s software progresses through countless small updates, learning as we go, and building iteratively to create products that work for their intended users. Code for America brings this iterative, data-driven, user-centered approach to building government software and programs. One place we’ve done this is with CalFresh, California’s version of the Supplemental Nutrition Assistance Program (what used to be called Food Stamps). The team discovered that there were not only software problems but also many problems with how the program interacted with its users. The team simplified the application process and ended up building a set of apps that could “debug” the CalFresh system, discovering where people were confused or discouraged, and understanding why the systems used to deliver the program were interfering with its ability to achieve its objectives.

Bringing programmers into government through programs such as Code for America (which works with cities and counties) or, at the federal level, the US Digital Service and the General Services Administration’s 18F unit, is teaching government to work in this user-centered, data-driven way. By putting the user rather than the government itself at the center of service design, we hope that Internet technologies will let us rebuild the kind of participatory government our nation’s founders envisioned.

Something else we can teach government: in the private technology sector, we’ve learned to build fundamental platforms and reuse them. We don’t build monolithic applications over and over again from scratch. We need to start building government software as a set of APIs...
and platforms. Government can take many practical steps right now to move in this direction, which I describe in my essay “Government is a Platform.” In the future, we can expand these ideas to figure out how government can become a more open platform that lets people inside and outside the government innovate and that lets outcomes evolve through interactions between the government and its citizens.

Another lesson from technology that we need to explore is how to build modern regulatory systems. You can think of Google search quality, banks’ credit card fraud detection, antispam systems, or the reputation systems used by many modern applications as a kind of regulation. But these regulatory systems work very differently from government regulatory systems, which basically specify a set of rules. Regulatory systems should focus on outcomes and adjust the rules (just as a fraud detection or search quality system continually adjusts to detect new kinds of attacks) to help achieve those outcomes. Too often, rule-based systems fail, and no one notices. We need to start thinking about government regulations: did they achieve their outcomes? If not, rethink or tweak them. Don’t just consider them a success because they achieved some innovative features and get valuable feedback. But this might not always fit well with safety- or life-critical applications—for example, a self-driving car or an implanted medical device.

We need to develop capabilities quickly—but we also need to deliver those capabilities with quality assurance and fewer vulnerabilities. Agile lets us get working software in front of users as quickly as possible but also need to develop them. The approach and mix will differ—there will be a different verification-and-validation (V&V) strategy for a game application than for the nation’s air traffic control system.

Furthermore, besides systems that are just larger or more connected, we’re entering an era of systems with that right amount of architecture to keep our velocity and quality high—and allow for continued system evolution without costly refactoring?

Other trends and advances are important. We’re building larger, more complex, and more connected systems. These systems are hard to test and verify through our traditional approaches. To address this challenge, we need to find new strategies to blend traditional testing, new advances in formal methods, modeling and simulation and automated testing, and continued data collection after fielding. Obviously, the approach and mix will differ—there will be a different verification-and-validation (V&V) strategy for a game application than for the nation’s air traffic control system.

**Paul D. Nielsen**

**IN THE NEAR TERM,** we need to blend the best features of the agile and disciplined approaches to software engineering. Both have contributed—but each might fail in the extreme. We’ve seen some attempts to balance these approaches, but I don’t believe we’ve quite got it yet.

We need to develop capabilities quickly—but we also need to deliver those capabilities with quality assurance and fewer vulnerabilities. Agile lets us get working software in front of users as quickly as possible and get valuable feedback. But this might not always fit well with safety- or life-critical applications—for example, a self-driving car or an implanted medical device.

Software engineering has also become more about the composition of existing functionality while adding some innovative features and not as much about creating all the functionality from scratch. I believe that good software architecture will enable this. At one extreme, agile might not pay enough attention to software architecture. At the other extreme, the more disciplined approaches might pay too much attention to architecture at the expense of moving forward. How do we find the right amount of architecture to keep our velocity and quality high—and allow for continued system evolution without costly refactoring?

**References**


PAUL D. NIELSEN is the director and CEO of Carnegie Mellon University’s Software Engineering Institute (SEI), a global leader in advancing software and cybersecurity to solve the nation’s toughest problems through focused research, development, and transition to the broad software engineering community. The SEI is a key innovator in areas central to US Department of Defense and civilian government operation in the cyberspace domain, including software architecture, software product lines, interoperability, the integration of software-intensive systems, network and system resilience, and the increasing overlap of software and systems engineering. The SEI also directly supports more than 50 US government entities in their efforts to efficiently and effectively acquire and sustain new software and systems.
increasing autonomy. It’s not as useful to argue over whether a system is autonomous as it is to understand that we’re seeing a general growth in autonomous functionality. We’re delegating more decision making to our software. There will be a spectrum of autonomy in our systems, and how much autonomy we let our systems exercise might be dynamically allocated on the basis of the situation or environment.

Autonomous-system development, architecture, and V&V will further stress our current software engineering practices. For example, as we develop systems that can learn in the field, how will we determine when they’re sufficiently ready for release and won’t go awry?

And yet, the advent of more-autonomous systems might also greatly help software engineers by providing new tools to address our current challenges. Development environments with more-autonomous features might help us quickly create high-quality, less vulnerable secure code.

We might be able to gather field data in volume and at high speed to improve code functionality and security much more efficiently. These tools might let software engineers concentrate more on high-end innovation and creativity—the part of our profession we most love.

I believe it’s an especially exciting time for software engineering and software engineers. The future is ours for the making!

**Kevin Fall**

**AN IMPORTANT TREND** in software engineering has been the desire to have more reusable code. Software development is more an exercise in composability than authorship. What we’ve seen recently as a result is the creation of more and more frameworks that let developers provide functionality more easily from reusable pieces—but generally with little validation and with an uncertain or unknown supply chain.

We also have technologies that can provide important software capabilities and help assure software quality. Examples include formal methods, system analysis, and machine learning for autonomy and control. But these technologies are difficult for the average software engineer to fully exploit.

The challenge is to get these technologies to work together effectively so that we can build bigger systems with confidence. How can we validate that systems built with these technologies will perform as intended? These problems will likely be addressed in the next five to 10 years.

Twenty years out, I see many challenges arising from how we’re going to effectively build software that works on advanced hardware. We already have quantum key distribution. Twenty years from now, if our software needs to work with quantum computers, for example, how will we reason about those software systems’ performance? How will you write software in a probabilistic system that will give you a range of answers to a calculation, not always a single “correct” one?

I think future software engineers will be able to use these technologies as an expanded set of tools in their toolbox. A developer would be able to pick these things up and maybe not understand them all in detail but have confidence that they’ll work. As I mentioned, software engineering is an exercise in composition, and software engineers will need to be able to import an artifact from who knows where and understand important attributes about it. The component itself might make a claim about its properties, but how does a developer know to trust it? (This dovetails nicely with security properties. People who work in malware are essentially trying to do the same analysis for a certain class of attributes on components.)

So will we have a meaningful software field 20 years from now (versus a collection of specializations that don’t share much)? Yes. The ability to compose radically different software components will be a common theme. The trick is whether mere mortals can actually do the programming, as the set of needed competencies continues to grow (and the competencies of the future might require disciplines that haven’t been invented yet). Maybe automata will do the programming for us, and the question will be whether we can specify the end result well enough.

KEVIN FALL is the deputy director and chief technology officer of the SEI, where he directs the R&D portfolio of the SEI’s technical programs in cybersecurity and software engineering. He’s also an adjunct professor at Carnegie Mellon University’s School of Computer Science.

Selected CS articles and columns are also available for free at http://ComputingNow.computer.org.