Quantity vs. Quality: Is less (code) really more?

Anne Cahalan * @northofnormal * she/her

👋

Hi! I'm Anne

Anne Cahalan * @northofnormal * she/her

I'm an iOS Engineer for a company called TeamSnap. We're taking the work out of play with apps that streamline the team sports and group activities organizational process for parents and players. I'm going to be showing some code examples of both good and bad things, and most of my examples will be in Swift--but I think the argument I'm making and the points I'm asking you to consider are pretty universal.

I ❤️ Clean Code

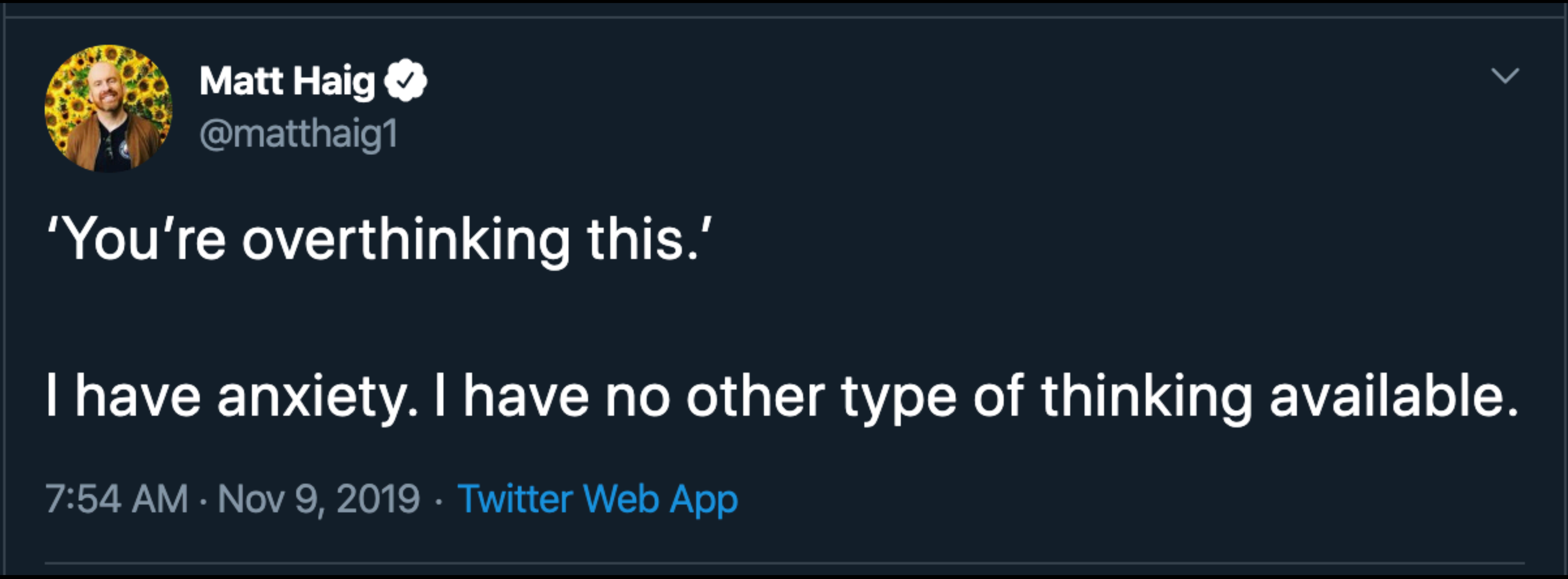
Anne Cahalan * @northofnormal * she/her

And I know this is a super controversial stance to take, but I am big proponent of clean code. I've spent a lot of time over the last couple years really thinking about what it means for code to be clean? What do those SOLID principals mean, and why do we care about them?

SOLID

- Single - repsonsibility principle
- Open - closed principle
- Liskov substitution principle (♥️)
- Interface segregation principle
- Dependency Inversion principle

quick recap, since I'm going to be referring to SOLID principles a couple of times this morning: SOLID is an acronym for five design principles that, I believe, Bob Martin promulgated. In short, the S is for single-responsibility, the idea that a class or a method should do only one thing; O for open-closed, the idea that things should be open to extension but closed for modification; L is for Liskov Substitution--that objects should be replaceable by their subtypes--I'm an unapologetic Barbara Liskov fangirl, so this is my fave of the five; I is for interface segregation, which is the principle that specific interfaces are better than general ones; and D is for dependency inversion, the idea that you should depend upon abastractions, not actual details. There's more to all of these, but that's another talk.


Matt Haig
@matthaig1

'You're overthinking this.'

I have anxiety. I have no other type of thinking available.

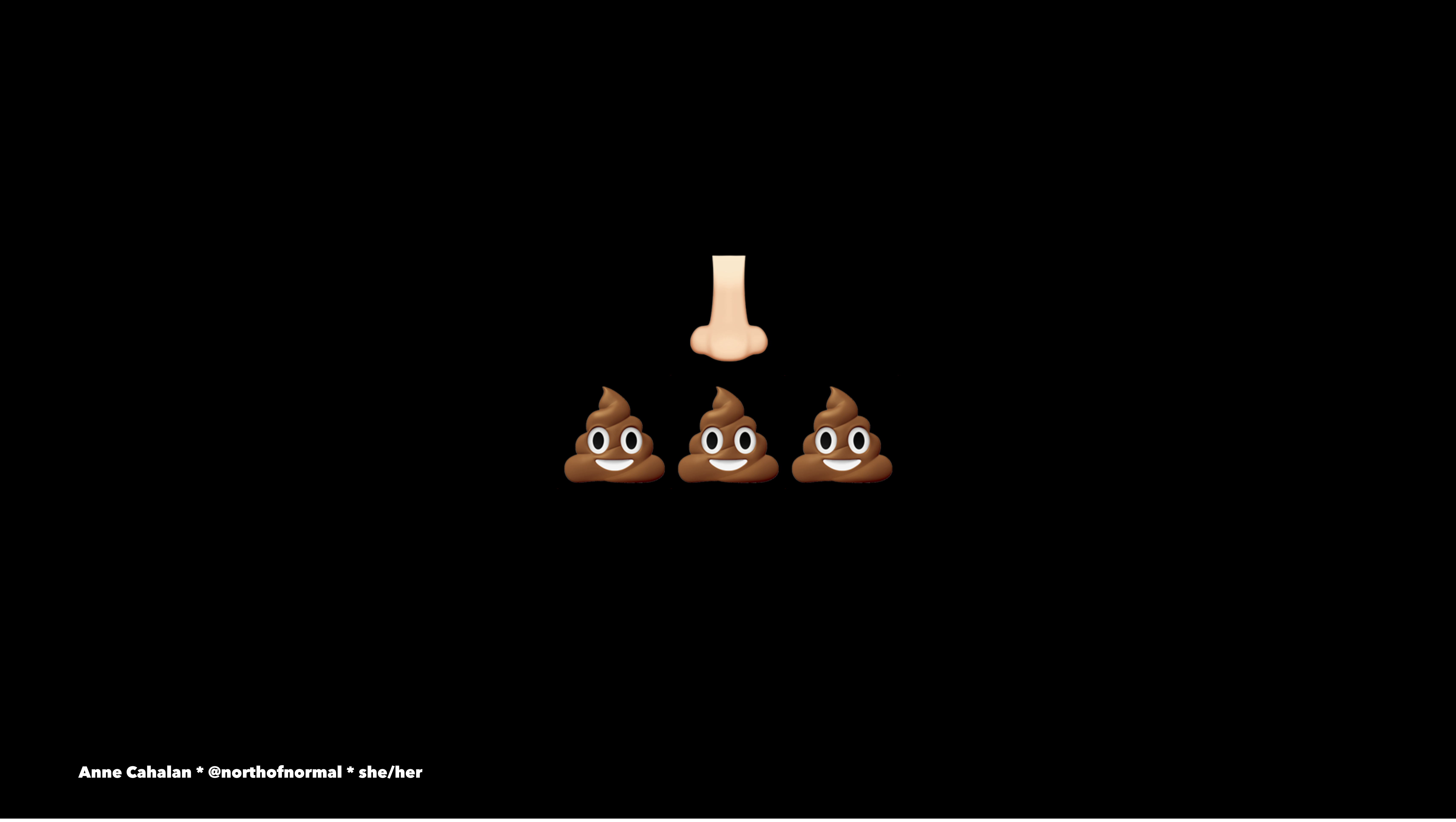
7:54 AM · Nov 9, 2019 · Twitter Web App

Some might accuse me over overthinking things, but those people probably don't have anxiety disorders, so those people can just go be smugly well-adjusted somewhere else. I spend a lot of time staring at perfectly cromulent code, code that's testable and digestible and that works great, and wondering, "but how can I make this better?"

Code Smells

- Duplicate Code
- God objects
- Cyclomatic complexity
- Long methods

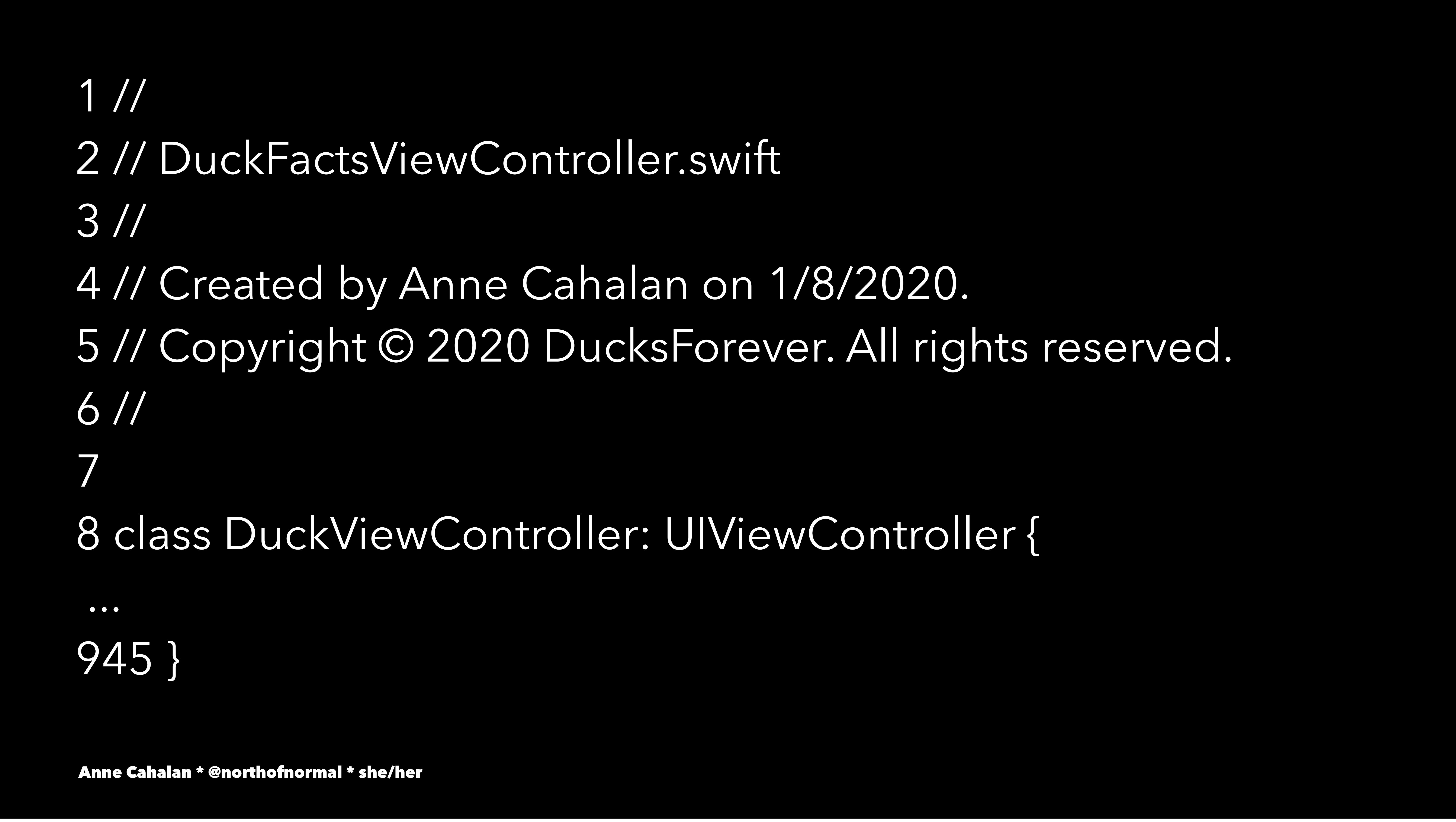
and one thing that you can always look out for, if you are me, are code smells--those things that don't make your code *stink* exactly, but definitely give off a suspicious odor. The common code smells that you always hear about all seem to involve too much code: Duplicate code violates that DRY (don't repeat yourself) principal, that gets coupled with SOLID a lot when giving or getting advice on how to keep your code clean. So you don't want a lot of duplicate code--code doing the same thing, in different places. You don't want one god class that does everything. We don't put all our code in one giant method that runs on launch that's like, "run the app". Cyclomatic complexity is fun, because it's measurable--it's a count of the various paths that can be taken through your code. What are all the possible end results of one method call? That should be a low number. Long methods, ones that are doing a lot of things, almost always violating single responsiblity and almost always have too high of a cyclomatic complexity--too many possible paths through and outcomes.



But why do these things smell? What's so bad about them?

- 💩 Not DRY
- 💩 Single Responsibility Principle violations
- 💩 Hard to test
- 💩 Complexity sucks

DRY code, code that has abstracted out common tasks from across the app into one external method, is easy to change. If all the cancel buttons need to be gray and disabled when the user isn't logged in, why make each button gray and disabled individually? Why not have a cancel button that you can use all over the place? If you suddenly need to make them enabled but have them make a duck noise...do you want to change that once, or try to hunt down every instance of a cancel button? If we accept that the SOLID principles are a good goal, and for the purposes of this talk we will, then a god object that does everything is certainly violating that. Also, long methods that do eight different things, large classes, code that's scattered all over doing the same thing in different places, all that is hard to test. Because, honestly, complexity sucks. Complexity is hard, complexity means you are staring at your code for ten minutes trying to figure out what's going on.



```
1 //
2 // DuckFactsViewController.swift
3 //
4 // Created by Anne Cahalan on 1/8/2020.
5 // Copyright © 2020 DucksForever. All rights reserved.
6 //
7
8 class DuckViewController: UIViewController {
 ...
945 }
```

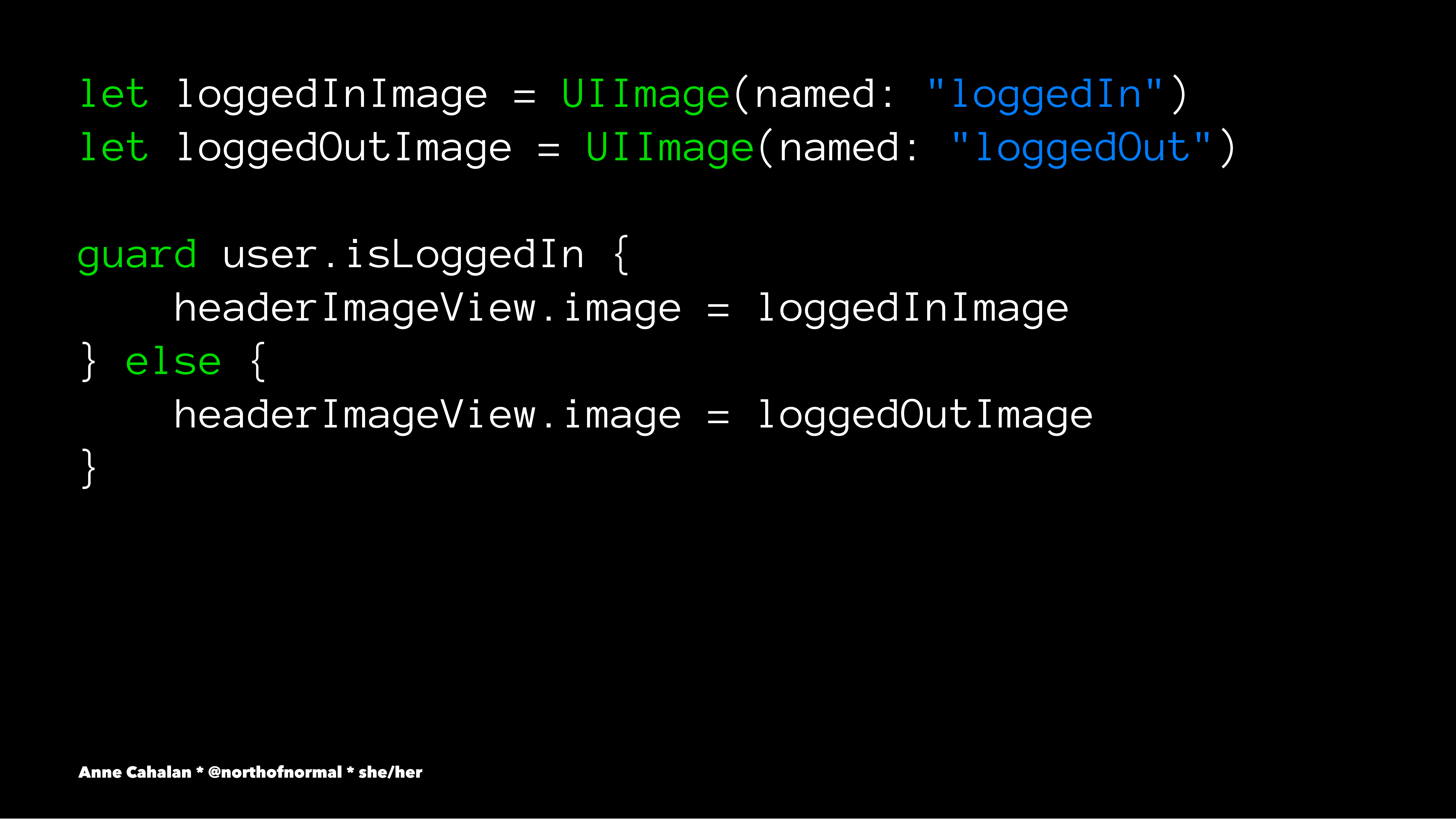
And we know this, because we've all opened a file that looks like this--the stereotypical massive view controller that's over 900 lines for some reason, and you open this and start scrolling through it and just...viscerally hate this class, and hate the moron that wrote it, and then you realize you are the moron and wow does that not help. Immediately when you open this file you just want to start taking a weed whacker to it. Forget whatever you are here for, before you do anything you have to start slashing and burning.

👩‍⚖️ Too much code is bad 👩‍⚖️

Okay, cool, so we are all agreed, too much code is bad code, high five, let's go hit the lazy river. Weelllll.....

📖 Story time 📖

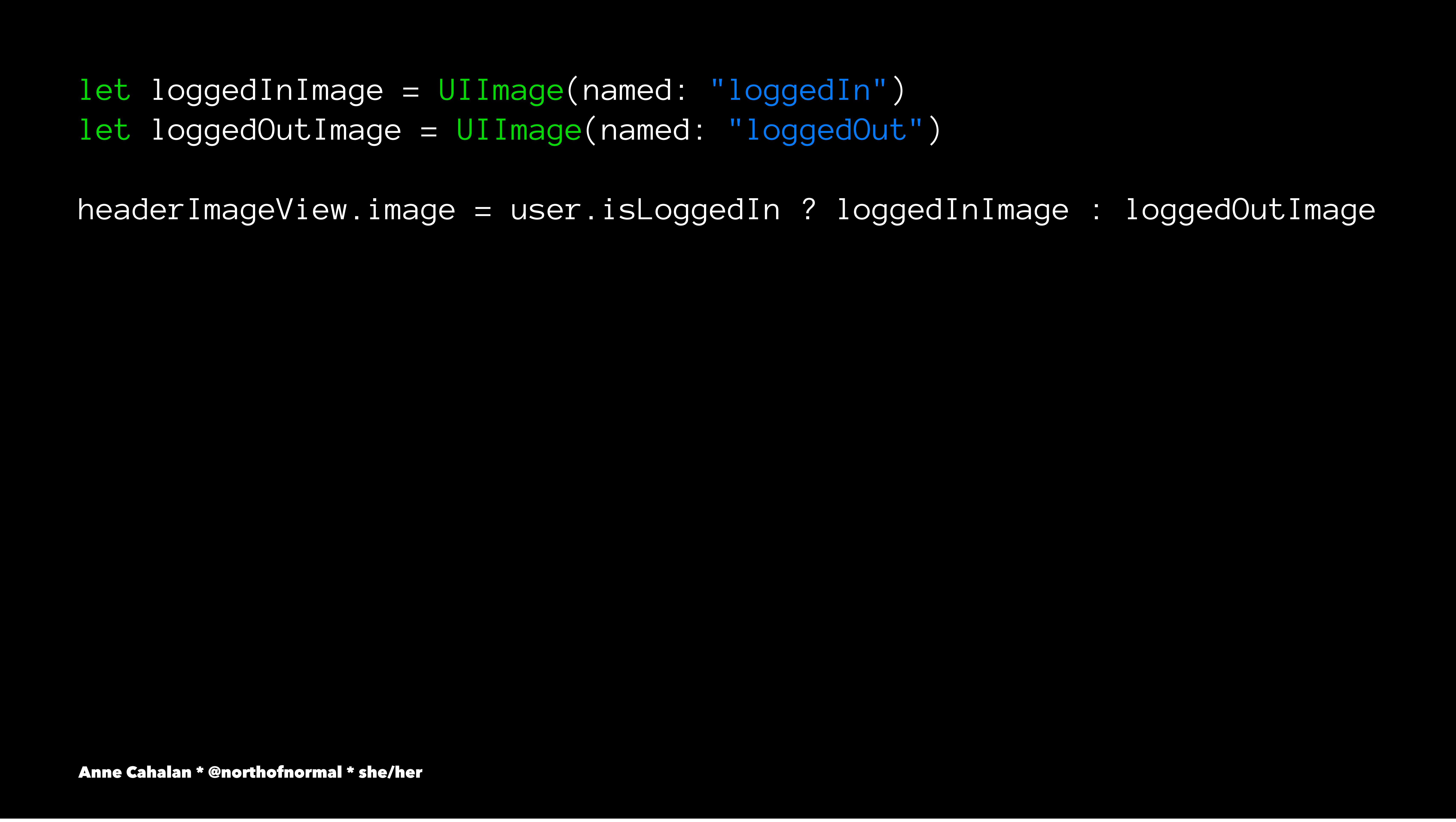
So this summer I was on a team with some other developers, and I'm not gonna throw them under the bus, they were great and smart dudes. But, the project they had been on before I showed up was one where the 900-line vcs were the norm, and they had a little PTSD. They were constantly on the quest for *less code*. Removing lines of code was an unalloyed good, if you could do something in ten lines, you could probably cut that down to eight. If you could do it in eight, why not six? (WHY NOT)



```swift
let loggedInImage = UIImage(named: "loggedIn")
let loggedOutImage = UIImage(named: "loggedOut")

guard user.isLoggedIn {
    headerImageView.image = loggedInImage
} else {
    headerImageView.image = loggedOutImage
}
```

So one day, I put up some code like this, and my buddy made a face. What about this, he says:

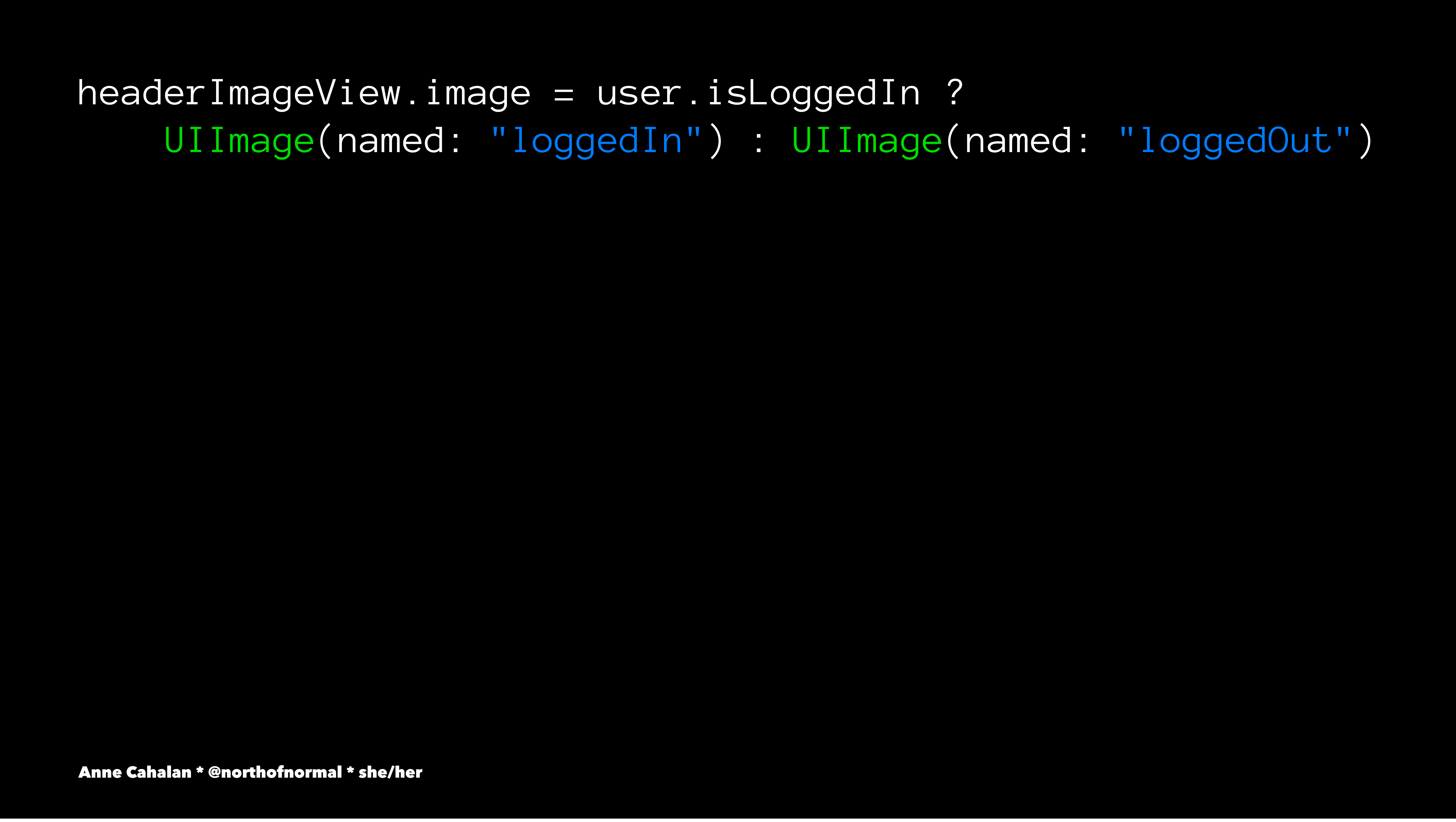


```swift
let loggedInImage = UIImage(named: "loggedIn")
let loggedOutImage = UIImage(named: "loggedOut")

headerImageView.image = user.isLoggedIn ? loggedInImage : loggedOutImage
```

Anne Cahalan * @northofnormal * she/her

So, okay, that's fine I guess. I'm generally anti-ternery but this isn't so bad. But then, he got all excited and did this:



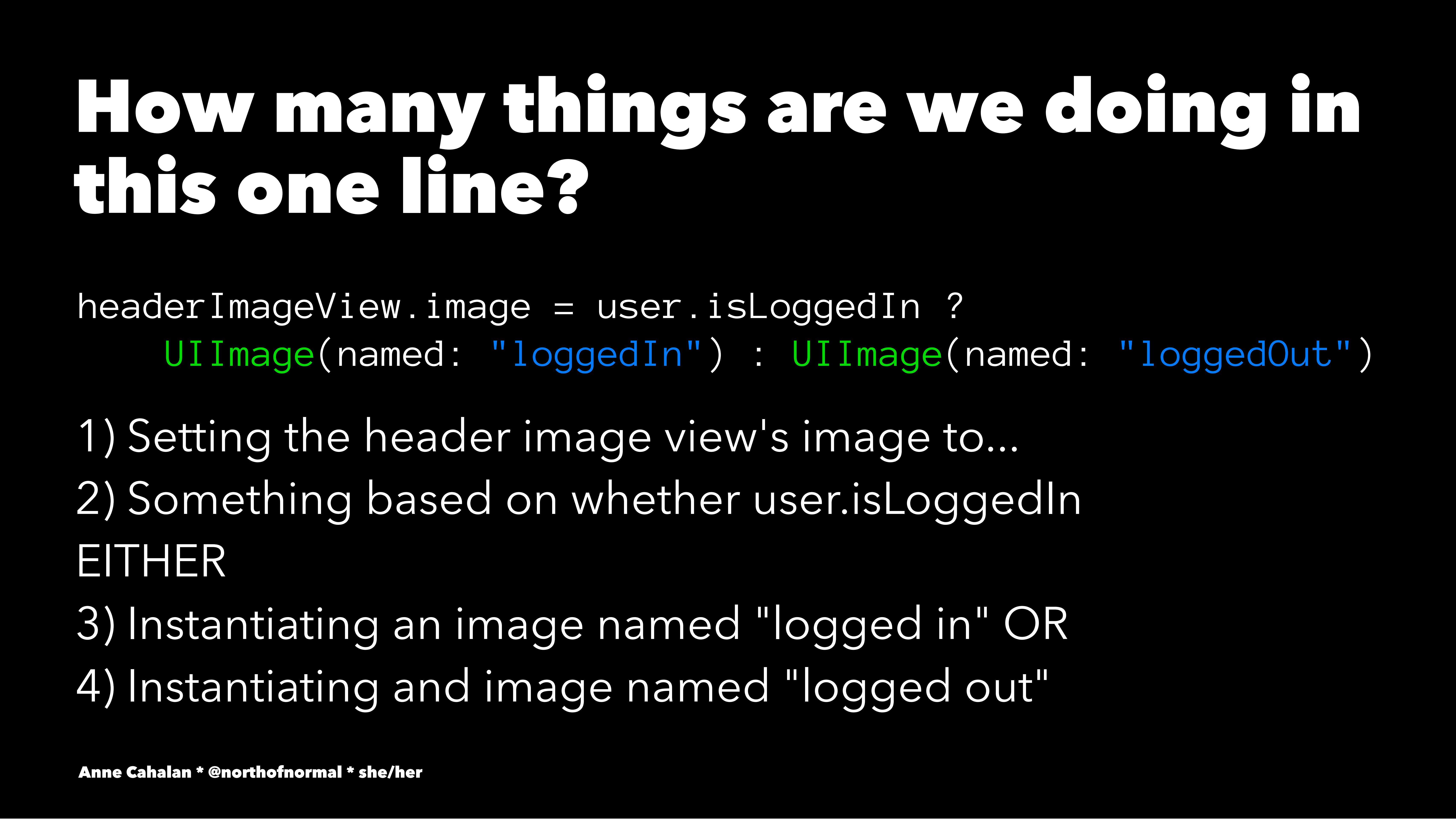
```
headerImageView.image = user.isLoggedIn ?
    UIImage(named: "loggedIn") : UIImage(named: "loggedOut")
```

And her turned to me and said, "see, we got rid of five lines of code this way!"

And my face did this, and thus this talk was born. Because in that moment, I realized something.

Too little code is just as bad as too much.

You know I mean it, because I punctuated it. And this example wasn't the worst I've seen, but it's pretty bad. Sure, we lopped off five lines of code, but...why? What did we gain, other than *less code*_? Did we reduce repitition? Not really. Did we reduce complexity? Not at all, in fact, we added complexity.



How many things are we doing in this one line?

```
headerImageView.image = user.isLoggedIn ?
    UIImage(named: "loggedIn") : UIImage(named: "loggedOut")
```

1) Setting the header image view's image to...
2) Something based on whether user.isLoggedIn
EITHER
3) Instantiating an image named "logged in" OR
4) Instantiating and image named "logged out"

Anne Cahalan * @northofnormal * she/her

We're setting the header image view's image to something based on whether user.isLoggedIn is true, and then either instantiating an image named "loggedIn" or an imaage named "logged out". That's...kind of a lot of things to be doing. And, let's be real, that's not all we're doing, we're also trying remember how terneries work, is the first one true...? Ternaries in and of themselves don't scan easily, right? You are bopping along, reading your code, this does this, okay, we're passing this paramater here and returnin an array, got it...I don't know about you, but a ternary is a speed bump. I have to stop and parse how the stupid thing even works.

Bold Statement:
Ternaries can be a code smell

There, I said it. They are, to me, indicative of a code smell that we don't talk enough about: Excessive Cleverness. They are a code flourish, their purpose is almost wholely decorative. They are the three loops under John Hancock's signature. They serve more to demonstrate how clever the code author is than they do to provide meaning and guidance to the next developer who has to interact with this code.

We need more code

```
let loggedInImage = UIImage(named: "loggedIn")
let loggedOutImage = UIImage(named: "loggedOut")

headerImageView.image = user.isLoggedIn ? loggedInImage : loggedOutImage
```

That said, I will concede that my first example, with the verbose if-statement was bit much. Ternaries can serve a purpose and CAN provide guidance in our code. But, by the simple expedient of extracting out those images into their own variables, we've made this very complicated speed bump of a ternary into something much easier to comprehend. By giving our images good and descriptive names, we've helped make the ternary easier to understand when scanning the code.

The problem wasn't (only) the ternary

to be fair, the problem with my example above wasn't entirely the fact that ternaries can be confusing. I think the real problem I was railing against is something else entirely: inline instantiation.

This is gross

```
databases.add(database: MySQLDatabase(config: MySQLDatabaseConfig(
            hostname: "localhost", username: "vapor",
            password: "password", database: "vapor")
), as: .mysql)
```

Here's a bit of code that I stole from my Server Side Swift precompiler yesterday. Or, I should say, this could have been code we used, but I like the people who come to my precompilers and I want to be nice to them. How many things are we doing in one line here? We're adding a database that we are making as a MySQLDatabase createed with a MySQL Database Configuration that has a bunch of its own properties. That's, what, three things at least we are doing in one line of code? That's...too many things. You can't scan this code, you can't glance at it and know what it's doing. You need to stop and parse each of those instantiations of a database, a configuration, and how they nest into each other.

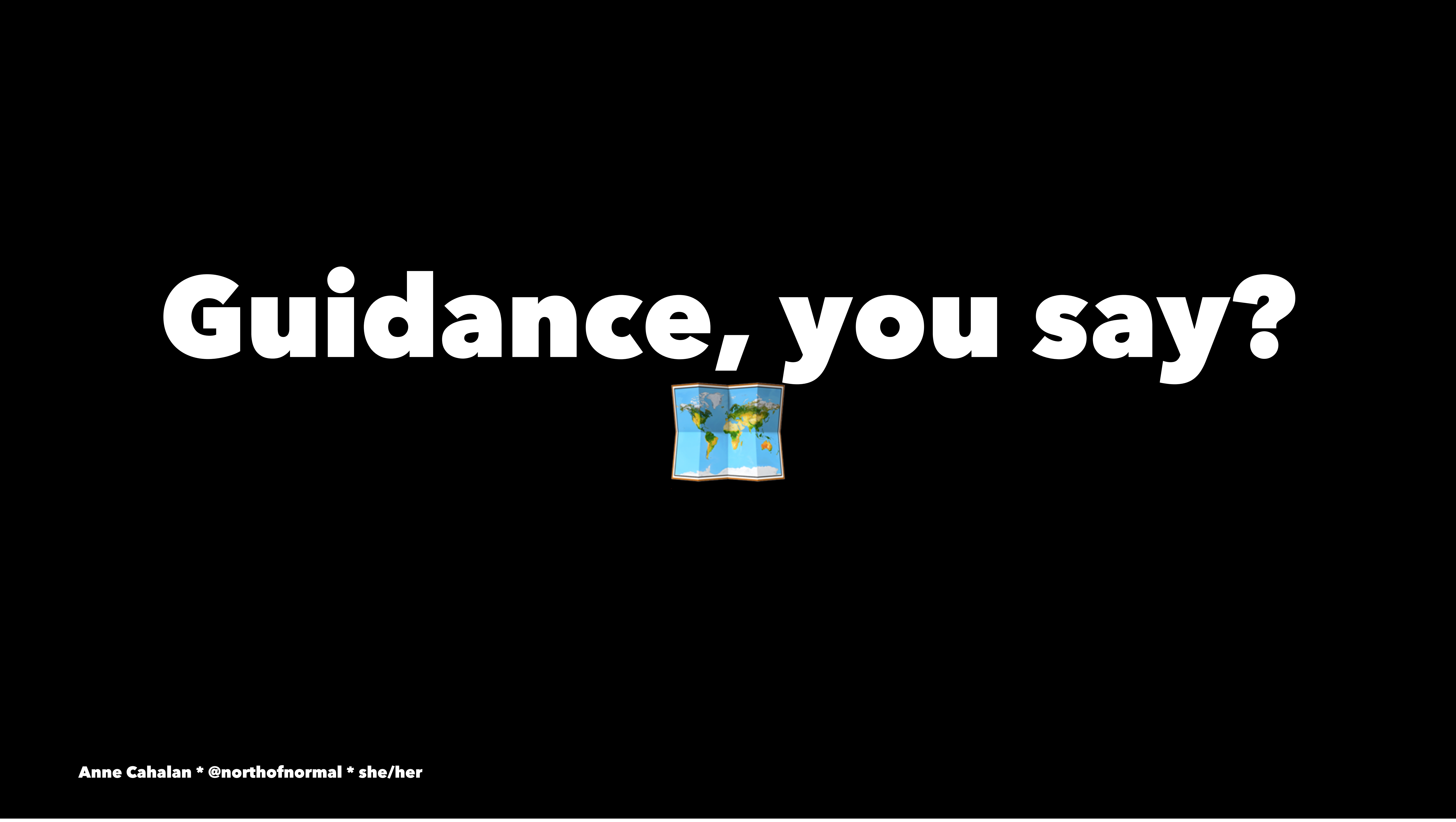
This is better

```
let mySQLConfig = MySQLDatabaseConfig(hostname: "localhost", port: 3306, username: "vapor",
                                                 password: "password", database: "vapor")

let mysql = MySQLDatabase(config: mySQLConfig)

databases.add(database: mysql, as: .mysql)
```

This is the code that we actually wrote in that precompiler. And what's happening here? We're creatinig a configuration, we're creating a database with that configuration, and we're adding that database. It does the exact same thing as the gross do-it-all-once example, but when we look at this code, we can take it in the correct sequence--make a thing and pass it into another thing--top to bottom instead of somehow inside-out. This code guides us to its intention, adding a particular database, very gracefully.



Guidance, you say?

🗺️

Indeed, I say. For whom do we write code? Why do we do this, in this particular way? The computer doesn't care. When you get down to the level where the stuff actually happens, a device doesn't understand Ruby or Java or Swift or Erlang. It understand on and it understands off. It gets or does not get an electical impulse. Code is the human-readable, human-understandable symbolism that we use to eventually tell a computer a series of ons and offs. And since code is meant to be human-readable, it should be, you know, HUMAN READABLE.

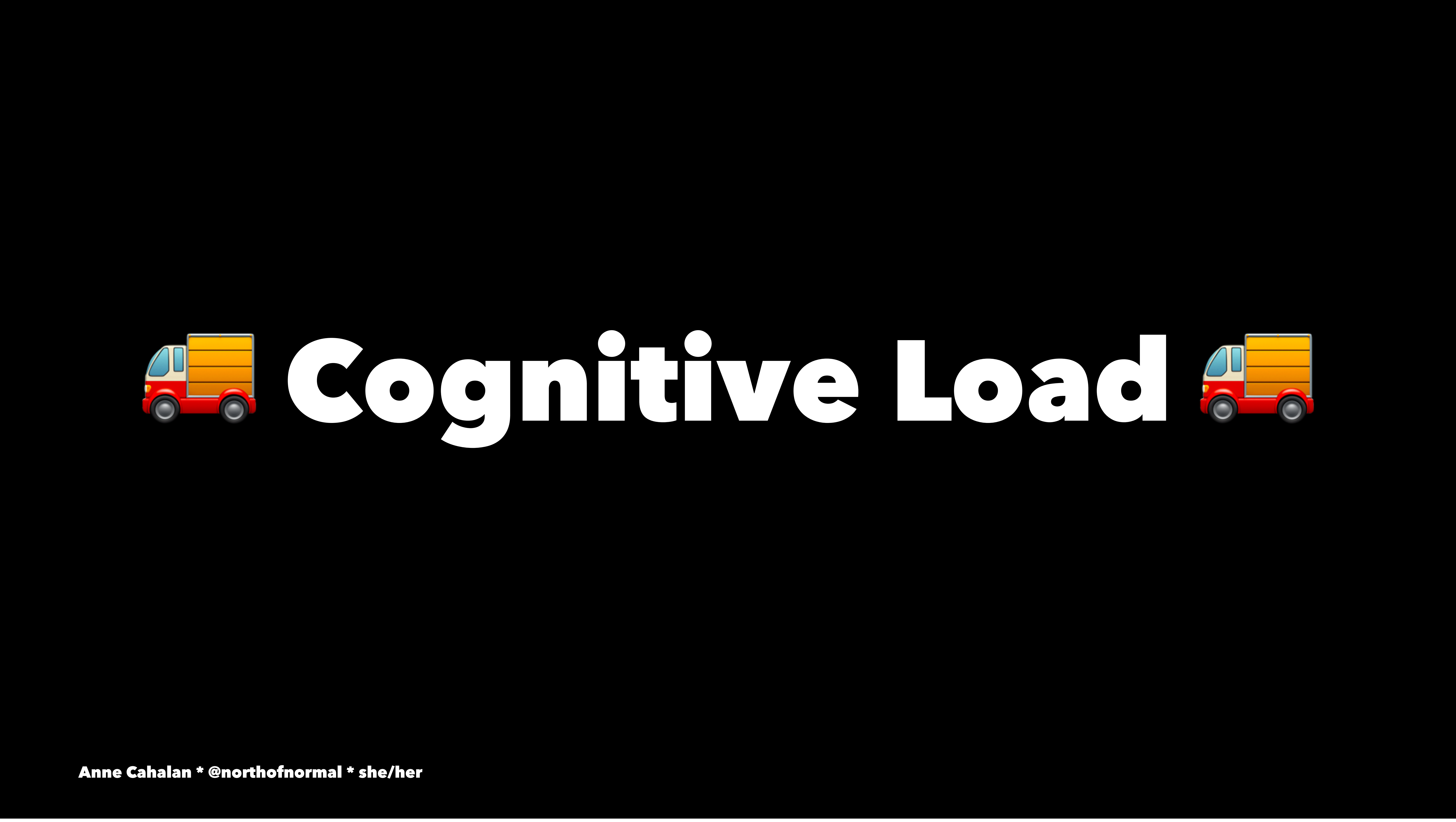
"Code should read like sentences"

Anne Cahalan * @northofnormal * she/her

This is a bit of advice I got early on, and one that I've held on to. Your code tells a story, it tells you what is going to happen to your data as moves through a paritcular path. How it changes, and how it causes change. That's a little woo-woo for an 8am talk, so I'm not going to go too far with this metaphor. But think about that database example, even as I was describing the code, one way was very clear: we're making a configuration, we're making databse with that configuration, we're adding that database. That's a very straightforward, picture-book level of story. The other version, well, that sounds less like a kindergardener story and more like the stories Drunk Me Tells: OK. We're adding a database, yeah, we need to make a database but the database needs a configuration...

New people read our code stories all the time

Like I said, the code story thing is a little woo, but let's stick with it. New people are coming to this story all the time. Very few of us work in an utter vaccuum. We have coworkers that we are collaborating with, new people onboard onto the project, heck, you write some code that makes sense to you one day then you go off and go write some other code and maybe a few weeks pass and in those weeks you go to the movies and pet a dog and eat dinner and, you know, have a life, and you come back to code that...isn't giving you enough information anymore. It's not telling you a coherent story. The story it's telling it is too complicated for the new junior guy you've brought in or the tech lead with ten years of experience but she's been working on something else for a while.



🚚 Cognitive Load 🚚

Anne Cahalan * @northofnormal * she/her

I want to talk a little about the idea of Cognitive Load. It's a concept from psychology, specifically instructional design, that refers to the used amount of human working memory resources that are engaged at a given moment. High cognitive load, simply, is when you are thinking about a lot of things; low cognitive load is when you can think about fewer things. It is literally the load on your cognition. If you've ever heard that thing about how you can only think about seven things at a time? I don't think that's science, but that's the idea.

Low Cognitive Load 👍🏻
High Cognitive Load 👎🏻

Anne Cahalan * @northofnormal * she/her

The entire field of instructional design would set themselves on fire looking at this slide, but for our purposes, this is what you need to know. High cognitive load situations lead to errors, faulty thinking, misplaced assumptions, and general disaster. If your code carries a high cognitive load, you are going to create bugs. The science is a whole lot more complex than that, but this is essentially the argument I'm making today. Complex code--whether it's lots of it or a little of it--is going to be buggier.

An Example

```swift
struct User {
  let id: Int
  let name: String
  let email: String?
  let role: Role
  let companyName: String
  let friends: [User]
}

extension User: Decodable {
  static func decode(_ json: JSON) -> Decoded<User> {
    return curry(User.init)
      <^> json <| "id"
      <*> json <| "name"
      <*> json <|? "email"
      <*> json <| "role"
      <*> json <| ["company", "name"]
      <*> json <|| "friends"
  }
}
```

This gnarly bit of business is from a Swift JSON decoding library called Runes, it was created by Thoughtbot, and when I was a junior developer, the senior dev I was paired with on a particulr project decided we were going to use this handle our network responses. And I was the junior dev, so what was I going to say about it? At the time, I could barely understand the normal, sane-swift-dev way of decoding JSON. This...was nonsense to me. It was a magical spell. I'm still not sure I could explain to you have it works and what it does. Imagine being me, junior dev, trying to learn Swift, trying to learn architecture patterns and algorithms and how protocols work and eight million other things.

And it sure was fancy, wasn't it? It was a very...clever...solution to a common problem. It didn't follow a single pattern relative to anything at the time in Swift, it used a bunch of ornate decorators that clearly had meaning, but offered no guidance whatsoever. This wasn't so much a speedbump as a complete stop sign. And my senior dev buddy couldn't stop talking about how much EASIER it was than the longer, more verbose, more descriptive and obvious alternatives.

Simple vs Easy, Complex vs Hard

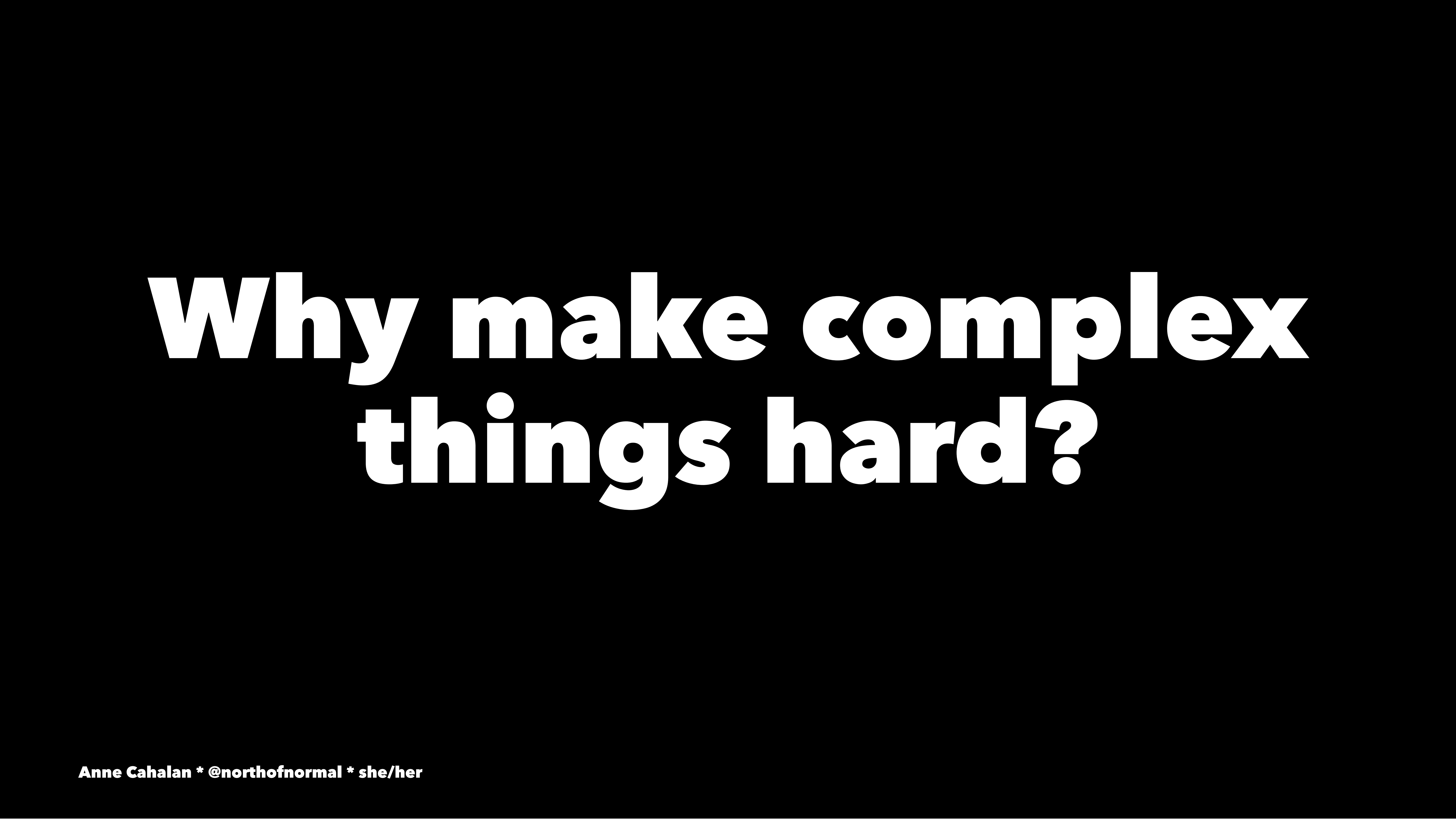
Anne Cahalan * @northofnormal * she/her

I think that goes to something that I've been circling around throughout this whole talk, which is the differences between simple and easy, complex and hard. Simple and Complex are concrete things, measurable, even--we mentioned cyclomatic complexity earlier. We can straight up measure that kind of complexity if our code and give it an acutal number. Some linters do this and flag methods that thave a complexity score higher than a certain threshold. Easy and Hard, though, those are relative. Something is easy or hard depending on your perception of it, your experience with it, whether you've ever done anything like it before. This sweater is a great example, because I made it myself. Like it? It was easy. No, really, it was easy--it's basically a rectangle with two arm holes and a bunch of cables, and oh did I mention I've been knitting for twenty years? Yeah. If someone with less experience knitting tried to make this, it would be the same size and have the same number of cables and require the same number of stitches--it would be, for both of us, equally COMPLEX. But for me it was easy and for someone else it might be hard.

Complex can be easy
Simple can be hard

Anne Cahalan * @northofnormal * she/her

This is why places like Turing Acadamy have rules against calling things "easy" or saying "you can just do..." when people are learning. Easy is relative, and "just" is dismissive. And if these things are true, their inverse is also a true, and that's a warning.



Why make complex things hard?

Anne Cahalan * @northofnormal * she/her

In a blind pursuit of cutting down our code to fewer and fewer lines, we run the risk of writing code that is both complex and hard for others to comprehend--hard even for ourselves to come back to after some time has passed. If we have a chance to take something complex and, by abstracting out bits into descriptive and meaningful variables, by taking some extra time and space to make clear our intentions, we can make that complex thing easier--we should definitely seize that chance whenever we can. Life is hard enough without our code making it harder, right?

What does this have to do with empathy?

Great question, and I'm glad you asked! Empathy is another thing we've been circling around in this talk, I think. Among other things, empathy is a practice of being kind. Clean, clear code that takes the time explain iself in human language is kind to not just to colleagues who may be encountering your code for the first time, but it's kind to yourself as well. You are always going to come back to your code after some time has passed, when you are far removed from the moment of writing it, and have to recreate what you were thinking and what your intentions were. Some of those times, you are going to be trying to find and fix a bug and, let's be honest, programming is way easier than debugging is. If you have provided guidance with your code, instead of placing speedbumps, if you have ensured that your cognitive load will be light as you move through your code, if you have taken steps to decrease complexity or to at least make complexity easier to navigate...that is a great kindness to everyone who will encounter your code, including future you.

Thanks!

Anne Cahalan * @northofnormal * she/her

Thanks for waking up this morning and going on this journey with me. I hope you've gotten something out of it. This is the conference debut of this talk, so if you have any feedback or suggestions, please let me know. I'm @northofnormal on twiter and just about everywhere else, I'd love to hear your thoughts on how much code is too little--and how much is too much, so feel free to get in touch. Enjoy the rest of the conference!