Software Architecture is Overrated, Clear and Simple Design is Underrated

Gergely Orosz

I had my fair share in designing and building large systems. I've taken part in rewriting Uber's distributed payment systems, designing and shipping Skype on Xbox One and open-sourcing RIBs, Uber's mobile architecture framework. All of these systems had thorough designs, going through multiple iterations and had lots of whiteboarding and discussion. The designs then boiled down to a design document that was circulated for more feedback before we started building.

All of these systems were large at scale: hundreds of developers build them - or on top of them - and they power systems used by millions of people per day. They were also not just greenfield projects. The payments system rewrite had to replace two, existing payments systems, used by tens of systems and dozens of teams, all without having any business impact. Rewriting the Uber app was a project that a few hundred engineers worked simultaneously on, porting existing functionality to a new architecture.

Let me start with a few things that might sound surprising. **First, none of these designs used any of the standard software architecture planning tools.** We did not use UML, nor the 4+1 model, nor ADR, nor C4, nor dependency diagrams. We created plenty of diagrams, but none of them followed any strict rules. Just plain old boxes and arrows, similar this one describing information flow or this one outlining class structure and relationships between components. Two diagrams within the same design document often had a different layout and were often added and modified by different engineers.

Second, there were no architects on the teams that owned the design. No IT architects or enterprise architects. True, neither Uber nor Skype/Microsoft have hands-off software architect positions. Engineers at higher levels, like staff engineers, are expected to still regularly code. For all the projects, we did have experienced engineers involved. However, no one person owned the architecture or design. While these experienced developers drove the design process, even the most junior team members were involved, often challenging decisions and offering other alternatives to discuss.

Third, we had practically no references to the common architecture patterns and other jargon referenced in common software architecture literature, such as Martin Fowler's architecture guide. No mentions of microservices, serverless architecture, application boundaries, event-driven architecture, and the lot. Some of these did come up during brainstormings. However, there was no need to reference them in the design documents themselves.

Software design at tech companies and startups

So how did we get things done? And why did we not follow approaches suggested by well-known software architecture approaches?

I've had this discussion with peer engineers working at other tech companies, FANG (Facebook, Amazon, Netflix, Google), as well as at smaller startups. Most teams and projects - however large or small - all shared a similar approach to design and implementation:

1. **Start with the business problem**. What are we trying to solve? What product are we trying to build and why? How can we measure success?
2. **Brainstorm the approach**. Get together with the team and through multiple sessions, figure out what solution will work. Keep these brainstormings small. Start at a high level, going down to lower levels.
3. **Whiteboard your approach**. Get the team together and have a person draw up the approach the team is converging to. You should be able to explain the architecture of your system/app on a whiteboard clearly, starting at the high-level, diving deeper as needed. If you have trouble with this explanation or it's not clear enough, there's more work required on the details.
4. **Write it up via simple documentation with simple diagrams** based on what you explained on the whiteboard. Keep jargon to the minimum: you want even junior engineers to understand what it's about. Write it using clear and easy to follow language. At Uber, we use an RFC-like process with various templates.
5. **Talk about tradeoffs and alternatives**. Good software design and good architecture are all about making the right tradeoffs. No design choice is good or bad by itself: it all depends on the context and the goals. Is your architecture split into different services? Mention why you decided against going with one large service, that might have some other benefits, like more straightforward and quicker deployment. Did you choose to extend a service or module with new functionality? Weigh the option of building a separate service or module instead, and what the pros and cons of that approach would be.
6. **Circulate the design document within the team/organization and get feedback**. At Uber, we used to send out all our software design documents to all engineers, until there were around 2,000 of us. Now that we're larger, we still distribute them very widely, but we've started balancing the signal/noise ratio more. Encourage people asking questions and offering alternatives. Be pragmatic in setting sensible time limits to discuss the feedback and incorporate it, where it's needed. Straightforward feedback can be quickly addressed on the spot, while more detailed feedback might be quicker to settle in-person.

Why was our approach different from what is commonly referred to in software architecture literature? Actually, our approach is not that different in principle, to most architecture guides. Almost all guides suggest starting with the business problem and outlining solutions and tradeoffs: which is also what we do. What we don't do is use many of the more complex tools that many architects or architecture books advocate for. We document the design as simple as we can, using the most straightforward tools: tools like Google Docs or Office365.

I assume that the main difference in our approach boils down to engineering culture at these companies. High autonomy and little hierarchy is a trait tech companies and startups share: something that is sometimes less true for more traditional companies. This is also a reason these places do a lot more "common sense-based design" over process-driven design, with stricter rules.

I know of banks and automotive companies where developers are actively discouraged from making any architecture decisions without going up the chain, getting signoff from architects a few levels up, who are overseeing several teams. This becomes a slower process, and architects can get overwhelmed with many requests. So these architects create more formal documents, in hopes of making the system more clear, using much more of the tools the common literature describes. These documents also reinforce a top-down approach, as it is more intimidating for an engineer, who is not an architect, to question or challenge the decisions that have already been documented using formal methods, that they are not that well-versed in. So they usually don't do so. To be fair, these same companies often want to optimize for developers to be more as exchangeable resources, allowing them to re-allocate people to work on a different project, on short notice. It should be no surprise that different tools work better in different environments.

Simple, jargonless software design over architecture patterns

The goal of designing a system should be simplicity. The simpler the system, the simpler it is to understand, the simpler it is to find issues with it and the simpler it is to implement it. The more clear language it is described in, the more accessible that design is. Avoid using jargon that is not understood by every member of the team: the least experienced person should be able to understand things equally clearly.

Clean design is similar to clean code: it's easy to read and easy to comprehend. There are many great ways to write clean code. However, you will rarely hear anyone suggesting to start with applying the Gang of four design patterns to your code. Clean code starts with things like single responsibility, clear naming, and easy to understand conventions. These principles equally apply to clear architecture.

So what is the role of architecture patterns? I see them similarly in usefulness as coding design patterns. They can give you ideas on how to improve your code or architecture. For coding patterns, I notice a singleton pattern when I see one, and I raise my eyebrow and dig deeper when I see a class that acts as a facade, only doing call-throughs. But I've yet to think *"this calls for an abstract factory pattern"*. In fact, it took me a lot of time to understand what this pattern does and had my "aha!" moment, after working with a lot of dependency injection - one of the few areas, where this pattern is actually pretty common and useful. I'll also admit that although I spent a lot of time reading and comprehending the Gang of four design patterns, they've had far less impact on becoming a better coder than the feedback I've gotten from other engineers on my code.

Similarly, knowing about common architecture patterns is a good thing: it helps shorten discussions with people, who understand them the same way as you do. But architecture patterns are not the goal, and they won't substitute for simpler system designs. When designing a system, you might find yourself having accidentally applied a well-known pattern: and this is a good thing. Later, you can reference your approach easier. But the last thing you want to do is taking one or more architecture pattern, using it as a hammer, looking for nails to use it on.

Architecture patterns were born after engineers observed how similar design choices were made in some cases, and those design choices were implemented similarly. These choices were then named, written down, and extensively talked about. Architecture patterns are tools that came *after* the solution was solved, in hopes of making the lives of others easier. **As an engineer, your goal should be more about solving solutions and learning through them rather than picking a shiny architecture pattern, in hopes that that will solve your problem.**

Newsletter

Enjoying this article? Subscribe to my newsletter to get issues like this in your inbox. It's a good read and the #1 technology newsletter on Substack.

Getting better at designing systems

I've heard many people ask for tips on becoming better in architecting and designing systems. Several experienced people will recommend reading up on architecture patterns and reading books on software architecture. While I definitely do recommend reading - especially books, as they provide a lot more depth than a short post - I have a few suggestions, that are all more hands-on than just reading.

- **Pull over a teammate and whiteboard your design approach**. Draw up what you are working on and why you are doing things. Make sure they understand. And when they do, ask for their feedback.
- **Write up your design in a simple document and share it with your team, asking for feedback**. No matter how simple or complex thing you're working on, may that be a smaller refactor or a large project, summarize this. Do it in a way that makes sense to you and a way that others can understand - for inspiration, here's how I've seen it done at Uber. Share it with your team in a format that allows commenting, like Google Docs, Office365, or others. Ask people to add their thoughts and questions.
- **Design it two different ways and contrast the two designs.** When most people design an architecture, they go with one approach: the one popping in their head. However, architecture is not black-and-white. Come up with a second design that could also work. Contrast the two, explaining why one is better than the other. List the second design briefly as an alternative considered, arguing why it was decided against.
- **Be explicit about tradeoffs** you make, why you made them, and what things you have optimized for. Be clear about constraints that exist and you've had to take into account.
- **Review other's designs. Do it better.** Assuming you have a culture, where people share their designs via whiteboarding and sessions or documents, get more out of these reviews. During a review, most people only try to take things in, becoming one-way observers. Instead, ask clarifying questions for parts that are not clear. Ask them about other alternatives they've considered. Ask them what tradeoffs they've taken and what constraints they've assumed. Play devil's advocate and suggest another, possibly simpler alternative - even if it's not a better one - asking them their thoughts on your suggestion. Even though you've not thought as much about the design as the person presenting it, you can still add a lot of value and learn more.

The best software design is simple and easy to understand. The next time you're starting a new project, instead of thinking, *"How will I architect this system, what battle-tested patterns should I use and what formal methodology should I document it with?"*, think *"How can I come up with the simplest possible design, in a way that's easy for anyone to understand?"*.

Software architecture best practices, enterprise architecture patterns, and formalized ways to describe systems are all tools that are useful to know of and might come in handy one day. But **when designing systems, start simple and stay as simple as you can. Try to avoid the complexity that more complex architecture and formal tools inherently introduce.**

Read my follow-up Twitter thread on this post. You can also read the Russian translation of this article on Habr and the Chinese translation on InfoQ China.

Subscribe to my weekly newsletter to get articles like this in your inbox. It's a pretty good read -

and the #1 tech newsletter on Substack.