Modular Architectures Make You Agile in the Long Run

Dan Sturtevant

From the Editor

Gene Kim once told me that organizations that require a developer to take 10 people out to lunch to get an API change done appear to have lower IT performance. He and I hypothesized that an overly high "lunch factor" would impede DevOps transformations, and added some questions on this to the *2017 State of DevOps Report* to learn more about the role of architecture in DevOps. The conclusion in the report reads, "Loosely coupled architectures and teams are the strongest predictor of continuous delivery." My colleagues Alan MacCormack, Carliss Baldwin, and Dan Sturtevant at the Harvard Business School have devised a way to measure and visualize architecture quality and its "cost of ownership" consequences that goes far beyond the "lunch factor." —*Mik Kersten*

THE *2017 STATE of DevOps Report* noted that loosely coupled architectures spur team performance by making it "easy to modify or replace any individual component or service without making corresponding changes to [those] that depend on it."[1] Put another way, systems with degraded modularity are incredibly difficult to change because pulling one thread always seems to lead to another headache. Engineers give up after months of fruitless investigation and failed changes. When architectural complexity proliferates, systems are no longer understandable. Teams can't communicate about them, learning curves grow, morale plummets, and staff turnover increases.

Over the past 15 years, our research team, led by Carliss Baldwin and Alan MacCormack at the Harvard Business School, have devised methods for measuring modularity and its erosion. We scanned thousands of code bases and found architectural flaws in many. We investigated how architecture degradation impacts business outcomes. This included studies of defects and safety,[2] developer productivity and development staff turnover,[3] vulnerability,[4] the ability to drive new revenue,[5] and de facto vendor lock-in.[6] Our team also recently founded Silverthread Inc. Over the past three years, Silverthread has helped more than 75 commercial and US federal government customers gain visibility, quantify the cost of ownership and risk, and regain control of their development projects.

These experiences have convinced us that that long-term agility is possible only if you're employing an agile product architecture. Not everyone shares this view. In the past
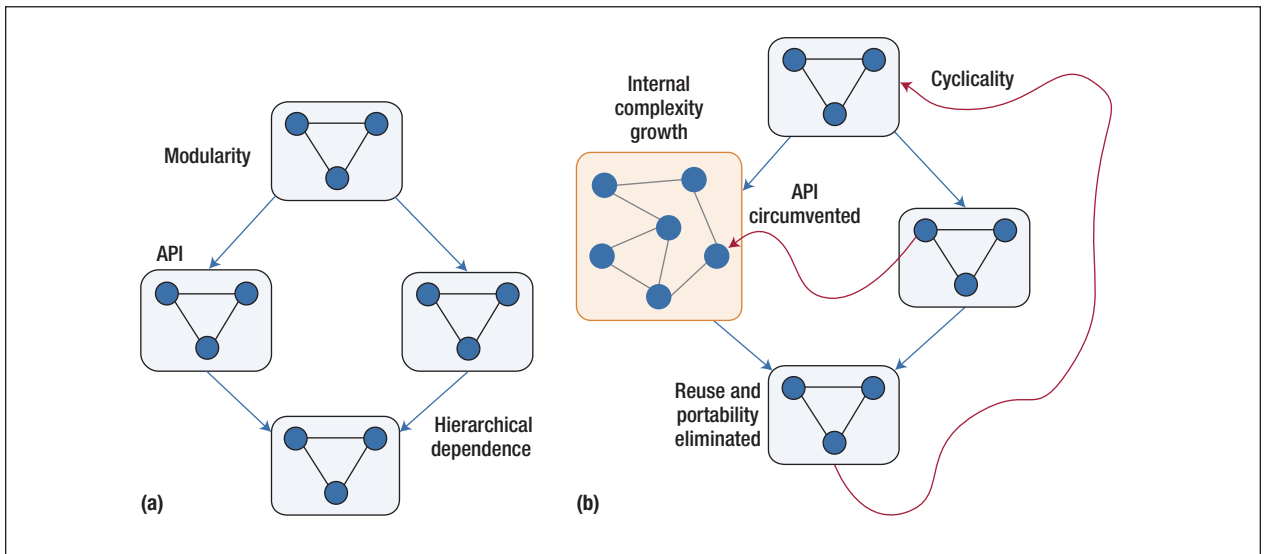
FIGURE 1. Architectural health. (a) This healthy architecture is structured into modules, each of which is manageable by a small team of people with IQs of 100. (b) In this unhealthy architecture, it's hard to understand what's going on and reliably anticipate side effects of changes.

decade, agile practitioners have focused intensely on improving software development processes and not so much on technical health. We've worked with several large organizations in which the application of lean principles produced underwhelming results. This is because velocity measurement, planning poker, attacking defect backlogs, Kanban cards, pair programming, or sprint-based planning does little to attack the root cause of problems that are inherently structural. If humans can't easily understand or modify their code, teams might be using the best agile practices, but their ability to respond to market demands will be far from agile. Technical debt will weigh down the value stream's performance.

Here, I introduce sophisticated ways to visualize and quantify software modularity and its erosion. These techniques can help you identify architectural-complexity hotspots. I highlight research that has measured economic impact and helped organizations improve

architectural outcomes. And I discuss how to use quantitative software design analysis to help teams decide if and when to refactor or rewrite software, using objective financial models of the likely return on investment (ROI).

Principles of Architectural Health

Today's enterprise software systems are so large that no one person can understand how everything works. There are no laws of physics to constrain solutions. Software systems are complex networks of unbounded abstractions with huge numbers of interconnections.

So, designers use well-accepted architectural principles to gain control of this complexity. These principles allow decomposition of a complex system into more understandable chunks so that teams of cognitively bounded humans can work on different parts as independently as is practical. They also ensure that changes don't propagate in

unintended ways, creating defects and rework and slowing down future adaptation.

We've learned to ask three key questions when assessing architectural health. Is the system modular or loosely coupled in terms of the connections between components? Are the components organized hierarchically—with a clear sense of those at the top, middle, and bottom? Are the components well designed and not unnecessarily complex?

Imagine a system so complex that a human would need an IQ of 400 to understand it completely. To cope with this complexity, we adopt a *healthy architecture* (see Figure 1a), which is structured into modules, each managed by a small team of people with IQs of 100. Each module has a simple API that hides its internal complexity, letting other teams use a simplified mental model. The code base is laid out hierarchically so that interteam relationships are clear and evolution is manageable.

In the *unhealthy architecture* in Figure 1b, one module has grown too big, requiring an IQ of 200 to understand what's going on and reliably anticipate side-effects of changes. In addition, APIs have been circumvented, exposing external teams to this growing internal complexity. Finally, cyclicality has been introduced, destroying the orderly hierarchy and creating complex communication and coordination requirements.

In software systems, entropy naturally turns healthy designs into unhealthy designs, unless proactive measures are in place to stop this evolution. Even the most well-designed systems erode over time. This slowly and imperceptibly increases the cognitive demands on developers, requiring everyone to increasingly rely on incomplete mental models. These breakdowns increase the potential for changes to propagate across the system, compounding errors, generating unpredictable behaviors, and creating tension across the organization. Because it's unrealistic to staff teams with people with IQs of 400 to anticipate such issues, the problems perpetuate, and the architecture continues degrading.

Figure 1a is similar to the whiteboard drawings engineers use every day. Unfortunately, over 80 percent of the code bases we scan look more like Figure 1b. Idealized pictures reflect flawed mental models. Developers don't see the hidden structure responsible for unanticipated side effects, frozen code, deadlocked organizations, and premature obsolescence.

Detecting Architectural-Health Problems

Unmapped, unpredictable linkages between software components are the root cause of macro-level complexity. To attack this complexity, we must understand these hidden relationships. At a structural level, code is made up of entities and directional relationships between them—"B uses A." Entities include functions, classes, datatypes, source files, and so on. Relationships arise through calls, inheritance, instantiation, and other programmatic techniques. If B uses A, then B depends on A to get its job done. If A doesn't perform as expected, then B's functionality might suffer. In turn, entities that depend on B might also suffer. In essence, linkages between entities increase the potential for changes to propagate through a system, creating unintended behaviors.

Modern code bases contain millions of entities and billions of paths between them. The most problematic of these paths are cyclical. The components along such paths both depend on, and are depended on by, many other components. As a code base grows, these cyclical groups—which we call *cores*—can proliferate as modularity and hierarchy erode, causing hundreds or thousands of source files to become mutually interdependent. In these cores, changes have strong, reinforcing ripple effects. A single change to a file can impact thousands of others, in distant parts of the system. Critically, this complexity can't be detected through inspection or code reviews. It's made visible only by tracing relationships between files across the system and its associated organizational groups.

Baldwin, MacCormack, and John Rusnak pioneered a visualization and analysis method that reveals a code base's hidden structure.[7] This method makes it clear which components are upstream in the system (depended on by others), which components are downstream (dependent on others), the system cores' location and size, and the degree to which the system as a whole is loosely coupled, versus being integral or monolithic. The method leverages a network-analysis technique called *Design Structure Matrices* (DSMs). DSMs capture directed relationships between system components. They provide both visual information about a system and quantitative metrics that capture its structure.

Figure 2 shows DSMs for two releases of an Olympus software product—before and after highly successful refactoring effort. Each DSM displays all the files in this system—more than 4,500 of them—in a square matrix. The dots represent a direct dependency between two files (for example, if file A uses file B, a dot appears in row A and column B). We sort files using algorithms that take into account their level of coupling and position in the system hierarchy. This results in a view in which as many dots as possible are below the diagonal. The dots above the diagonal indicate cyclical dependencies. Files that are part of the same cyclic group can be clustered together to show the system cores. In Figure 2, the red squares indicate the large cores.

In Figure 2a, the core is more than 800 files, or 15 percent of the system. Figure 2b shows that Olympus's successful refactoring effort split this core into two smaller ones with 250 and 150 files. In essence, our methods let us demonstrate that this refactoring significantly reduced cognitive complexity and the potential for changes to propagate. Our methods also let organizations make financial projections of the value released in such efforts; such projections are critical for projecting the ROI of refactoring projects.
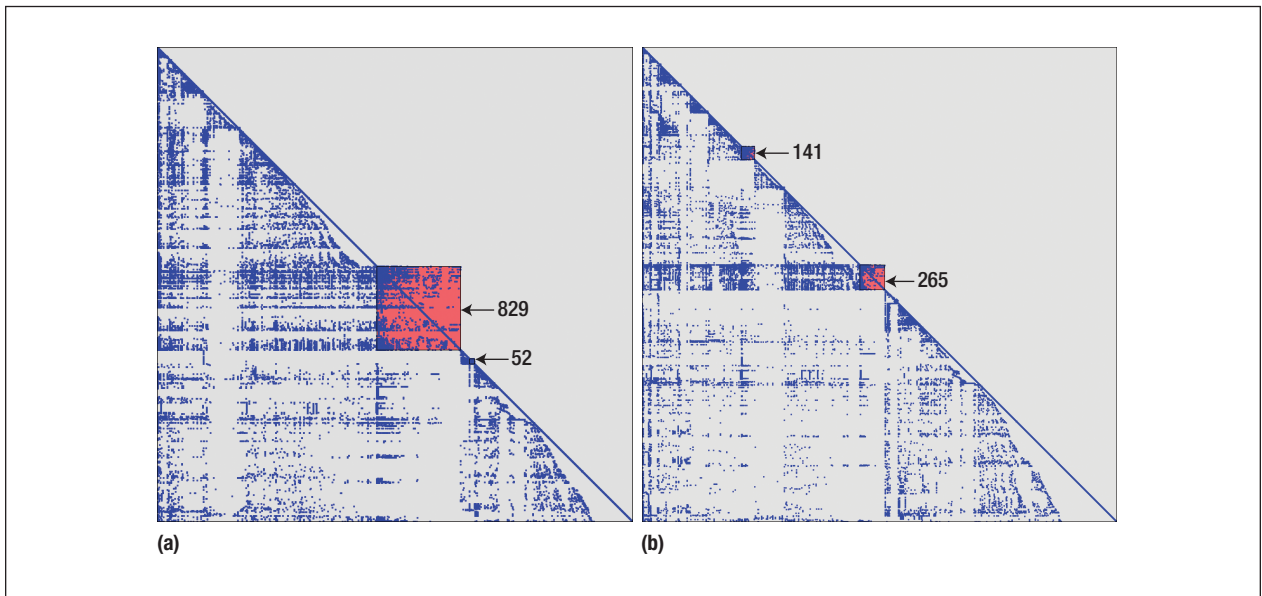
FIGURE 2. Design Structure Matrices for a commercial code base (a) before refactoring and (b) after refactoring. The dots above the diagonal indicate cyclical dependencies; the red square indicates the significant cores.

The Cost of Complexity

Architecture erosion has serious cost-of-ownership and risk consequences. Consider IronBridge,[3] a successful firm that developed and maintained a large product platform with 20,000 files. In this organization, one team faced significant performance challenges. Compared to other teams, it was more likely to miss schedules and ship buggy code. It had also experienced multiple unsuccessful modernization efforts. This team was responsible for subsystems containing approximately 2,600 files.

Our analysis found that 2,000 of those files were part of a large core. Historical scans revealed that this issue originated more than a decade prior and had grown over time. More than 100 developers were contributing new features and bug fixes to this core, creating a large extended team with combinatorically difficult communication requirements. Interviews revealed that even the most experienced developers had mental models that differed significantly from the actual relationships in the code base and from the mental models held by others.

To study this complexity's cost, we mined information from the firm's management systems. After controlling for traditional code-quality metrics, we found that the core files had substantial performance challenges regarding quality, productivity, and cost:

- The files had three times as many defects.
- Developers experienced a 60 percent decline in productivity and spent 70 percent of their time fixing defects (versus 20 percent in the periphery).
- More than 80 percent of the commits failed integration tests.

Not surprisingly, this complexity and the performance challenges it generated ultimately had a very human cost. Turnover among the developers who worked mostly in the core was 10 times greater than that for developers who worked elsewhere.

The combination of decreased productivity and increased effort on non-value-added activities creates a recipe for a severe competitive disadvantage. Confronted with competitors that possess healthy architectures and thus are more agile, firms that resist tackling complexity are likely to sow the seeds of their own destruction.

Architectural Agility for DevOps

Agile processes have deservedly received much attention because they're a better way to run development. However, 15 years of research and commercial projects have taught us that agile processes aren't always enough to make a project agile. The analysis of thousands of commercial, government, and open-source systems has led us to the following conclusions.

ABOUT THE AUTHOR

DAN STURTEVANT is the cofounder and CEO of Silverthread. Contact him at dan@silverthreadinc.com.

First, maintaining a healthy organization requires managing architectural health as a code base grows. We've never encountered a project or system that performed well when architecture degradation was significant.

Second, architectural agility can provide a competitive advantage. We often see competitive products with similar functionalities but very different architectural-health profiles.[8] Differences in nonessential complexity typically correlate with differences in innovation and market success.

Third, architecture degradation can lead to technical bankruptcy. We recently scanned a 50-year-old US federal government code base with a core of 11,000 files. Predictive analytics suggested that a typical change would take more than 80 days (versus 15 required for a three-week sprint) and that 80 percent of resources were being wasted fighting fires. Follow-up conversations validated these projections and revealed that the organization had been hamstrung for five years.

Finally, as Olympus demonstrated in Figure 2, refactoring can make a big difference.

If you simply use agile processes in a nonagile product architecture, you'll get faster at delivering nonvalue. Architecture will become the biggest bottleneck to your DevOps transformation. You need a balanced focus on agile process and agile architecture. With this approach, your organization can sustain excellence and succeed at the pace of delivery enabled by DevOps.

References

1. N. Fosgren et al., *2017 State of DevOps Report*, Puppet, 2017; puppet.com/resources/whitepaper /state-of-devops-report.
2. A. MacCormack and D.J. Sturtevant, "Technical Debt and System Architecture: The Impact of Coupling on Defect-Related Activity," *J. Systems and Software*, Oct. 2016, pp. 170–182.
3. D.J. Sturtevant, "System Design and the Cost of Architectural Complexity," PhD dissertation, MIT, 2013.
4. A. Akaikine, "The Impact of Software Design Structure on Product Maintenance Costs and Measurement of Economic Benefits of Product Redesign," master's thesis, MIT, 2010.
5. S.M. Gilliland, "Empirical Analysis of Software Refactoring Motivation and Effects," master's thesis, MIT, 2015.
6. C.W. Berardi, "Intellectual Property and Architecture: How Architecture Influences Intellectual Property Lock-In," PhD dissertation, MIT, 2017.
7. C. Baldwin, A. MacCormack, and J. Rusnak, "Hidden Structure: Using Network Methods to Map System Architecture," *Research Policy*, vol. 43, no. 8, 2014, pp. 1381–1397.
8. A. MacCormack, C. Baldwin, and J. Rusnak, "Exploring the Duality between Product and Organizational Architectures: A Test of the 'Mirroring' Hypothesis," *Research Policy*, vol. 41, no. 8, 2012, pp. 1309–1324.