Editor: George Fairbanks
gf@georgefairbanks.com

Why Is It Getting Harder to Apply Software Architecture?

George Fairbanks

TODAY, WE FIND ourselves in a surprising situation: we understand software architecture fairly well, but we find it difficult to put it into practice. This is because most software developers work within iterative processes that focus attention on what's new, and because factory-inspired metaphors encourage ever-quicker movement from feature requests to deployed code. How did that happen?

Around the year 2000, we changed our philosophy of software development in two ways.

A big idea from Bertrand Russell's *A History of Western Philosophy* is that "[T]he circumstances of men's lives do much to determine their philosophy, but, conversely, their philosophy does much to determine their circumstances."^{1} Around the year 2000, we changed our philosophy of software development in two ways. Those two changes, in turn, led to the circumstances that we see today.

Digital Object Identifier 10.1109/MS.2021.3071520
Date of current version: 18 June 2021

Two Philosophy Changes

The first change in philosophy was in the object-oriented community's attention, from design to management. In the 1990s, the discussion was about how to design object-oriented systems. Thought leaders prescribed engineering activities, like choosing objects, modeling the problem domain, expressing it with a modeling notation, allocating responsibilities, and describing contractual behavior including preconditions and postconditions. Patterns were the means of communicating how to design.

Around 2000, thought leaders shifted their attention to management activities: organizing meetings, interacting with the business, seating arrangements, and, most importantly, the use of iterations. Despite broad agreement that iterative processes worked better than waterfall processes, most teams in the 1990s were still using waterfall. My inference is that agile advocates found it hard to apply their design ideas under these circumstances, so they set out to create circumstances suitable for object-oriented design and programming.

The second change in philosophy was the popularization of factory-style production metaphors, including: automation, minimization of work in progress, and reducing cycle times. These ideas found a natural home in the emerging practice of DevOps. The traditional split between software developers, testers, and system operators was joined into a single role. This created the desired circumstances: a developer with this combined role had the perspective and ability to identify inefficiencies, automate them, and seek continuous improvements.

These two changes in philosophy led to the intended changes in circumstances, but also other changes. Today, developers find it easier to focus on incremental changes instead of the system as a whole, and they seek improvement by reducing the time between a feature request and its moving to production. These circumstances make it easy to pile up technical debt, despite warnings and advice. The rest of this article shows how these circumstances arose: through changes in the dominant decomposition, reinforcement by tooling, overloading of developers, and the inevitable dominance of quantifiable metrics over intangible design concepts.

Dominant Decomposition

Engineering systems typically have a single *dominant decomposition*. Consider libraries, for example. Some libraries organize shelves by the size of the books, to optimize for scarce space. Other libraries organize shelves by the book topic, to make browsing easier. A library has a choice—organize by size or by topic—and that choice sets the dominant decomposition of the library.

Software processes also have a dominant decomposition. Consider waterfall and iterative processes. A waterfall process focuses on the *system as a whole*. In phases, developers collect requirements for the whole system, analyze the whole system, design the whole system, and so forth until the whole system is finished. In the early phases of a waterfall, there is no code yet, so developers cannot yet be focusing on a series of code patches. In this way, waterfall processes require holistic thinking.

Typically, an iterative process focuses on *what's new* in the current iteration: the new features, user stories, and code patches. Developers using an iterative process do the same kinds of activities as in waterfall—analysis, design, and implementation—but focus their attention on what's new, not on what already exists. In particular, the code patches they make are largely additions to the existing code, not a rewrite of the whole system.

Tools Reinforce the Decomposition

In the 1990s, tools encouraged thinking about the system as a whole. If software development were exaggerated in a movie, viewers would see a machine in a big room with developers in lab coats walking up and changing it. Pessimistic version control systems were common in the 1990s, meaning that opening a source code file for editing would also lock it, preventing others from editing it at the same time. This reinforced the idea that you were directly editing "the single system," the one held in the central version control system.

In contrast, today it's common to create patches against a locally held copy of the system's code, sending those patches to be reviewed by your team, then push those patches into the main version control repository. You recognize that your local copy of the source code inevitably falls behind the main repository, and the lifecycle of a patch includes not just authoring but also peer review of that patch, catching the patch up to the current code, and subsequent revisions.

Typically, an iterative process focuses on *what's new* in the current iteration: the new features, user stories, and code patches.

Big open source projects like the Linux kernel were early adopters of the patch-focused view, and chose version control systems that made it possible. Linus Torvalds invented the Git version control system in 2005. Git became popular with developers besides the Linux kernel hackers, and they adopted its patch-focused view. Teams can and do think about the system as a whole, but their tools direct attention on the series of patches.

Juggling Two Decompositions

Today, developers must keep the dominant decomposition in mind (the new features expressed as a stream of patches) as well as the secondary decomposition (the system as a whole). That is not easy, however, and is a skill that must be learned. Let me tell a quick story that I think illustrates the challenge.

When I was in college, a bunch of us would play pool. None of us were good, but we improved over time. At first, my attention was on hitting that first ball into a pocket. After a while, I was okay at that, and I noticed that to improve, I needed to leave the cue ball in a good place for a second shot. As I tried to do that, I found myself making more mistakes on my first shots. Eventually, I got good enough that my first-shot performance recovered, and I could be ready for a second shot.

Let's call hitting the first ball as the dominant decomposition, because if you cannot do that, nothing else matters. But you won't be successful if you can't also set up the second shot. When we're still learning, we can barely do the first thing, then we stumble as we try to balance multiple concerns, then with enough practice we can do both.

Software developers go through this progression, too. At first, they struggle to implement any feature, then they struggle to implement features while also balancing technical debt (doing both poorly), then with enough practice they can do both well. Most teams have developers at each skill level. From this viewpoint, a feature-focused iterative process asks quite a bit from developers.

Factory Metaphors

In the 1990s, most systems had scheduled downtime, which was used to push new code into production. Deployments were not automated, and I had to walk over to a teammate's desk to ask what code was in production, or what updates had been applied to the database.

Git became popular with developers besides the Linux kernel hackers, and they adopted its patch-focused view.

On some teams, we deployed code so infrequently, just a few times per year, that everyone knew what code was deployed.

Today, these practices are rare and production systems run nonstop. How do developers make changes to a system with no scheduled downtime? They rely on the patch as an atomic unit and follow an intricate dance with feature toggles and a sequence of patches. How do they move their code to production so quickly? Through automation: they write code that compiles, tests, and deploys the code, perhaps even monitoring it for trouble and rolling back to a working version.

I've always been a fan of automating development processes, especially involving testing and moving code to production. These go hand-in-hand with a factory metaphor. Automation has led to incremental improvements, year over year, just like you'd expect in factories.

A big driver of those improvements is faster cycle times. Modern factories pride themselves on avoiding stockpiles of work in progress, and on how quickly they can transform their raw materials into products. In software development, teams pride themselves on how short their iterations are, and how quickly a feature is deployed into production. This produces an ever-better machine but perhaps at the cost of the code becoming a poor partner in thought.

Code as a Machine, and as a Thought

In an earlier column, I wrote about code's dual nature as a machine and as a thought.[2] It's possible to write code that works perfectly well as a machine, yet it is an imperfect carrier of our thoughts. One way to do that would be to replace all of the variable names with meaningless identifiers, like *x* and *y*. The code would continue to work equally well as a machine, but less well as a carrier of our thoughts. Code can also fail as a thought because it reveals obsolete ideas. Technical debt occurs when our thoughts move forward, yet the code we wrote yesterday still expresses our former thoughts.

When developers are designing a system, they form their thoughts about how to solve a problem, then they write code that matches those thoughts. Quick cycles give us quick feedback about which ideas don't work out in practice, which makes design easier. On balance, though, it's hard to design within quick cycles. There are factors to weigh, alternatives to generate, and implications to reason through. Factories do the same thing over and over, but each design problem is unique, even when it is similar to previous ones.

What's worse, on today's projects, the other developers on the team will be evolving the system. They won't be trying to hide the changes from you, exactly, but their communication will be imperfect. On small projects, you could perhaps read every change to the code and reason through its significance in the overall design, but that's a hefty burden. As cycles become shorter, and the system becomes larger, it's harder to keep up with all of the changes, to the point where developers may stop trying, and instead keep their focus limited.

A Healthy Balance

There are two parts to software development: creating a design and expressing it as code. The code is tangible but the design is conceptual. Keeping a project healthy means doing both well. Here's my concern: whenever you mix the conceptual with the tangible, it's easier to neglect the conceptual. When you miss a tangible target, it's obvious, but when you miss a conceptual target, you might not recognize it, or might rationalize that, because it's impossible to measure, you were really quite close.

Blindly applying a factory process to software development will drive improvements to the tangible part (the code) at the expense of the conceptual part (the design). We see plenty of examples of this today, where teams have great feature velocity at first, are puzzled when velocity slows, and eventually the project is abandoned. As Cunningham warned, if we bolt features onto an existing codebase without consolidating those ideas into the code, the design will suffer, and over time "[e]ntire engineering organizations can be brought to a standstill under the debt load of an unconsolidated implementation."[3]

This challenge exists in any process, but it's worse when the dominant decomposition is the feature. For all its faults, the waterfall process forced us to think holistically about the design. But an iterative process can work just fine. Plenty of teams keep their designs healthy within iterative processes. Those that succeed, I think, are finding ways to keep thinking holistically about the system. If you worry about the health of your system's design, ask how your process guides developers to think holistically, and if developers are rewarded for doing so.

One additional point here, and it's a bit of a forward reference because I intend to write more about Peter Naur's ideas on theory building in the future. He says:[4]

> *[P]rogramming properly should be regarded as an activity by which the programmers form or achieve a certain kind of insight, a theory, of the matters at hand. This suggestion is in contrast to what appears to be a more common notion, that programming should be regarded as a production of a program and certain other texts.*

Design and architecture are part of what Naur calls a theory. If we follow Naur and regard programming as the forming of a theory (the conceptual part), then it's dangerous when we tailor our processes toward the production of the program (the tangible part). Instead, processes should guide us to the neglected activities that deserve attention. Naur describes his experience watching developers who misunderstood the theory of a program make poor choices when implementing features, choices that degraded the design.

New Circumstances, New Philosophies

Bertrand Russell observed that our circumstances determine our philosophy, and our philosophy determines our circumstances. In the past few decades, we changed our philosophy to embrace iterative processes and factory metaphors. As a result, today it's easier to build typical applications, get them to production without drama, and keep them running 24/7.

By the late 1990s, we understood software architecture pretty well, and it was poised to become a standard part of software development. Around the same time, the software development world became inhospitable to the holistic thinking that characterizes architecture.

Has the time come for software architecture? Perhaps. Today, because of the circumstances, many teams say technical debt is their primary challenge. We should expect these changed circumstances to lead to a changed philosophy. I'm sure the new philosophy will not be waterfall processes and manual deployments. Instead, I think we will find ways to focus on the system's overall design as our primary concern, with each new feature a secondary, but still critical, concern. Under these circumstances, developers will manage technical debt better, and it will be easy and natural to apply architecture ideas in everyday practice. Of course, that will again change our circumstances, but my crystal ball is too hazy to see what happens next.

ABOUT THE AUTHOR

GEORGE FAIRBANKS is a software engineer at Google, USA. Contact him at gf@georgefairbanks.com.

References

1. B. Russel, *A History of Western Philosophy*. New York: Simon and Schuster, 1967.
2. G. Fairbanks, "Code is your partner in thought," *IEEE Softw.*, vol. 37, no. 5, pp. 109–112, Sept./Oct. 2020. doi: 10.1109/MS.2020.3000084.
3. W. Cunningham, "The WyCash portfolio management system," in *Proc. Addendum Object-Oriented Program. Syst., Languages, Appl. (OOPSLA 92)*, Vancouver, Canada, Oct. 5–10, 1992. doi: 10.1145/157709.157715.
4. P. Naur, "Programming as theory building," *Microprocess. Microprogram.*, vol. 15, no. 5, pp. 253–261, May 1985. doi: 10.1016/0165-6074(85)90032-8.