The Atlantic

Programmers: Stop Calling Yourselves Engineers

It undermines a long tradition of designing and building infrastructure in the public interest.

Andrew Brookes / Corbis

IAN BOGOST | NOV 5, 2015 | TECHNOLOGY

I'm commiserating with a friend who recently left the technology industry to return to entertainment. "I'm not a programmer," he begins, explaining some of the frustrations of his former workplace, before correcting himself, "—oh, engineer, in tech-bro speak. Though to me, engineers are people who build bridges and follow pretty rigid processes for a reason."

His indictment touches a nerve. In the Silicon Valley technology scene, it's common to use the bare term "engineer" to describe technical workers. Somehow, everybody who isn't in sales, marketing, or design became an engineer. "We're hiring engineers," read startup websites, which could mean anything from Javascript programmers to roboticists.

The term is probably a shortening of "software engineer," but its use betrays a secret: "Engineer" is an aspirational title in software development. Traditional engineers are regulated, certified, and subject to apprenticeship and continuing education. Engineering claims an explicit responsibility to public safety and reliability, even if it doesn't always deliver.

The title "engineer" is cheapened by the tech industry.

Recent years have seen prominent failures in software. Massive data breaches at Target, Home Depot, BlueCross BlueShield, Anthem, Harvard University, LastPass, and Ashley Madison only scratch the surface of the cybersecurity issues posed by today's computer systems. The Volkswagen diesel-emissions exploit was caused by a software failing, even if it seems to have been engineered, as it were, deliberately.

But these problems are just the most urgent and most memorable. Today's computer systems pose individual and communal dangers that we'd never accept in more concrete structures like bridges, skyscrapers, power plants, and missile-defense systems. Apple's iOS 9 update reportedly "bricked" certain phones, making them unusable. Services like Google Docs go down for mysterious reasons, leaving those whose work depends on them in a lurch. "Your password contains invalid characters," a popular tweet quotes from an anonymous website, before twisting the dagger, "No, your startup contains incompetent engineers."

These might seem like minor matters compared to the structural integrity of your office building or the security of our nation's nuclear-weapons arsenal. But then consider how often your late-model car fails to start inexplicably or your office elevator traps you inside its shaft. Computing has become infrastructure, but it doesn't work like infrastructure.

When it comes to skyscrapers and bridges and power plants and elevators and the like, engineering has been, and will continue to be, managed partly by professional standards, and partly by regulation around the expertise and duties of engineers. But fifty years' worth of attempts to turn software development into a legitimate engineering practice have failed.

Just as the heavy industry can greenwash to produce the appearance of environmental responsibility and the consumer industry can pinkwash to connect themselves to cause marketing, so the technology industry can "engineerwash"—leveraging the legacy of engineering in order to make their products and services appear to engender trust, competence, and service in the public interest.

* * *

By the 1960s, large national-defense systems were largely managed by computers. But the creation of such systems was a disaster—almost everything was delivered late, over budget, and with unnecessary complexity. Late in the decade, the NATO Science Committee sponsored two conferences dedicated to establishing an engineering approach to software creation. The 1968 conference report shows that the notion was still aspirational:

> The phrase "software engineering" was deliberately chosen as being provocative, in implying the need for software manufacture to be based on the types of theoretical foundations and practical disciplines, that are traditional in the established branches of engineering.

Commercial applications meant to service ordinary people, from inventory control to airline reservations to banking, needed to be reliable. Programming merely involved implementation.

Software-engineering trends came and went during the ensuing decades. Structured programming paradigms of the 1960s, meant to make software development more predictable and less risky, gave way to the object-oriented paradigm of the '80s and '90s, meant to make programming better mirror the business processes it facilitates.

Meanwhile, the overall challenges of software engineering became more familiar and more entrenched. A decade after his 1975 intervention *The Mythical Man-Month: Essays on Software Engineering*, Fred Brooks lamented that little had changed. In response, he proposed incremental development, or prototyping. Today's software development is iterative, and for good reason: Software wasn't ever really akin to manufacturing and construction, where changes were difficult or impossible after initial implementation.

Computing is turning engineering into a type of speculative finance rather than a calling.

But, software was never not akin to manufacturing and construction, either. Almost 50 years after the NATO Science Committee conferences, some of its participants' warnings still hold. "In the competitive rush to make available the latest techniques," the '68 report opines, "we strive to take great forward leaps across gulfs of unknown width and depth." The same sentiment still holds today.

So, what happened? The personal-computer revolution, for one. In the 1960s and '70s, computers were expensive and scarce. They were confined to research, in governmental, corporate, and industrial contexts. But with the rise of the microcomputer in the late 1970s, anyone could own, use, and program one.

This democratization of software development ignited the consumer and business-software revolution. But it also changed the stakes of software engineering. Developing Microsoft Excel or the back-office systems at American Airlines was hardly glamorous or fast-paced. A giant product like a spreadsheet or a reservation system was still something like a bridge or a building: It had to work right, especially since patches and revisions were expensive and required physical intervention. Such cases require an engineering approach, while trying one's hand at a program for upload to the local BBS (or the modern app store) does not.

The informality of software development accelerated even more with the rise of the web, starting in the mid '90s and continuing through today. As software services moved to websites, smartphones, and the Cloud, two things happened.

First, the pressure to get things right the first time around was relieved, because updates and changes could be applied centrally, as in the mainframe era. Over time, the ease of rapid repair became an excuse for rapid development, and Brooks-style prototyping mutated into the constant software updates we experience today. Facebook has wisely retired its one-time internal-development philosophy, "move fast and break things," but no business reliant on civil or structural engineering would ever have adopted such a motto in the first place.

And second, software became more isolated from the world, even as it became more predominant. Earlier computing systems were imbricated with other aspects of business, industry, government, and society. An automobile customer-management system has to integrate with dealers, suppliers, shippers, banks and lenders, regulators, legacy systems, and customers. But today's software mostly stands alone. Instagram, a photo-sharing service valued at $35 billion last year, just uploads and downloads images between its servers and its app.

To be sure, today's Cloud-connected tools still rely on infrastructures, especially the physical servers and networks that handle millions of users accessing billions of files. But those activities have largely been outsourced to infrastructure giants.

Likewise, integrations with messaging, financials, and storage have been abstracted such that individual software developers can treat them as black boxes. That sometimes allows software to run better and more reliably, but it also allows developers to avoid interfacing with the messy world outside their co-working spaces.

As a result, software development has become institutionally hermetic. And that's the opposite of what "engineering" ought to mean: a collaboration with the world, rather than a separate domain bent on overtaking it.

* * *

The traditional disciplines of engineering—civil, mechanical, aerospace, chemical, electrical, environmental—are civic professions as much as technical ones. Engineers orchestrate the erection of bridges and buildings; they design vehicles and heavy machinery; they invent and realize the energy systems that drive this equipment; and they contrive methods for connecting all of these systems together.

It's no accident that the most truly engineered of software-engineering projects extend well beyond the computer. Autonomous-vehicle design offers the most obvious contemporary example. When Google designs self-driving cars, it musters its own computational systems, like mapping and navigation. But it also integrates those into a world much larger than browsers and smartphones and data centers. Autonomous vehicles share the roads with human-driven cars, pedestrians, and bicyclists. Those roads are managed, maintained, and regulated. Self-driving cars also interface with federal motor-vehicle standards and regulations, along with all the other material demands and foibles of a machine made of metal and plastic and rubber rather than bits. Engineering addresses complex, large-scale systems.

This is why it is so infuriating when Uber insists that it is just a technology platform, and thus not subject to the oversight of transportation-services regulation. Love or hate it, Uber is not just an app developer—it's a car-service network activated by software, and thus subject to public interest and oversight. And no matter what Uber says, the company still advertises careers in "engineering, design, and product" categories on its website. Engineering roles are illustrated by a bearded guy staring at source code on two monitors.

Uber Screenshot

Other engineering disciplines are subject to certification and licensure. If you've ever hired a civil, structural, or hydraulic engineer for a construction or repair project, that individual probably had to be certified as a Professional Engineer (PE). Licensing processes vary by state, but Professional Engineers generally need to hold a 4-year degree from an accredited program in their discipline, pass one or more exams, and possess 4 or more years of professional experience under the supervision of a licensed engineer. Not all working engineers are or need to be Professional Engineers, but to open an engineering consulting practice or to claim that one is an "engineer" in a formal context, licensure is usually required. It's in the state's interest to ensure that someone claiming to be an engineer (or an architect a surveyor or a cosmetologist or a massage therapist) isn't just making up his or her qualifications.

Professional Engineering certification is usually offered only in fields where something could go terribly, horribly wrong with unqualified actors at the helm. California, for example, issues Professional licenses for agricultural, chemical, civil, control system, electrical, fire protection, industrial, mechanical, metallurgical, nuclear, petroleum engineering, and traffic engineers.

In 2013, the National Council of Examiners of Engineers and Surveyors (NCEES), which all 50 states use for licensure examination, began offering testing for software engineers. The exams were produced in collaboration with the IEEE, who maintains a Software Engineering Body of Knowledge (SWEBOK).

But it's unlikely that Silicon Valley workers would pursue such a license. For one, software engineers are unlikely to open a private office like a structural engineer might do. Even if all engineers are supposed to work under a licensed engineer to use the name, at big companies, many do so under layers of management.

The information-technology industry simply doesn't value certification as much as engineering does, or even as much as IT once did. Silicon Valley bigwigs like Peter Thiel have been flouting formal degrees for years, and even big companies like Google have indicated that they don't value a college degree over and above the ability to do whatever work Google decides is important.

But by definition, "engineering" has traditionally entailed the completion of an Accreditation Board for Engineering and Technology (ABET)-approved 4-year degree. ABET's accreditation requirements for computer science are vague, but they do expect "an ability to apply design and development principles in the construction of software systems of varying complexity."

Accredited computer-science programs might be moving further away from software engineering anyway. Agile software development has become predominant, focused on rapid iteration rather than long-term planning and

intricate documentation. One popular agile method is Scrum, which is focused on short "sprints" toward a series of changing goals.

Engineers bear a burden to the public, and their specific expertise emanates from that responsibility.

Lightweight approaches like Scrum are more compatible with the fast-moving marketplace of computer technology. An app or a web service isn't a bridge or a building. Software is temporary, and provisionalism is considered a feature, not a bug. But at the same time, the stakes of software development are becoming akin to that of bridges and buildings. Not only do computers run our cars and airplanes and medical devices, but also our banking systems, health-care organizations, insurance-underwriting practices, telephony and communication networks—even our social and entertainment activities. And even if successful, methodologies like Scrum never allow that infrastructure to stabilize. Some new tweak can be made, some new feature can always be added.

Meanwhile, start-up culture is changing engineering education anyway. Entrepreneurship is exalted. Accelerators and incubators abound. Not all students in computer-science programs think they'll become startup billionaires… But not all of them don't think so, either. Would-be "engineers" are encouraged to think of every project as a potential business ready to scale and sell, rather than as a process of long-term training in disciplines where concerns for social welfare become paramount. Engineering has always been a well-paid profession, but computing is turning it into a type of speculative finance rather than a calling.

* * *

The U.S. Bureau of Labor and Statistics (BLS) calls the "engineers" who work at Google and Uber and Facebook and its ilk "Computer Programmers" or "Software Developers." The former write code, the latter design systems. Nobody has to follow the BLS's definitions, and you can understand why more grandiose titles would be appealing to Silicon Valley disruptors. "Engineer" conjures the image of the hard-hat-topped designer-builder, carefully crafting tomorrow. But such an aspiration is rarely realized by computing. The respectability of engineering, a feature built over many decades of closely controlled, education- and apprenticeship-oriented certification, becomes reinterpreted as a fast-and-loose commitment to craftwork as business.

Engineerwashing entails a shift from the noun to the verbal sense of "engineer." An engineer is a professional who designs, builds, and maintains systems. But to engineer means skillfully, artfully, or even deviously contriving an outcome. To engineer is to jury-rig, to get something working more or less, for a time. Sufficiently enough that it serves an immediately obvious purpose, but without concern or perhaps even awareness of its longevity. Engineering in this sense embodies MacGyver scrappiness, a doggedness compatible with today's values of innovative disruption. But then, no reasonable person would want MacGyver building their bridges or buildings. Or software!

Perhaps software calamities like data breaches and dieselgate will raise the hackles of the public, such that the standards for software development will be revealed and, in time, reformed. But given the difficulty of renewing,

creating, or enforcing more urgent regulatory oversight over the technology industry, the job titles of technology workers probably seems like a minor matter.

Just as Silicon Valley has deftly reframed its business interests as a process of "changing the world," so it has also reframed engineering as a process of building something temporary. After all, professionals like graphic designers and hedge-fund managers also build things, but we don't normally call them engineers (brand engineering? speculation engineering?). They do work that might or might not be infrastructural, and that might or might not be conducted in the public interest. And those latter matters are what separate engineering from mere business or craft.

All of which leads us back to the bridges to which my friend negatively compared his "engineer" colleagues. In Canada, many civil engineers wear an iron ring symbolizing the ethical commitment their profession undertakes. The ring is proffered in a ceremony called the Ritual of the Calling of an Engineer, in which an oath penned by Rudyard Kipling is recited. It reads, in part, "My Time I will not refuse; my Thought I will not grudge; my Care I will not deny toward the honour, use, stability and perfection of any works to which I may be called to set my hand." (The U.S. Order of the Engineer offers a similar but less poetic rendition of the oath and the ring.)

A persistent legend holds that the rings are forged from steel reclaimed from the Quebec Bridge, which collapsed catastrophically upon construction in 1907, killing dozens of workers. Though false, the myth still holds up allegorically. Even if it doesn't embody it, the Iron Ring's steel represents the Quebec Bridge, and every other. Engineers bear a burden to the public, and their specific expertise as designers and builders of bridges or buildings—or software—emanates from that responsibility. Only after answering this calling does an engineer build anything, whether bridges or buildings or software.

ABOUT THE AUTHOR

IAN BOGOST is a contributing editor at *The Atlantic*. He is the Ivan Allen College Distinguished Chair in media studies and a professor of interactive computing at the Georgia Institute of Technology. His latest book is *Play Anything*.

Twitter