Editor: **Grady Booch**
IBM, grady@
computingthehumanexperience.com

Then a Miracle Occurs

Grady Booch

DEVELOPING a software-intensive system is like raising a child.

As Will Pleasant-Ryan observed, both babies and code have smells, you get used to changing requirements, you have to take the long view, you can't do it alone, and eventually you have to let go.[1] I would add to Will's most excellent simile that, although you might think you know everything up front, you certainly don't, and therefore you end up discovering a lot of things along the way. Furthermore, in this age of cognitive systems, you also have to both teach them and give them the skills to learn on their own. Finally—as they all probably should anyway—they sometimes turn out very different from what you first expected.

Developing a software-intensive system is like producing a movie.

Tony DeRose of Pixar had much to say about this simile,[2] as did the good folks in the early agile community.[3] Both software development and filmmaking require the skills of a diverse set of stakeholders who must work together intensely for short bursts of time. Talent matters, and as such you must often deal with strong personalities. Although the bones of the delivered artifacts might be clear, opportunities exist for creatively pivoting along the way to tell the right story at the right time in the right way. If you're really successful, you'll be asked to make a sequel, and you might even turn a small indie project into a large money-making franchise.

Developing a software-intensive system is like releasing a new drug.

Pedram Alaedini and his colleagues observed that software development and drug development are "highly complex, rely on multidisciplinary teams, and usually come in late and over budget. In addition, they are often judged based on the same quality factors: reliability, safety, and efficiency."[4]

I would add that in both cases, the stakes are high: your efforts might improve the lives of millions, but you might also cause some deaths along the way.

Developing a software-intensive system is like writing a novel.

J.J. Merelo gave us a riff on this comparison.[5] Both tasks are creative endeavors. Both require collaboration. Both must "find their own voice, but at the same time rely on the tropes of the genre and on plot lines that have been there for millennia" (in the world of software, we call these *patterns*). Both must appeal to their particular audience. Both are full of surprises and contain many little languages. And while they must adapt themselves to the medium, their very presence changes the medium itself.

Developing a software-intensive system is like dating.

The delightful book *Dating Design Patterns*[6] explored this rich

0740-7459/15/$31.00 © 2015 IEEE

simile. Surprise Statefulness, Interested Listener, Decorated Visitor, and Fifth Position Break are all patterns found in dating and well-structured software.

Developing software is like making a work of art.

Jonathan Wallace provided some interesting insight on the matter.[7] Art springs from the mind of the creator, as does software. Both might be rendered on a literal blank page or might have a patron who dictates certain characteristics. Most art (and software) is bound by the materials at hand, but great art (and great software) creates and defines its own medium. Art builds on the fundamental physics of light, the foundations of color theory, the chemistry of materials, and the cognitive and emotional nature of the human mind. Software builds on logic, mathematical theory, the physics of hardware, and human needs. Both art and software might themselves be works of beauty.

Developing a software-intensive system is like the travels of Lewis and Clark.

If you stay close to home, you might be quite content walking the same path every day. This is the bread and butter of most life and most software. In well-understood domains, the journey is predictable, the risks are known, and the outcomes are generally satisfying.

However, if some hint of exotic lands captures you, you must find a new way. If the rewards are high but the path unmapped and particularly dangerous, rarely will you go on your own. Others must accompany you, especially those with the same spirit of adventure. Although you can equip yourself with the best knowledge and the best equipment, you know that along the way you'll have to rely on your own instincts. Sometimes—quite often, actually—you'll try one direction but have to backtrack and try another. You might lose some people along the way. You might even fail and never make your destination. But if you do succeed, you'll reap the rewards of being the first, and others will soon follow you. If you fail but the journey is still interesting, then you'll have succeeded in showing others part of the way.

If some hint of exotic lands captures you, you must find a new way.

Developing software is like teaching cats to walk in a line.

It's not easy: cats and developers are often quite independent creatures that want to do things their own way, on their own time, with minimal ceremony. But if you offer the right incentives (a nice piece of fish for the cats, the vision of making something Really Cool for the developers), then you'll find them collaborating beyond reason.

Developing a software-intensive system is like building with Legos.

I loved building with Legos as a kid. Back then, you'd get just a box of random blocks, and you'd use your imagination to build whatever came to mind. These days, most Legos come in a kit, often with special parts but always with detailed instructions. On one hand, having standard parts is nice: you don't have to reinvent everything, and you can always fall back on common ways to build certain things. On the other hand, sometimes you have to wedge existing components in unnatural ways to get them to do what you want. The results might be clumsy, but mostly they'll be good enough.

Developing a software-intensive system is like building with Legos and having your own 3D printer to make custom blocks.

The same rules I just mentioned apply, but this time you can craft your own parts. Most will be one-offs. But, if they prove particularly useful, you might find yourself replicating the custom component over and over, to the point where others want to use it.

Developing a software-intensive system is like building a dog house.

Here, the Nike development process is sufficient: Just do it. If the results are good enough, you'll end up with a dry, happy dog. If the results are a failure, you can just tear down the old doghouse and start from scratch or break the old one apart and put it back together in a new way. Failing that, you can always get a new, more accepting dog.

Developing a software-intensive system is like a city growing old.

Your city probably started generations ago as a modest outpost along some interesting terrain. It grew because it was a useful place—economically viable, rich with

natural or human resources—and a place that attracted others to live. Time passes, as it tends to do, and suddenly you find yourself living in a 100-mile city badly in need of new things to keep it viable: a new transit system, an upgraded sewer system, and special services for a changing population. Problem is, no matter what you do you can't just start over (too much legacy), and you can't just apply the Nike development process (too many stakeholders with concerns far more diverse than you could ever imagine). Change takes time, money, and patience. However, if you fail to change, your city will die. Guaranteed.

Developing quality software-intensive systems that matter and systems that endure and make a difference is all of these things.

It's also none of them.

As I reflect back on the history of software engineering, some clear patterns emerge. In the earliest days—the dawn of digital computing in the '30s and '40s—we'd see small teams of systems people (back then, software was so very mixed up with hardware). Personal processes tended to dominate. From the '50s through the '70s, the programming priesthood arose, giving us the dogma of structured methods and waterfall processes. Don't get me wrong: I'm not using these terms in disparaging, emotional ways. They were quite reasonable in their time, given the nature of the software-intensive systems we built. In the '80s, with the rise of personal computing, we saw a split: personal processes were reinvented for these personal things, and larger teams grew (with outsourcing as a new feature) at the other end. By the '90s, the presence of the Internet and the rise of object-oriented languages changed everything, yielding incremental and iterative methods—the precursor to today's agile methods.

From the outside, from the public's viewpoint, developing software-intensive systems looks either very simple or incredibly mysterious. Given the recent emphasis on teaching programming to every K–12 student, you might be left with the impression that everything is just a Simple Matter of Programming. Of course, computing insiders know that this simply isn't true. Similarly—especially to the public fed on a diet of television and movies with incendiary topics, dubious science, and questionable timelines—it would seem that all software happens when you put bright people in a room, give them a lot of venture capital, and then wait until a miracle occurs.

Developing a software-intensive system isn't like that at all, and we, as insiders, have a responsibility to tell the stories of computing in an engaging, truthful way.

One parting thought. The field of software engineering might seem to be largely settled: we think we know the essentials, and everything that might follow is simply a variation on some agility-at-scale best practices.

If you believe that, you are so very wrong.

The future will bring us systems of breathtaking complexity and importance. The advent of machine learning changes how we grow systems because we must not only build the software but also train the system. The evolution of quantum computing and neuromorphic computing challenges the very way we program our devices.

Indeed, we've just begun. Software is the invisible writing that whispers the stories of possibility to our hardware, yielding computing systems of exquisite complexity and promise.

And we, as insiders to computing, have the privilege of making the possible manifest.

References

1. W. Pleasant-Ryan, "Six Ways Developing Software Is Like Being a Parent," blog, 6 Sept. 2014; http://spin.atomicobject.com/2014/09/06/development-and-parenting.
2. T. DeRose, "The Connection between Movie Development and Software Development," *Proc. 2013 Int'l Conf. Software Eng.* (ICSE 13), 2013; http://dl.acm.org/citation.cfm?id=2486901.
3. "SoftwareAsFilmMaking," 2012; http://c2.com/cgi/wiki?SoftwareAsFilmMaking.
4. P. Alaedini, B. Ozbas, and F. Akdemir, "Agile Drug Development: Lessons from the Software Industry," *Contract Pharma*, 14 Oct. 2014; www.contractpharma.com/issues/2014-10-01/view_features/agile-drug-development-lessons-from-the-software-industry.
5. J.J. Merelo, "Five Reasons Why Writing Is So Much Like Software Develoment," blog, 30 May 2013; https://medium.com/i-m-h-o/five-reasons-why-writing-is-so-much-like-software-development-6d154a43719c.
6. E. Gordon et al., *Dating Design Patterns: Elements of Reusable Object Oriented Paired Programming*, Solveig-Haugland, 2003.
7. J. Wallace, "Is Software Art or Engineering?," *The Ethical Spectacle*, Nov. 1999; www.spectacle.org/1199/software.html.

GRADY BOOCH is an IBM Fellow and one of UML's original authors. He's currently developing *Computing: The Human Experience*, a major transmedia project for public broadcast. Contact him at grady@computingthehumanexperience.com.

See www.computer.org/software-multimedia for multimedia content related to this article.