Just Write Dumb, Boring, Predictable Code

Even a giant Hong Kong-style apartment building is just an aggregation of hundreds of tiny little apartments

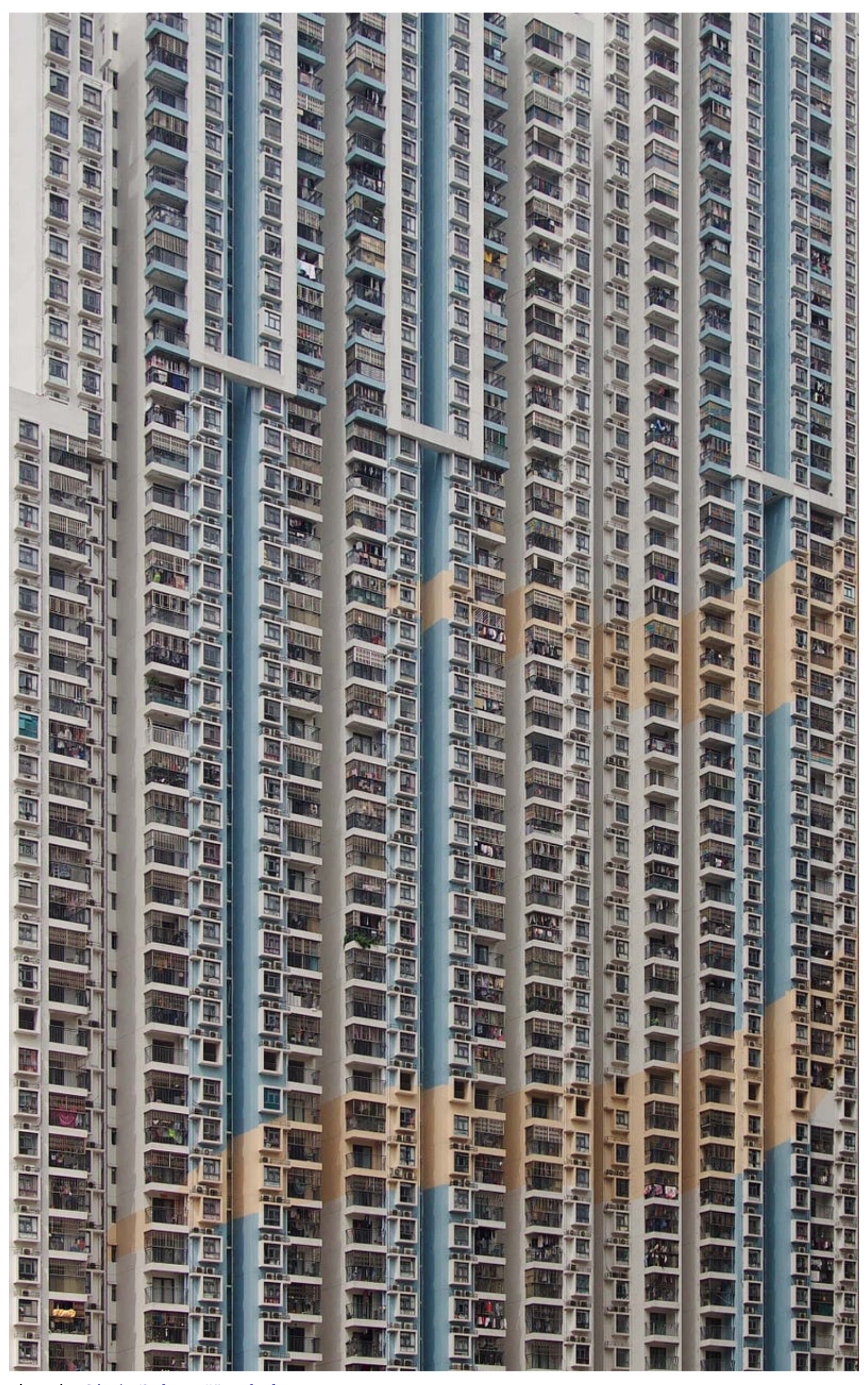
Photo by Sérgio Rola on Unsplash

The one belief every developer needs to debunk is that "my system is so complicated!"

I really, honestly question that it is all that complicated.

Is that "complicatedness" an outcome of poor structure, mental models, solutioning? Or is it inherently complex and multi-faceted? As software engineers and/or architects, our job is to make that complexity tolerable and manageable — simple, even.

I trust computers a whole lot more than I trust people, and that's for a simple reason: They are almost always more reliable than people. They don't change their mind. At least not often enough to be a nuisance.

Yet most developers' reality seems to deal with gripes about weird behavior and whatnot in their software.

Again: If you trust your software to actually run the way you intended it to, then why is it not working? Let me be frank and wager a guess: You didn't build it quite right. I'm not picking a fight, but is that really not the root cause?

What is so hard about writing software? Making something messy simple, I'd say. Why then focus on the syntactic details, if we are (metaphorically) semantically and structurally impaired? Thankfully a lot of wisdom has been produced around this area.

I love concepts like the "always valid domain model," in which software should not allow the internal representation to be in an invalid — and thus faulty, problematic, error-prone — state. It goes beyond trivial validation and ensures a full-cycle, truthful handling of your data, resulting in predicable, boring — working! — software.

After all, and this will be very inconclusive, writing software can pragmatically boil down to only a few key components. What if you pushed yourself to revisit and actually implement some key bits of the software engineering canon?

Let me pick some favorites with my flavor below…

All code is a liability

Infrastructure introduces even more liability. More infrastructure is worse than having less of it, given that your solution is at least equally good at fulfilling the requirements.

Operations (API paths, functions, code paths…) result in either queries ("reads") or commands ("writes") in a system. Never both.

Write deterministic — if possible, even idempotent — code

Behavior across calls must never change based on the same input being passed in.

Write unit tests for everything

First, for all of your use cases, both "positive" (happy flow) and "negative" (known failure state) ones, and then bit by bit, for logical functionality to reach ~100% coverage. Every time you learn something new about your behavior, keep the tests updated with that knowledge, i.e., write new tests to cover it.

You never write tests for yourself. You write them for others. And yourself, in the future, because, to be frank — memory is short and frail.

Every system has outer interfaces, which your primary tests need to assert. How these are called — API, event, RPC — is of no consequence whatsoever. Accept all possibilities and decouple the business logic from the outer layer that receives the initial function call. Keep the outer layer so stupid you don't have to write tests for it.

Use abstractions and interfaces to decouple implementation from concepts. Aim for meaningful, semantic abstractions that result in easier-to-read code. It's not an "elaboration contest"; it's about making each individual level of abstraction maximally dumb and predictable.

Question the validity of integration tests

What exactly do you think you get from integration tests that you don't get logically from unit tests? If you can answer the question without sweating during Russian roulette, write integration tests — as few as possible.

Hate magicians and magic; don't be clever and don't trust mystery code

Write clear and semantic code. Comment and document liberally on the structural level (because how things are wired together is not necessarily immediately apparent) but avoid commenting individual lines. If you do, question why that implementation is OK in the first place. Sounds like some magic is happening there, and you know what we think about magicians!

Side effects can be intentional (OK!) or unintentional (not OK!)

Remove the possibility of any clearly unintentional side effects. Never test side effects or input/output, which is a related phenomena.

The scope of a system can often be outlined around its transactional scope. Keep everything that changes together, together. This way, a system can easily represent logically consistent things, such as entities, without being displaced illogically across many systems.

Ensure total system independence

No external system can assert influence on your internals. This is why we have contracts and API schemas.

Ensure interoperability and use strategies like versioning to avoid breaking user functionality. Avoid keeping more than two versions of the service at any given time.

Do not assert behavior across more than a single system

The fallacy is to think of a whole "journey": It is no more than the aggregate effect of several independent parts. There is no "big journey" unless all the small parts work on their own.

"Testing in production" is an unfortunate but darkly cynical way to say: You will never know all the ways your code will fuck up, so you better make sure you have insights whenever that bad day comes. Use logging, metrics, and traces with rich, structured formats together with competent try/catch behaviors to ensure the system behaves well and that you have information at hand when you need to investigate.

Using these concepts, there should be no major difference between a "trivial" and "complex" service — they are equally testable, deterministic, and competently built.

When you've done these things to your code, see if it's still "so complicated," untestable, and unmanageable. I think you'll be surprised by what you find.

By Mikael Vesavuori on February 22, 2023.

Canonical link

Exported from Medium on July 16, 2023.