How To NOT Over Engineer

NTCU

Jason Edelman

9/5/2023

>>> What are we really asking?

It would seem we are trying to NOT over-engineer, but what is over-engineering?

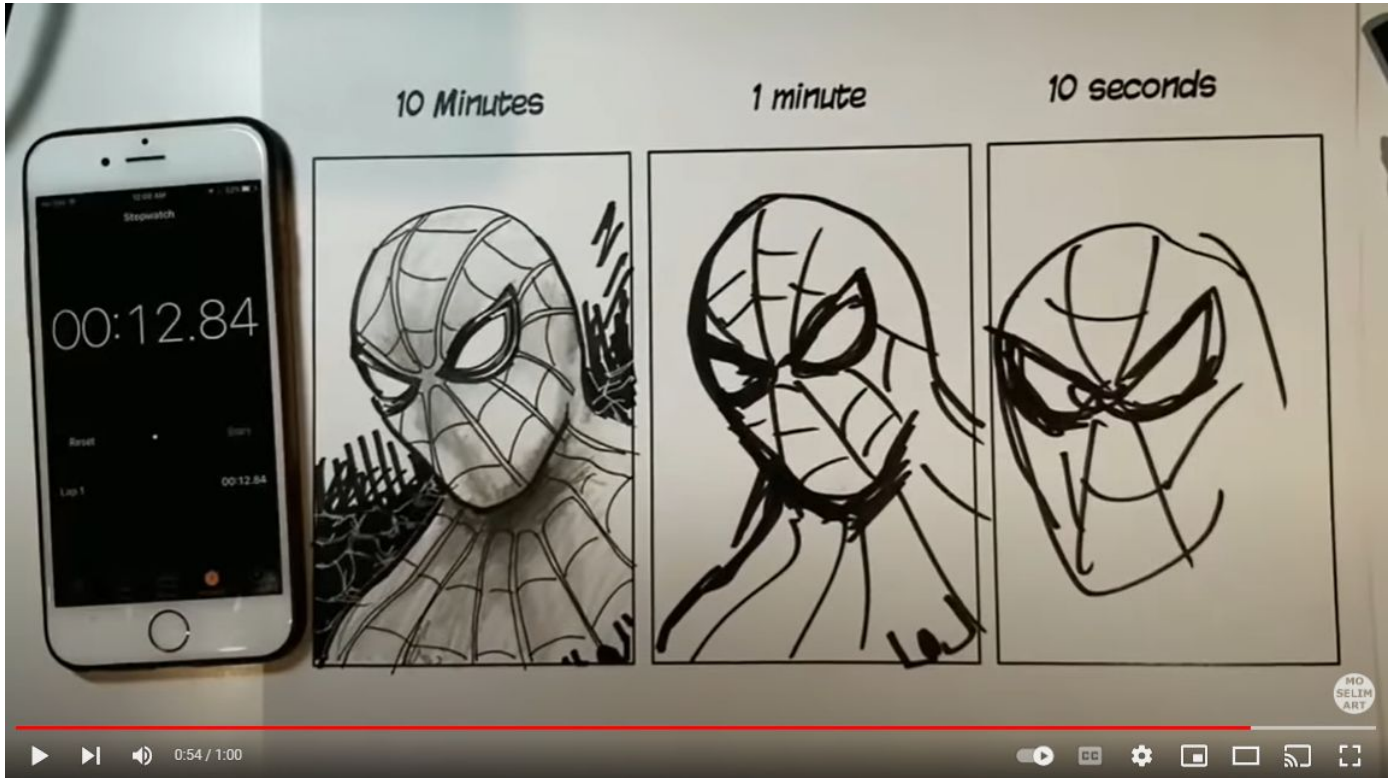
*Overengineering (or over-engineering) is the act of designing a product or providing a solution to a **problem** in an elaborate or complicated manner, where a **simpler** solution can be demonstrated to exist with the same efficiency and **effectiveness** .... - Wikipedia*

--

What is the problem? Is the problem well-understood? How would the customer describe the problem? How would a competitor or someone else describe the problem?

What is simpler? Are you the judge of simpler? Is simpler describing how it is built, how how it is extended, or simply how a user will use it? How would a CCNA solve a "basic network problem" vs. a CCIE?

What is effective? Are you the judge of saying something is effective? Are we automating your network or a customers' network?



Problem: Need a Spiderman drawing to present to a room full of aspiring children artists on what they should be able to do after attending a 2 week bootcamp.

Would a 3 minute Spider man have solved the “problem”? Would it have been as effective and simpler?
Would a professional artist not compromise and only give “great” with 10 or more minutes?
Is it best to match skill level to customer or ensure everyone understands what “simple” and “effective” coupled with understanding the “target audience?”

4 signs you're over-engineering

- 100% test coverage is not always the answer. The Pareto Principle states that 80% of the consequences come from 20% of causes. ...
- Excessive indirection and abstraction of your code. ...
- Premature use of micro-frontends and/or micro-services. ...
- Ignoring the YAGNI principle (You Aren't Gonna Need It)

https://christopherkade.com/posts/over-engineering

2. Excessive indirection and abstraction of your code

Have you ever been confronted with a bug in production, felt the rush of adrenaline while trying to fix it as quickly as possible and not being able to pin point where the exact source of the problem is? Did you have to go through layers upon layers of files, wondering why previous authors (yourself included) hid away details by placing them in some external function, four layers deep?

This is not as much of a problem when building a feature, as you're focused on creating code that is clean and reusable, but it requires a lot more mental gymnastics when trying to understand it when you're not the original author. Over abstraction can create unmaintainable & untestable monoliths, that's why I believe writing concrete code first and then abstracting is important.

If you ever review code and can't understand the underlying business logic within, that should already raise a red flag.

Always remember the Rule of Threes: an abstraction without at least three usages isn't an abstraction. Good abstraction are extracted and not designed.

4. Ignoring the YAGNI principle (You Aren't Gonna Need It)

The YAGNI principle (created as part of Extreme Programming by Ron Jeffries) states that a feature should only be developed when required and not when you foresee that you'll need it. The main point being that developers should not waste time on creating extraneous elements that may not be necessary and can hinder or slow the development process.

If you try to future proof your code all the time, you'll end up more often than not with unused code gathering dust in your repository.

Let's say you need to declare a class `User` that has a method `getAllUsers`, you might start thinking that you'll eventually need `getUserById` and `getUserByEmail` and code them right away. But that's when YAGNI comes in - you should probably reconsider it and only code it when a feature requires it, and for a couple of reasons:

- The requirements might change in the future and make you update already unused methods
- The methods might simply never be used and take up space for no reason

Just like other programming principle (KISS, DRY etc.), YAGNI is pretty straight forward

This list was inspired by this tweet from Cory House, I recommend going through it as it'll show you some other mistakes that can cause over-engineering.

Have you every been faced with obvious over-engineering in a previous project or jobs? I'd love to hear some of your stories !

Thanks for reading, I hope you got something out of this list 😆. And feel free to follow me on Twitter, I'm always happy seeing my circle of dev friends grow 🥳

>>> My Belief…

My Perception

We have brilliant engineers at Network to Code and no one wants to seem like they don't have the most technologically advanced solution.

We have Network Engineers who've converted to become NetDevOps Engineers and it naturally seems like you need to be building more and more advanced code vs. solving the problem?

We have a tendency to create requirements we don't have yet? Are we going to need all of those requirements?

My Reality

It makes you more brilliant to solve problems and create outcomes vs. having the best implementation. *Focus on the customer. Focus on the solution.*

Does a network engineer know what "good enough" is when it comes to "networking"? Does the same person have the same intuition when it comes to NetDevOps? Does the lack of experience get in the way of solving problems vs. being afraid of not knowing every detail? *Solve problems.*

Can we experience the problem first hand? It is NOT a bad thing to experience a problem before adding a feature or making a change. If we want to have zero bugs, feature requests, or changes, we'll remain stuck in Failure to Launch mode. *Experience the problem.*

>>> Delivering Outcomes at NTC

- *Focus on the customer. Focus on the solution.*
- *Solve problems.*
- *Experience the problem.*
- Ask for feedback. A LOT.
- Consult with the customer(s). Ask the “what if we did…” questions?
- Work with Pros/Cons that go beyond technology, e.g. project timelines
- Ask yourself “What features can be removed?” to satisfy the requirements?
- Mono repos are okay. They are not bad a thing.
- You don’t need to foresee many and all abstractions.
- How would you teach someone? Does that help get to “simple?”
- ***What would you demo and showcase to a customer?***


>>> network .toCode()
Thank you.