Simplicity

Written by Max Luebbe
Edited by Tim Harvey

> *The price of reliability is the pursuit of the utmost simplicity.*
>
> C.A.R. Hoare, Turing Award lecture

Software systems are inherently dynamic and unstable.[38] A software system can only be perfectly stable if it exists in a vacuum. If we stop changing the codebase, we stop introducing bugs. If the underlying hardware or libraries never change, neither of these components will introduce bugs. If we freeze the current user base, we'll never have to scale the system. In fact, a good summary of the SRE approach to managing systems is: "At the end of the day, our job is to keep agility and stability in balance in the system."[39]

System Stability Versus Agility

It sometimes makes sense to sacrifice stability for the sake of agility. I've often approached an unfamiliar problem domain by conducting what I call exploratory coding—setting an explicit shelf life for whatever code I write with the understanding that I'll need to try and fail once in order to really understand the task I need to accomplish. Code that comes with an

expiration date can be much more liberal with test coverage and release management because it will never be shipped to production or be seen by users.

For the majority of production software systems, we want a balanced mix of stability and agility. SREs work to create procedures, practices, and tools that render software more reliable. At the same time, SREs ensure that this work has as little impact on developer agility as possible. In fact, SRE's experience has found that reliable processes tend to actually increase developer agility: rapid, reliable production rollouts make changes in production easier to see. As a result, once a bug surfaces, it takes less time to find and fix that bug. Building reliability into development allows developers to focus their attention on what we really do care about—the functionality and performance of their software and systems.

The Virtue of Boring

Unlike just about everything else in life, "boring" is actually a positive attribute when it comes to software! We don't want our programs to be spontaneous and interesting; we want them to stick to the script and predictably accomplish their business goals. In the words of Google engineer Robert Muth, "Unlike a detective story, the lack of excitement, suspense, and puzzles is actually a desirable property of source code." Surprises in production are the nemeses of SRE.

As Fred Brooks suggests in his "No Silver Bullet" essay [Bro95], it is very important to consider the difference between essential complexity and accidental complexity. Essential complexity is the complexity inherent in a given situation that cannot be removed from a problem definition, whereas accidental complexity is more fluid and can be resolved with engineering effort. For example, writing a web server entails dealing with the essential complexity of serving web pages quickly. However, if we write a web server in Java, we may introduce accidental complexity when trying to minimize the performance impact of garbage collection.

With an eye towards minimizing accidental complexity, SRE teams should:

- Push back when accidental complexity is introduced into the systems for which they are responsible

- Constantly strive to eliminate complexity in systems they onboard and for which they assume operational responsibility

I Won’t Give Up My Code!

Because engineers are human beings who often form an emotional attachment to their creations, confrontations over large-scale purges of the source tree are not uncommon. Some might protest, "What if we need that code later?" "Why don’t we just comment the code out so we can easily add it again later?" or "Why don’t we gate the code with a flag instead of deleting it?" These are all terrible suggestions. Source control systems make it easy to reverse changes, whereas hundreds of lines of commented code create distractions and confusion (especially as the source files continue to evolve), and code that is never executed, gated by a flag that is always disabled, is a metaphorical time bomb waiting to explode, as painfully experienced by Knight Capital, for example (see "Order In the Matter of Knight Capital Americas LLC" [Sec13]).

At the risk of sounding extreme, when you consider a web service that’s expected to be available 24/7, to some extent, every new line of code written is a liability. SRE promotes practices that make it more likely that all code has an essential purpose, such as scrutinizing code to make sure that it actually drives business goals, routinely removing dead code, and building bloat detection into all levels of testing.

The "Negative Lines of Code" Metric

The term "software bloat" was coined to describe the tendency of software to become slower and bigger over time as a result of a constant stream of additional features. While bloated software seems intuitively undesirable, its negative aspects become even more clear when considered from the SRE perspective: every line of code changed or added to a project creates the potential for introducing new defects and bugs. A smaller project is easier to understand, easier to test, and frequently has fewer defects. Bearing this perspective in mind, we should perhaps entertain reservations when we have the urge to add new features to a

project. Some of the most satisfying coding I've ever done was deleting thousands of lines of code at a time when it was no longer useful.

Minimal APIs

French poet Antoine de Saint Exupery wrote, "perfection is finally attained not when there is no longer more to add, but when there is no longer anything to take away" [Sai39]. This principle is also applicable to the design and construction of software. APIs are a particularly clear expression of why this rule should be followed.

Writing clear, minimal APIs is an essential aspect of managing simplicity in a software system. The fewer methods and arguments we provide to consumers of the API, the easier that API will be to understand, and the more effort we can devote to making those methods as good as they can possibly be. Again, a recurring theme appears: the conscious decision to not take on certain problems allows us to focus on our core problem and make the solutions we explicitly set out to create substantially better. In software, less is more! A small, simple API is usually also a hallmark of a well-understood problem.

Modularity

Expanding outward from APIs and single binaries, many of the rules of thumb that apply to object-oriented programming also apply to the design of distributed systems. The ability to make changes to parts of the system in isolation is essential to creating a supportable system. Specifically, loose coupling between binaries, or between binaries and configuration, is a simplicity pattern that simultaneously promotes developer agility and system stability. If a bug is discovered in one program that is a component of a larger system, that bug can be fixed and pushed to production independent of the rest of the system.

While the modularity that APIs offer may seem straightforward, it is not so apparent that the notion of modularity also extends to how changes to APIs are introduced. Just a single change to an API can force developers to rebuild their entire system and run the risk of introducing new bugs. Versioning APIs allows developers to continue to use the version that their system depends upon while they upgrade to a newer version in a safe and considered

way. The release cadence can vary throughout a system, instead of requiring a full production push of the entire system every time a feature is added or improved.

As a system grows more complex, the separation of responsibility between APIs and between binaries becomes increasingly important. This is a direct analogy to object-oriented class design: just as it is understood that it is poor practice to write a "grab bag" class that contains unrelated functions, it is also poor practice to create and put into production a "util" or "misc" binary. A well-designed distributed system consists of collaborators, each of which has a clear and well-scoped purpose.

The concept of modularity also applies to data formats. One of the central strengths and design goals of Google's protocol buffers[40] was to create a wire format that was backward and forward compatible.

Release Simplicity

Simple releases are generally better than complicated releases. It is much easier to measure and understand the impact of a single change rather than a batch of changes released simultaneously. If we release 100 unrelated changes to a system at the same time and performance gets worse, understanding which changes impacted performance, and how they did so, will take considerable effort or additional instrumentation. If the release is performed in smaller batches, we can move faster with more confidence because each code change can be understood in isolation in the larger system. This approach to releases can be compared to gradient descent in machine learning, in which we find an optimum solution by taking small steps at a time, and considering if each change results in an improvement or degradation.

A Simple Conclusion

This chapter has repeated one theme over and over: software simplicity is a prerequisite to reliability. We are not being lazy when we consider how we might simplify each step of a given task. Instead, we are clarifying what it is we actually want to accomplish and how we might most easily do so. Every time we say "no" to a feature, we are not restricting innovation;

we are keeping the environment uncluttered of distractions so that focus remains squarely on innovation, and real engineering can proceed.

---

[38]This is often true of complex systems in general; see [Per99] and [Coo00].

[39]Coined by my former manager, Johan Anderson, around the time I became an SRE.

[40]Protocol buffers, also referred to as "protobufs," are a language-neutral, platform-neutral extensible mechanism for serializing structured data. For more details, see *https://developers.google.com/protocol-buffers/docs/overview#a-bit-of-history*.

← PREVIOUS
Chapter 8 - Release Engineering

NEXT
Part III - Practices