How Abundance Changes Software Engineering

Diomidis Spinellis

IN THE END, it seems it was all about numbers. Consider the marvelous behavior of a humble honeybee. It flies and navigates; it communicates and exhibits social behavior; it perceives shapes, colors, patterns, odors, and movements. Achieving these tasks with a computer has been challenging scientists for decades. Yet, over the past few years we’ve been conquering one tough problem after another. Technologies such as self-driving cars, automatic translation, speech recognition, and face tagging are entering the mainstream.

Although algorithmic innovation has played an important role, the key enabler has been raw processing power. A honeybee’s brain contains about a million neurons and a billion synapses, while modern CPUs contain a few billion transistors. Granted, a synapse is a lot more complex than a logic gate, and silicon gates switch much faster than neurons, but on a rough scale, you could argue that computers are now achieving some sort of parity with tiny biological brains. We see a similar abundance of resources in other areas of computing: gigabit networking, terabyte storage units, petabyte databases, immense cloud-based datacenters, millions of software components, and, for some fortunate companies, billions of users.

This abundance is changing the nature of software engineering. First, by reducing the cost of failure, abundance changes how we developers use computing technologies. Second, abundance changes our role by moving the focus from technology to management.

When Failure Is an Option

The rise of processing power is letting us adopt different ways to ensure a program’s correctness. So, compile-time type checking is giving way to languages with dynamic type systems, such as JavaScript and Python, even for use in production. Such systems are obviously wasteful; their inability (failure) to verify types up front means they might pay the cost of type checking every time they execute a statement. Yet their versatility often justifies their price. Similarly, although significant progress has been made in formally ensuring a program’s correctness, nowadays we often prefer to run thousands of unit and regression tests every time we...
Many users can also bring with them millions of service requests per minute. With such numbers of requests, detailed performance characterization and service provisioning are often difficult or unnecessary. Instead, we accept that servicing some requests will be slightly delayed, and we use these delays as a signaling mechanism for dynamic load balancing and the elastic provision of additional computing resources.

On the software side, we now have thousands of components just a mouse click away. From HTML parsing to QR code scanning and from cryptographic protocols to full-text search, it’s all there. This wealth of elements is letting us move from generic, elaborately designed frameworks to organic ecosystems in which our systems can gradually grow according to our specific needs.

From Technology to Management
The profusion of easily available software components and systems is also changing our focus as software developers: from specifying, designing, implementing, and contributing software components. This entails learning how to find and choose components on the basis of their quality and ability to meet our system’s requirements. We must also become skilled in keeping track of the selected components’ evolution in a way that keeps our systems secure, reliable, and maintainable. And, because ecosystems die when we do, we must contribute to the ecosystems we use. This means expanding our attention from managing our organization’s teams to successfully participating in ecosystems we use.

We often don’t think of our software’s users as a resource, but they are. Globalization, the distribution of software as a service, and network effects often endow our organizations with millions of users. When users are plentiful, we can sometimes do without detailed requirements analysis and instead experiment with various options through A/B testing. We simply divide our users into groups and try different versions of the software on them to decide which features to adopt and which failed ones to axe.

A lavish user base is also allowing us (or forcing us) to reduce an application’s feature set to the lowest common denominator. Our mandate is no longer to stuff an application with features to satisfy every one of its (in the past, few) users but to select carefully those features that will satisfy the majority in our large user group. This phenomenon is most pronounced in minimal but handy apps that run in widespread devices such as tablets.
in larger open source and commercial communities.

Moreover, the abundance of computing resources is changing our goals. Often we’re interested not in developing software that can run on constrained resources but in utilizing the available resources in the most productive, profitable, and innovative way. For example, in the 1980s the Lotus Corporation profited mightily by managing to cram a fully featured and blindingly responsive spreadsheet program into a 4.77-MHz IBM PC with 256 Kbytes of RAM. Nowadays, instead of struggling to shoehorn applications into constrained hardware, Wall...
Street’s technology darlings are coming up with ways to use the widespread broadband connectivity and vast cloud-based infrastructures to offer undreamed-of services. Often, our difficult task is no longer to design algorithms, data structures, and schemata but to manage immense datacenters and data stores.

Finally, the shift to cloud computing and the provision of services on a global scale is shifting our responsibility from working on software with clearly defined boundaries to managing planet-wide system deployments. Our objective isn’t so much to deliver quality software but to offer a correspondingly reliable, secure, efficient, and maintainable software-based service.

On the basis of what I’ve argued here, someone might think that all we need to do from now on is to sail on the tailwinds of abundance. However, this isn’t the whole story. To take it to an extreme, it’s like arguing that the moon landings were a matter of having high-energy combustion fuel. Yes, the feats we’ve seen in the past couple of years have been the result of increasing raw computing power. However, that power isn’t enough. And it doesn’t come alone, nor will it increase forever.

To realize the systems I’ve described, in both software and hardware, scores of very intelligent people have devoted their lives to delivering ingenious designs and implementations. Perhaps someday the singularity will arrive and machines will design machines. Until then, the enormous processing power we use requires similar brawn to create it. A neural network might perform marvels, but that requires raw processing power, infrastructure architecture, and operations, plus the design of the neural network itself, for which advances come from select groups around the world. The same applies, for example, to testing. We now have the power to test thousands of components and users, but if we don’t know what we’re doing, all tests are worthless.

So, in the end, was it all about numbers? Well, yes, but to get to the numbers we also need a lot of good old-fashioned engineering, algorithmic thinking, and sweat.

Correction
In “App Store 2.0: From Crowdsourced Information to Actionable Feedback in Mobile Ecosystems” (Mar./Apr. 2017, pp. 81–89), in the fifth line of the second column on p. 83, “see sia” should be “see Figure 2a.” IEEE Software regrets the error.