Editor: **George Fairbanks**
gf@georgefairbanks.com

The Fog of Software Design

Timothy J. Halloran

From the Editor

Your software projects never quite go as smoothly as the textbooks promise, do they? Tim Halloran explains where the fog of design comes from and tells you how to handle it. —*George Fairbanks*

OUR ABILITY TO design software is frustrated because critical information is shrouded by fog. We misunderstand requirements, we imperfectly predict how those requirements will evolve, we misinterpret the existing code, and we wish we understood design principles better. These forces are the fog of software design. Developers who are aware of them can take steps to overcome the fog and deliver better software.

Carl von Clausewitz introduced the phrase "the fog of war," saying, "War is the realm of uncertainty; three quarters of the factors on which action in war is based are wrapped in a fog of greater or lesser uncertainty."[1] Good military leaders work to build up a "sensitive and discriminating judgment" when considering battle reports, intelligence, and other information—they suspect all of being incomplete, misleading, or factually wrong. von Clausewitz cautions military leaders to anticipate uncertainty and incomplete situational awareness in military operations.

Software designers face similar uncertainty. The fog analogy can help us reflect on some of the information challenges we face in our design work and avoid throwing up our hands in frustration and just coding. Many of the ideas in this article were raised by Parnas and Clements more than three decades ago in their approach to "fake" a rational design process.[2]

In this article, I will discuss the fog of software design, the problems it creates, and how to clear the fog as much as we can. Let's begin with requirements.

The Fog of Understanding Requirements

Software requirements are rarely well articulated or complete. Parnas and Clements lament, "In most cases the people who commission the building of a software system do not know exactly what they want and are unable to tell us all that they know."[2] A humorous quote from Henry Ford about requirements for the automobile captures a similar sentiment, "If I had asked people what they wanted, they would have said faster horses." Understanding requirements is a big topic that underpins why teams today favor iterative development. In my analogy, the fog obscures the requirements, and we have to invest time and effort to make them clear. Information we are given can be misleading (for example, "faster horses"). Venturing out into the dense fog is uncomfortable, and we tend to gravitate to the less foggy areas of the problem domain—to our peril.

One large defense software project I worked on failed because it avoided dense fog, ignoring the risky work of understanding the requirements for interfacing with aging sensor systems. Engineers found it more comfortable to work on designs for clever data processing and a vibrant new user experience. Management

Digital Object Identifier 10.1109/MS.2021.3056937
Date of current version: 16 April 2021

0740-7459/21©2021IEEE

was thrilled, good progress was made, and prototypes emerged that looked wonderful. Sadly, after several years of work, we never got the sensor system drivers to keep up with the data needs of other flashier parts of the system. Costs got too high, and the project was canceled. The team focused on the low-risk parts of the design and implementation where the fog wasn't too thick and doomed the project.

How can you deal with requirements in the fog? Spend your time on risky or contentious design elements. This is hard because it is human nature to gravitate toward comfortable work that we can make steady progress on. Resist this urge. Sound the foghorn and venture into the thickest regions. Be worried if your design work feels too comfortable or easy. Work hard to uncover high-risk elements like the aging sensor systems that doomed my project. My advice contradicts the current fad of "highest customer value first." You should tackle the highest risk first. A risk-driven approach works best in my experience and is consistent with Boehm's spiral model.[3]

The Fog of Anticipating Requirements

To make good design choices today, software developers try to anticipate future requirements. Guessing what is going to change over time is like peering into the fog to try to see into the future. Most of the time, this turns out to be a bad idea.

I have wasted many hours peering into the fog of system evolution and getting it utterly wrong. One example blunder was in a Java dynamic analysis tool I designed while working for a start-up company. This tool used a Structured Query Language (SQL) database to store Java program events (for example, lock acquired and field read) and query useful information for the user (for example, race condition observed). It used Apache Derby as the database; however, I wanted to support other databases, such as Oracle. There might have been a vague business requirement but nothing concrete—the marketing folks didn't need this flexibility. To implement this feature, we used Java's ResourceBundles to support database-specific SQL language variations and abstracted database bootstrapping. The implementation was complex and required weeks of work by multiple engineers and thousands of lines of code. The feature worked flawlessly, was well tested, and was pretty easy to maintain. But all of that time was wasted. We never used a different database or reused the code in another tool, yet we paid the development price and the complexity price for our bad prediction.

Guessing system evolution and building infrastructure software is related. Good judgment in this area is difficult. Of course, not all infrastructure work is bad. However, I've learned to be skeptical about infrastructure projects. Is this needed? Will it be used? Designers and engineers (like me) love this kind of work. Management tends to as well. Most infrastructure, like my database flexibility feature, is clever and can be

THE FOUR FOGS OF DESIGN

Our ability to design software is frustrated because critical information is shrouded by fog.

The fog of understanding requirements

- Software requirements are rarely well articulated or complete. Be tenacious about uncovering risks and really understanding your problem domain. Avoid "comfortable work" on low-risk parts of the system even if this makes management happy.

The fog of anticipating requirements

- Designers try to anticipate future requirements. Most of the time, this is a bad idea. Don't build what you don't need. Infrastructure projects take resources away from other work. Be sure they are reusable and really needed.

The fog of existing code

- When we rewrite existing code, we often don't understand—or have forgotten—the key design decisions that made the production system successful. Design intent has been lost. This can doom a project or, worse, cause it to drag on forever.

The fog of design knowledge

- We don't consider many potential designs due to our limited expertise. Get feedback and deeply value design ideas from others. Contention, while uncomfortable, helps you design better and longer-lasting software systems.

used in several of a company's products. But there is an opportunity cost, and that's my caution—infrastructure development means other projects don't get resourced. What would be a good criterion with which to judge an infrastructure project? Consider if it abstracts a difficult requirement and is usable by multiple software systems. How many systems should reuse it? I agree with Tracz, who argued that "you need to reuse it three times" to have confidence that it is really reusable.[4]

How do we clear the fog of system evolution? Don't guess. Base decisions on requirements or the real potential for reuse. My database flexibility feature was based on neither of these and should not have made it into the design. Realize that infrastructure projects take resources away from other, perhaps more profitable, work. Remember Tracz's rule of three for reuse when contemplating nontrivial infrastructure projects.

The Fog of Existing Code

When we rewrite or modernize existing code, we often don't understand—or have forgotten—the key design decisions that made the production system successful. This creates a special kind of fog around a production system: the fog of lost design intent. This can doom a project, or worse, cause it to drag on forever.

I tend to sum up my Air Force career with the phrase, "I retired a lot of mainframe computers." I'm being a bit tongue in cheek, but moving code off a mainframe to a modern computer (typically to a new programming language as well) is a classic software modernization project. It expends software development costs to make the system easier (and hopefully cheaper) for the organization to support long term. Here, let's consider substantial modernization efforts, such as a total system rewrite in a new language. In these larger projects, the fog around the existing code is insidious—the gory details are all there in the source code, and teams often feel confident that they understand it all. They never do.

One failure I observed was a project trying to move a large mainframe Cobol system to operating system (OS)/2. It seemed to me that the designer was far more concerned about using every shiny new OS/2 feature than understanding the production system. More commonly, I've observed a string of modernization projects that never quite seem to replace the production system, taking years to finally succeed (if they do succeed at all). How do we overlook the successful design elements of production software? I believe it's because system maintainers rarely work on them. They are stable, reliable, and easily taken for granted.

How do we clear the fog of existing code? Design the updated system such that its major components (pieces) can be used in production quickly—before the entire implementation is complete. Avoid implementing everything and then "flipping the big switch." Carefully transition each piece into production, for example, with an experimental rollout, but take the time and effort to launch them one by one. Why? This approach exposes your work to the unforgiving production environment. You'll cut through the fog of existing code and rediscover key design decisions in the old system that you missed. This is the only technique I've had success with in practice, but it has a cost. The emerging new code has to interact with the old production code. This increases the development cost, creates a lot of work in the old code base, might drive hardware costs, and limits architectural improvements to the system (at least during transition). However, in my experience with "retiring" mainframes, it ensures project success.

The Fog of Design Knowledge

None of us knows everything about software design. This internal fog starts where our design knowledge ends or where we become uncomfortable. We don't consider potential designs because we haven't been exposed to them, or we view them as risky because we don't have personal experience applying them. Further, as Parnas and Clements note, "We are often burdened by preconceived design ideas—ideas that we invented, acquired on related projects, or heard about in a class."[2]

As a concrete example of where this type of fog blinded me, let's consider using the visitor pattern in program analysis. When I first started designing analysis tools for Java security, I did not know the visitor pattern,[5] so each time we added an analysis, we also had to add another polymorphic method to our Java abstract syntax tree classes—which turned into a maintenance mess. When I was introduced to the visitor pattern, I viewed it with suspicion. Didn't it seem complex? Wasn't a simple method easier? My lack of knowledge and discomfort with new ideas resulted in us having to redesign our tools within a year of launch.

How can you clear the fog of limited design knowledge? Get feedback. Treasure other design ideas and feedback from as many folks as you can engage. This is an excellent way to expand the pool of design knowledge beyond yourself and help you become comfortable with

new approaches. Deeply consider the feedback you get from others, don't just dismiss it. I caution, however, that this will create contention. Alternative designs will be proposed, and parts or all of your design will be criticized. This can be uncomfortable but leads to a better outcome. I've seen this advice to gather feedback and embrace a contentious design process ignored so often that I'd almost call it Halloran's law: any noncontentious software design will be reimplemented within a few years. Why? Like my program analysis systems that didn't use the visitor pattern, these designs tend to be mediocre. You might be able to implement them, as I did, but they will quickly reveal their deficiencies.

Clearing the Fog

How can we clear the fog of software design? Let's look at three approaches, and then I'll give you my advice.

Your schedule-stressed coworkers may espouse the "panic and hack" approach. They argue that if we agree the fog is an obstacle to design, let's skip design altogether and just implement features. They say that the code will flow, we'll meet our schedule, and management will be happy (for a while). Unfortunately, this won't scale. Larger software systems built this way, if they ever launch, will have both maintenance problems and short lifetimes. I disagree with your coworkers—design is critical and should not be skipped.

Academics and design books tend to take an omniscient perspective on design, without any fog, so that a student can learn the principles. This is an essential simplification for teaching; however, it is impractical. Be wary. The perfection of a pattern will get messy in a real system. Specifications will rarely exist, and you'll have to infer them from code. Things always look clean and easy in a book, so learn the principles from them, but you will need additional advice for coping with the fog.

Parnas and Clements suggest that we "fake" a rational design process and produce (copious) documentation that rewrites the project's messy history.[2] Their ideas benefited from experience on the design of an update to the A-7E Avionics System, a complex real-time embedded system for a military aircraft.[6] Parnas and Clements understood the fog but perhaps overgeneralized their experience. With three decades of hindsight, I disagree with the volume and precision of the documentation suggested by their process. For most domains, such documentation is economically impractical. All too often, the rate of change on the project would quickly render such documents obsolete. Even for real-time embedded systems, new techniques have emerged, such as model checking, that enable precision in a more useful form than documentation.

Critical information driving your design will be shrouded in fog, so expect to find broken abstractions, incomplete specifications, and misleading documentation. What is my advice? You can't prevent the fog, but you can anticipate it and be ready. More concretely, use design abstractions and iterate. Keep your work rigorously risk driven. Tailor documentation formality to your problem domain. Solicit feedback on your work. Vigorous design discussions lead to better designs. Don't panic, keep your head, and you can design despite the fog.

ABOUT THE AUTHOR

TIMOTHY J. HALLORAN is a software engineer at Google, Pittsburgh, Pennsylvania, 15206, USA and a retired U.S. Air Force Lieutenant Colonel. Contact him at hallorant@gmail.com.

References

1. C. v. Clausewitz, *Vom Kriege*, Bonn, Germany: Dümmler, 1832, Book 1, ch. 3, p. 101.
2. D. Parnas and P. Clements, "A rational software process: How and why to fake it," *IEEE Trans. Softw. Eng.*, vol. SE-12, no. 2, pp. 251–257, Feb. 1986. doi: 10.1109/TSE.1986.6312940.
3. B. Boehm, "A spiral model of software development and enhancement," *Computer*, vol. 21, no. 5, pp. 61–72, May 1988. doi: 10.1109/2.59.
4. W. Tracz, *Confessions of a Used Program Salesman*. Reading, MA: Addison-Wesley, 1995.
5. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*. Reading, MA: Addison-Wesley, 1995.
6. L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 2nd ed. Reading, MA: Addison-Wesley, 2003.