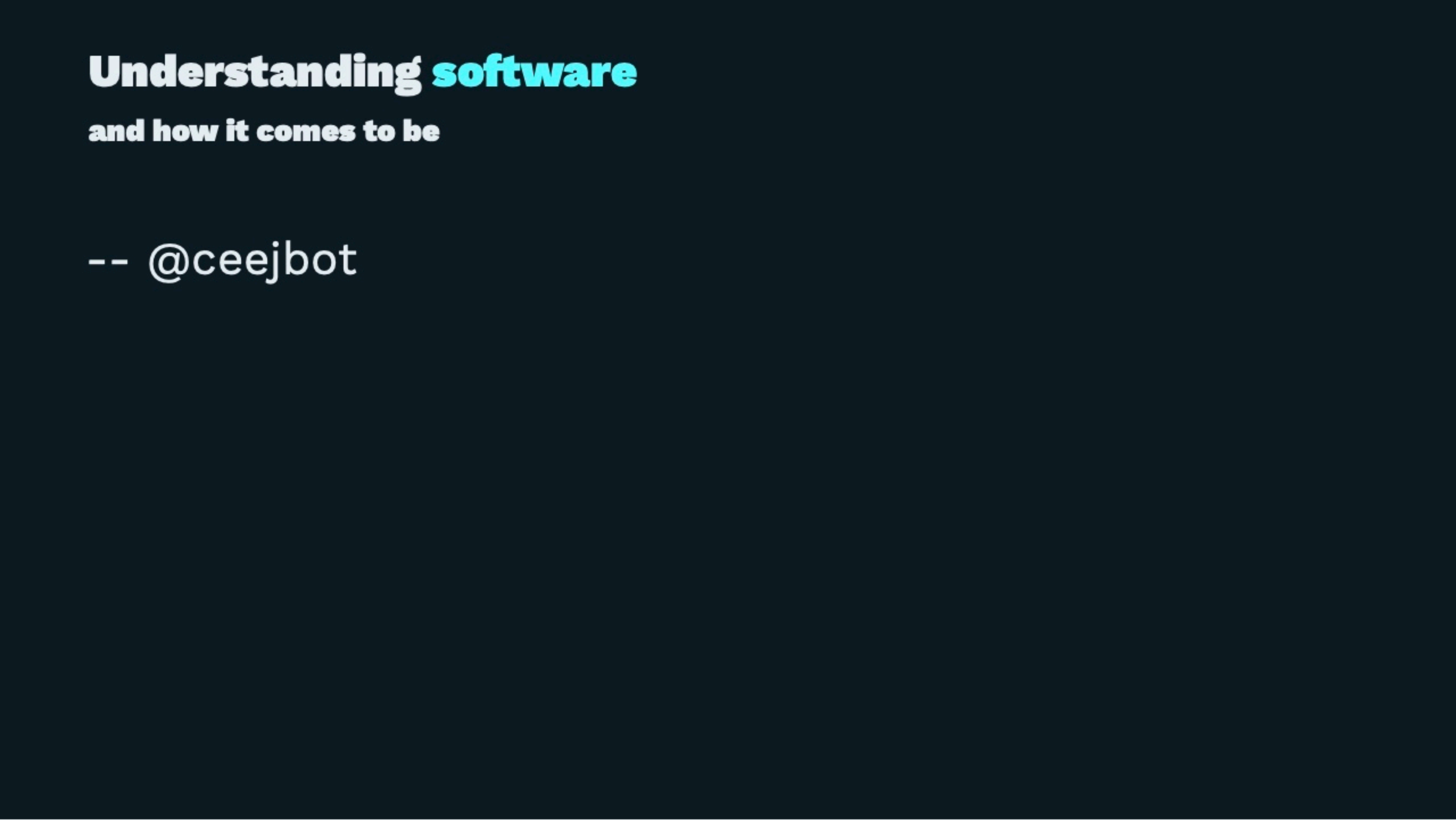
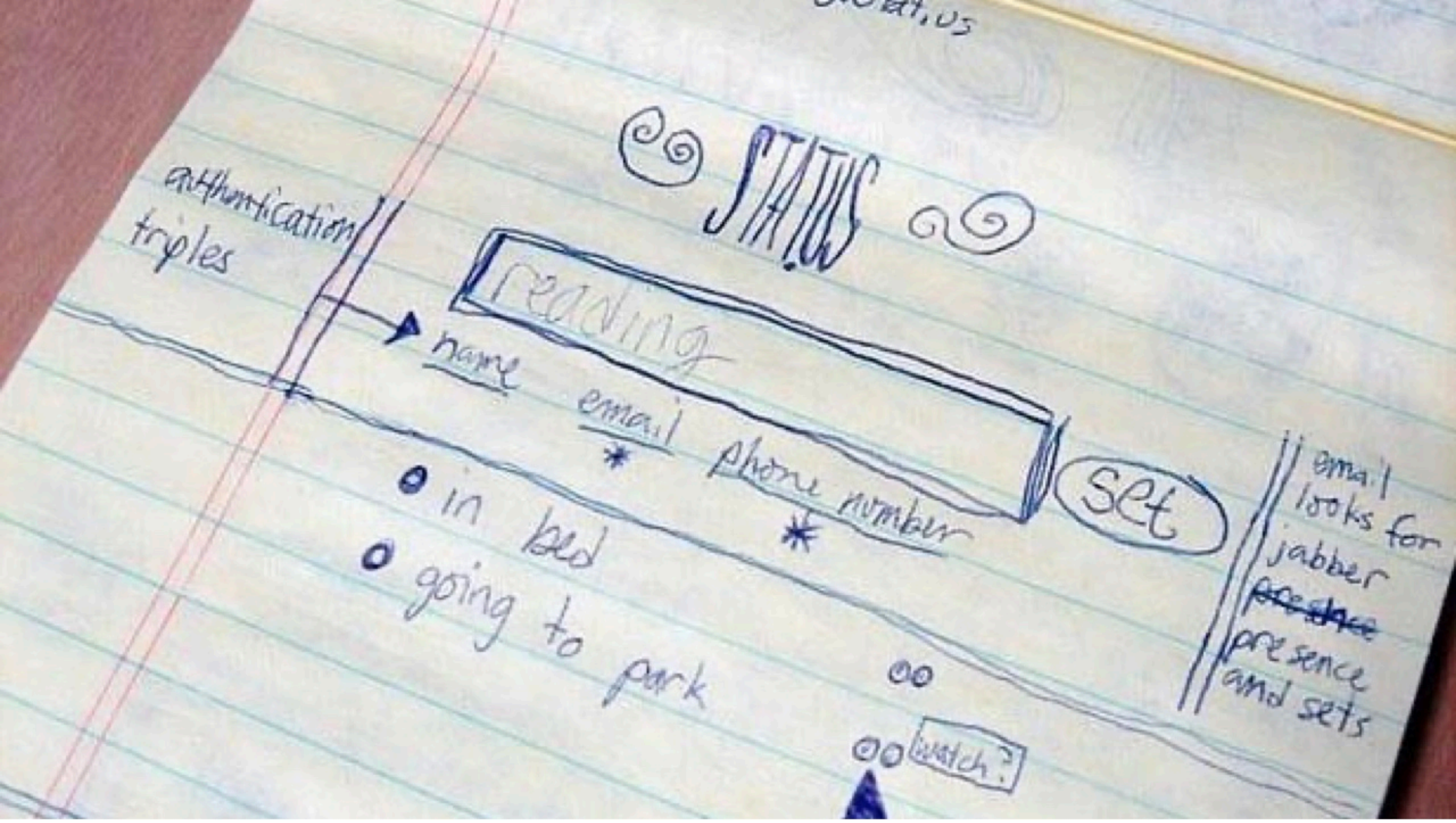


Understanding software

and how it comes to be

-- @ceejbot

A note about the slides: they're anchor points to call out important words or to remind you where we are in the presentation. You don't have to let them fill a whole screen if you don't want to. There aren't any flashing lights or animations in the presentation, either.



How does this sketch turn into a company that had thousands of developers, millions of daily users, and an effect on the entire world? At its starting point was nothing, and then software happened, and for about 15 years we had *something*. Politics, news, culture-- all happened because of this software. I've always found this amazing-- somebody has an idea, and this THING appears out of nothingness.

What is **software**?
How do we **build** it?
What happens **afterward**?

Carl Sagan said if you want to make an apple pie, first you must invent the universe. We aren't going to go that far back, but we are going to talk about these three questions. I need to caveat all of this: My answers to these questions come from a specific perspective-- me & my career experiences. I am not going to talk about how it's done at Google or Facebook or other weird gigantic companies. Going to talk about how software is built by small to medium sized teams, in the Silicon Valley, ones that happen to have a lot of ex-Apple product influence.

This company writes software

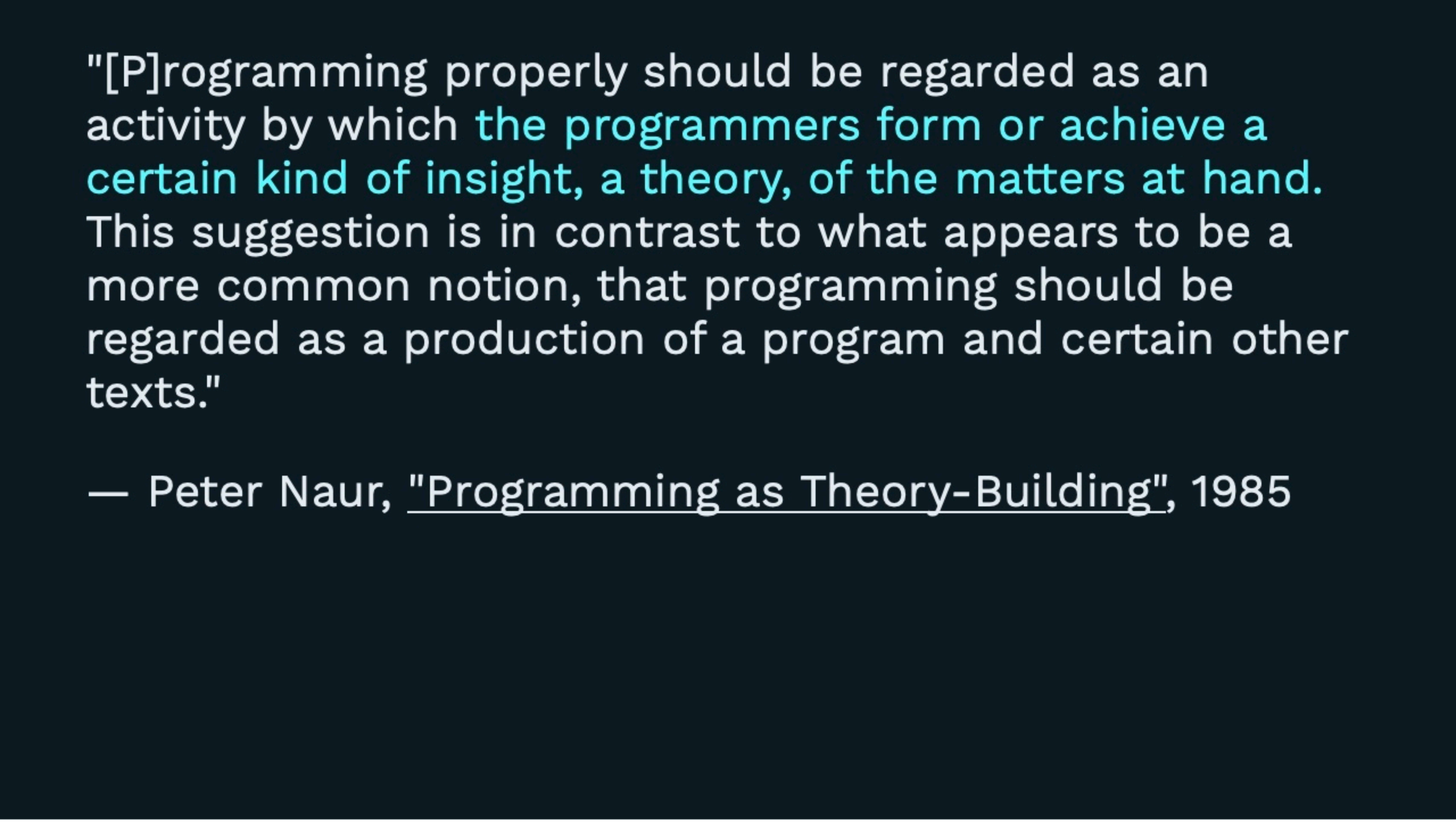
— Everyone here contributes to this work.

— Everyone here would benefit from understanding how we do it.

We write a lot of software.

What is programming anyway?

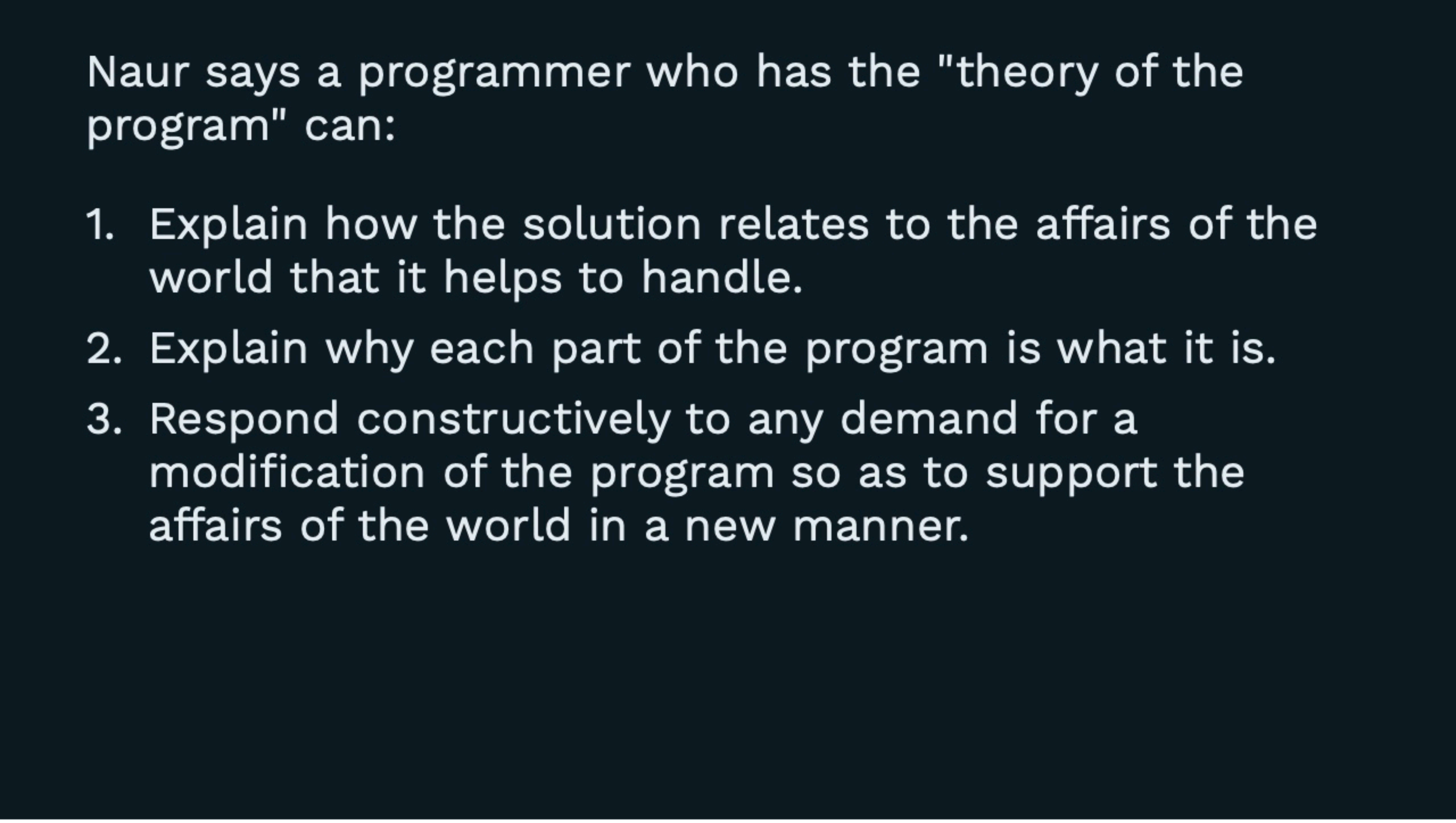
A traditional answer is that programming is typing long text files with instructions to make a computer do things. But when I'm sitting with my feet up on my desk, or when I'm pacing around my house muttering, or when I'm scribbling in that notebook, I'm also programming. I'm going to go to one of my favorite essays of all time for another answer.



"[P]rogramming properly should be regarded as an activity by which the programmers form or achieve a certain kind of insight, a theory, of the matters at hand. This suggestion is in contrast to what appears to be a more common notion, that programming should be regarded as a production of a program and certain other texts."

— Peter Naur, "Programming as Theory-Building", 1985

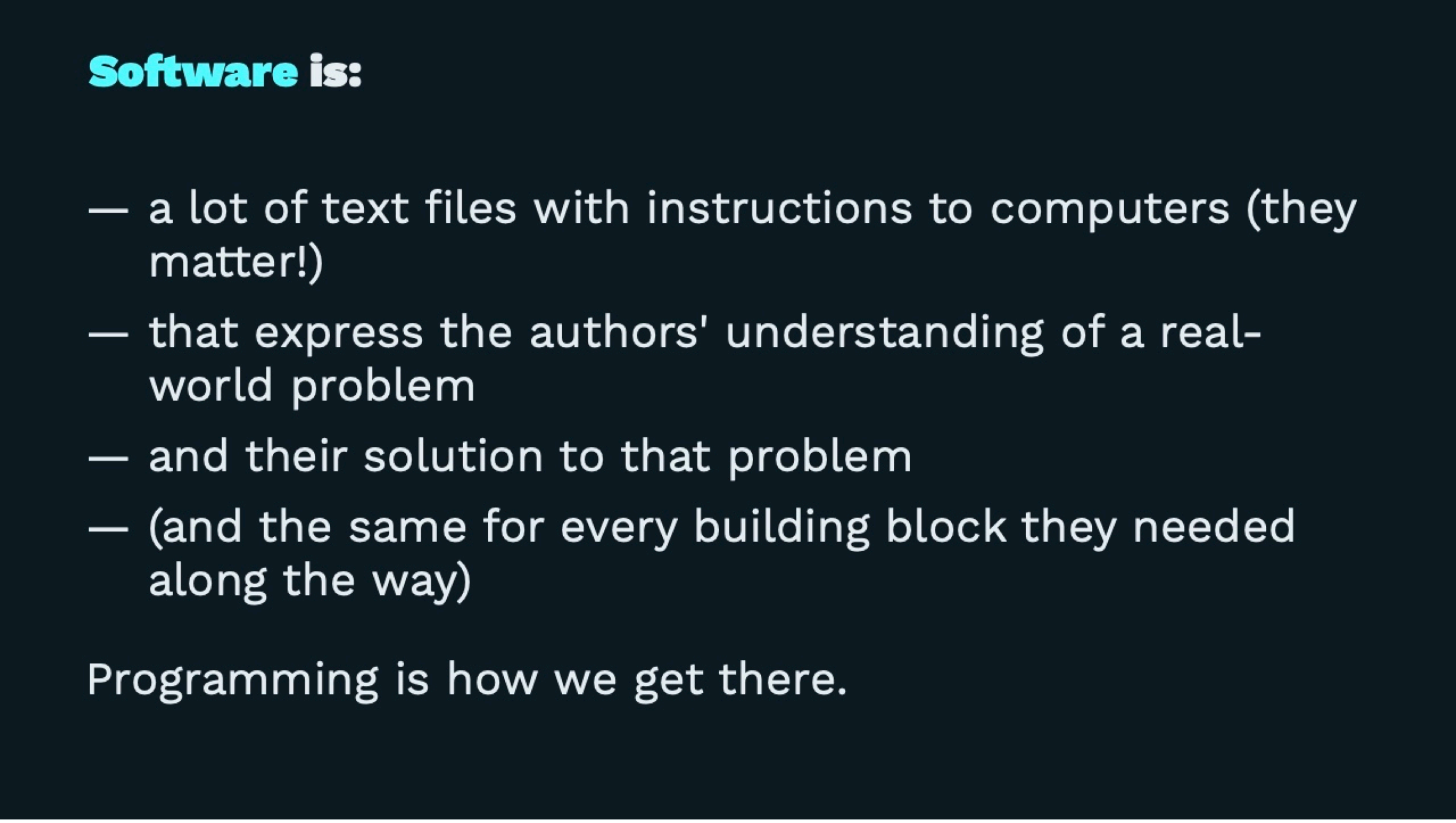
Peter Naur is the Naur of Backus-Naur Form, which some of the programmers in the audience might remember, and one of the designers of Algol, the extremely influential programming language. This is from a 1985 essay about what he'd learned about how to write and maintain and operate software. I think this is right on target. Let's spend a moment looking at Naur's theory of the program.



Naur says a programmer who has the "theory of the program" can:

1. Explain how the solution relates to the affairs of the world that it helps to handle.
2. Explain why each part of the program is what it is.
3. Respond constructively to any demand for a modification of the program so as to support the affairs of the world in a new manner.

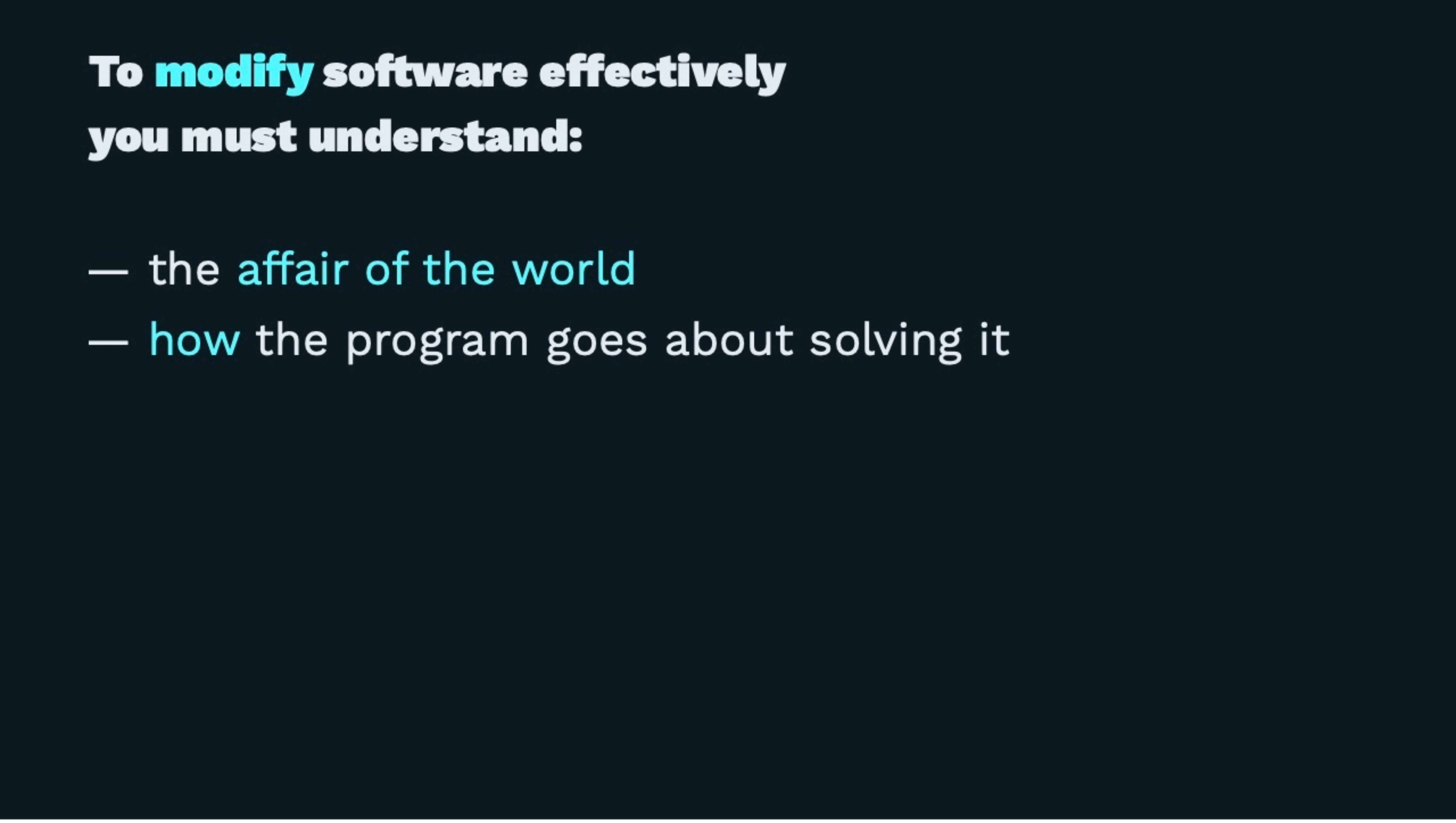
Naur was writing an an earlier era, so he talks about single programs here. Today, we write many programs and connect them all together into software systems. What he called "the theory of the program" is what I would call "the model of the system", but both phrases get at the heart of the concept.


Software is:

— a lot of text files with instructions to computers (they matter!)
— that express the authors' understanding of a real-world problem
— and their solution to that problem
— (and the same for every building block they needed along the way)

Programming is how we get there.

And this is what we have to understand to function effectively. Let's zero in on one part of that.


To modify software effectively you must understand:

— the affair of the world

— how the program goes about solving it

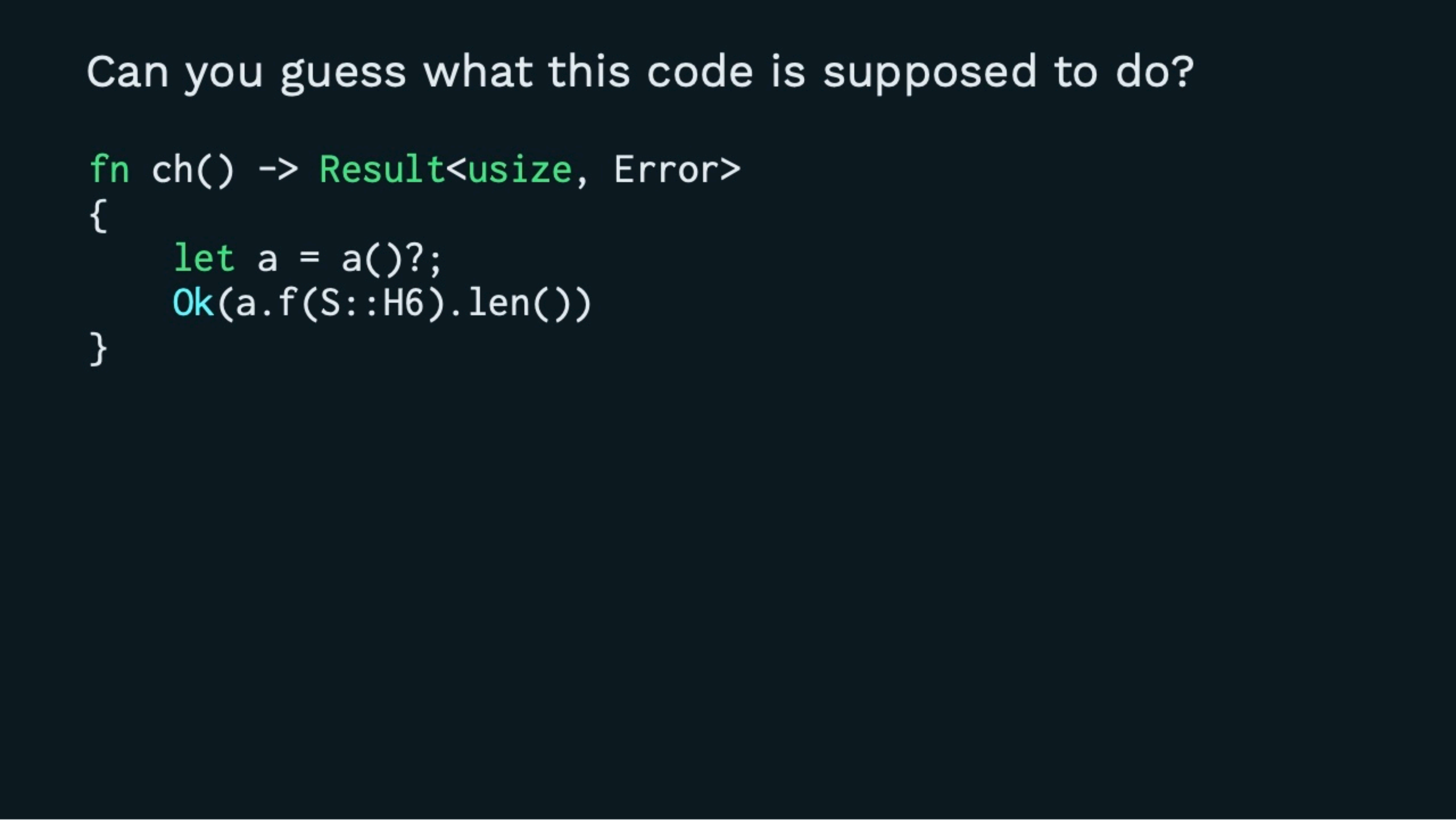
The how is mind-bogglingly complex, and very few people working on any team project understand the whole thing. Some people who've been involved with it for a long time might have a better understanding than others, but it's possible that nobody understands the whole thing.

^ Now, I want to back up from the theory a little bit to talk about those text files. They do matter!

Code is **communication** with computers and humans.

- Code defines data (nouns) and functions (verbs).
- We name things carefully because the names are meaningful to humans.
- A program becomes a language of its own. (Hat-tip to Dijkstra.)

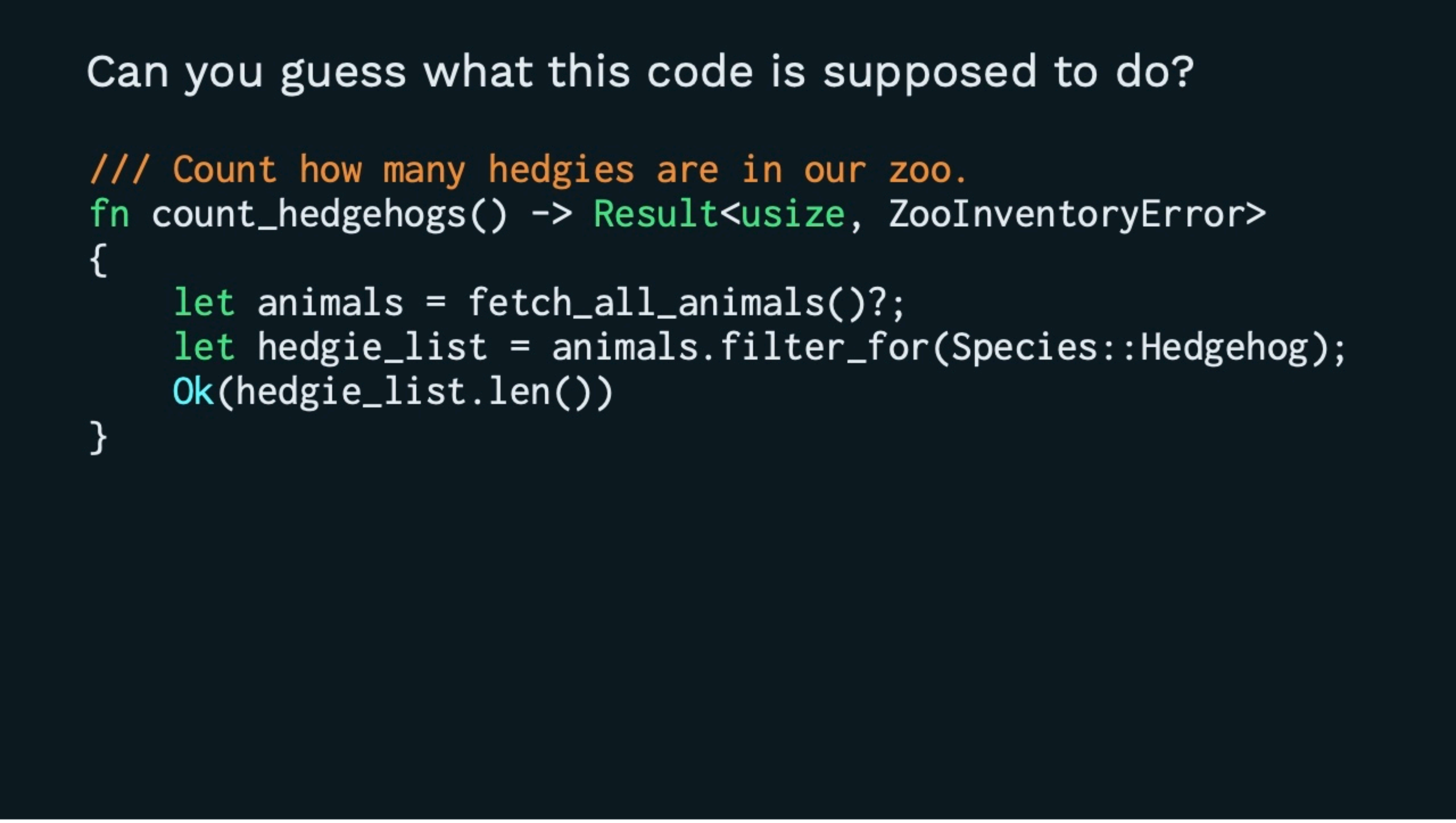
In the jargon of programmers, every complex system is a domain-specific language expressing our understanding of the problem.



Can you guess what this code is supposed to do?

```
fn ch() -> Result<usize, Error>
{
    let a = a()?;
    Ok(a.f(S::H6).len())
}
```

The programmers in the audience all guess that it's getting the length of something, but they have no idea what that's the length of, or what any of the other stuff does.



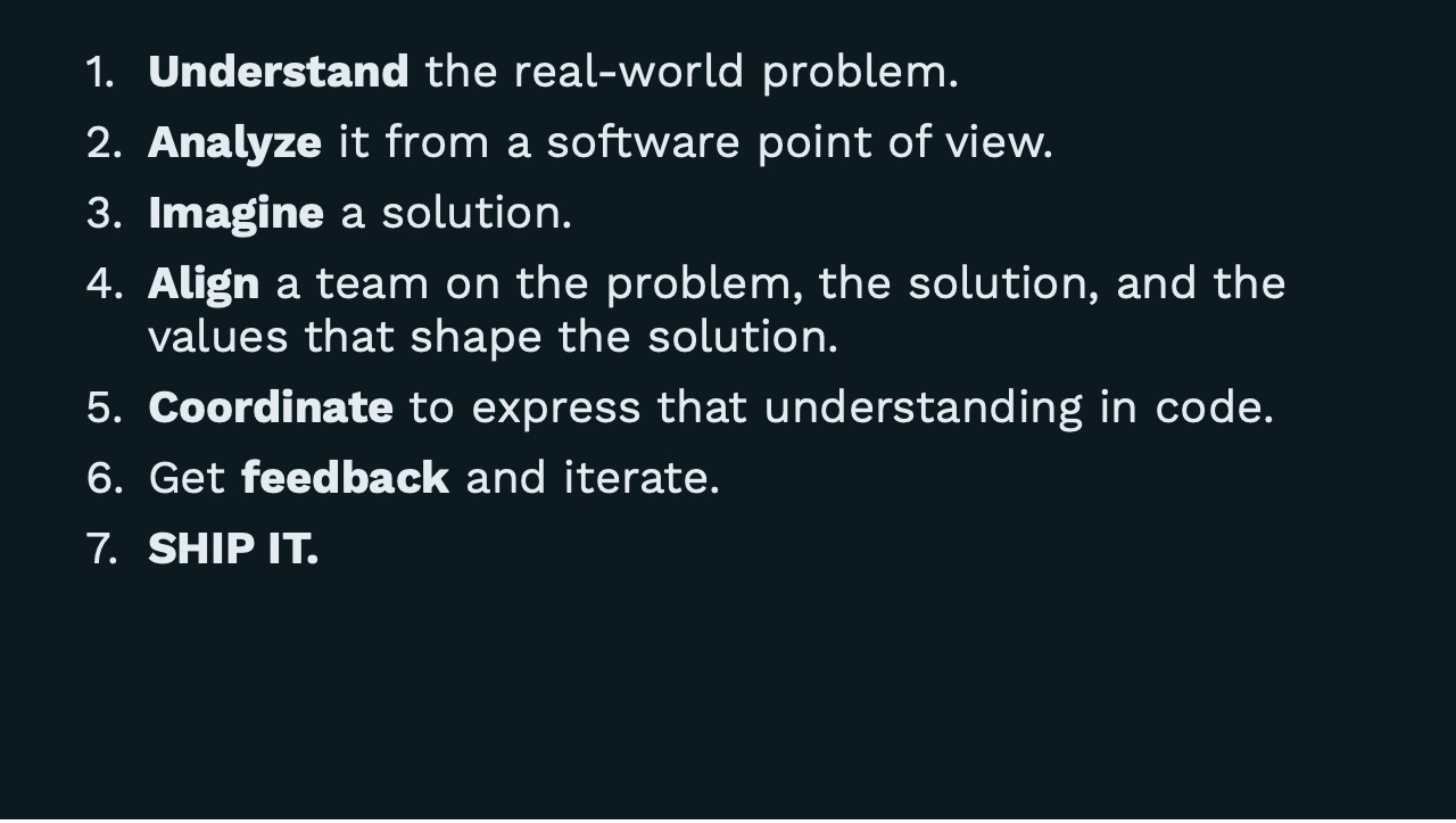
Can you guess what this code is supposed to do?

```
/// Count how many hedgies are in our zoo.
fn count_hedgehogs() -> Result<usize, ZooInventoryError>
{
    let animals = fetch_all_animals()?;
    let hedgie_list = animals.filter_for(Species::Hedgehog);
    Ok(hedgie_list.len())
}
```

You probably have a good guess about what this means, even if you don't know the specific programming language I'm using or any programming language at all. This code communicates to humans as well as computers. This might do the exact same thing as the previous code when run, but this version has an additional layer of useful meaning, and supports Naur's theory-building better. (It could lie, and be about counting numbats, but we try not to do that.)

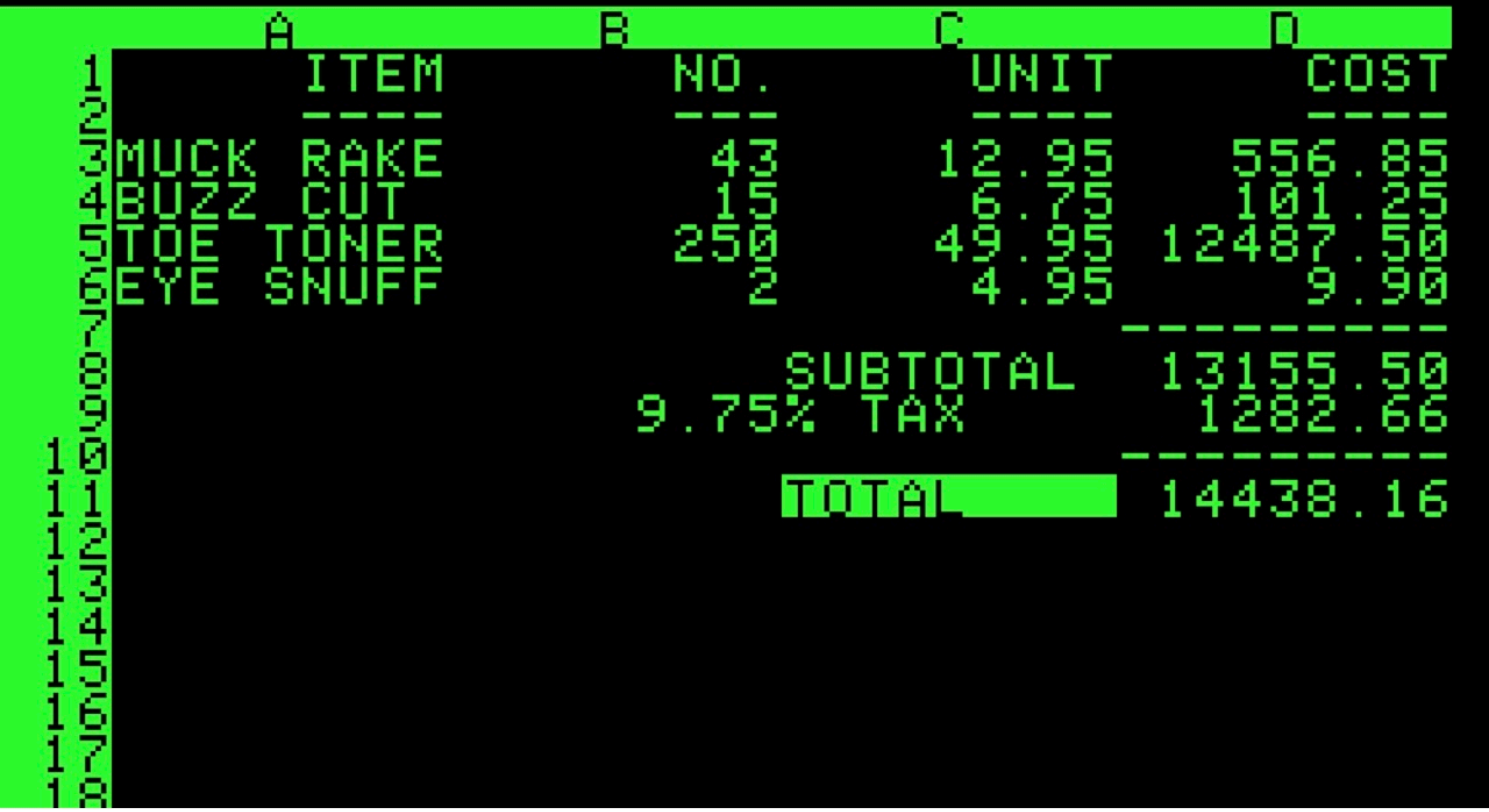
How do we invent that specific language to express a problem?

One thing that I have learned is that no two software solutions of a problem ever look alike. I know what little I know about sudoku solving from the talk Cory gave at a lunch and learn a couple of weeks ago. But if you gave me and Cory the task of writing a sudoku solver, we'd write COMPLETELY different programs. If you gave us the task of writing a solver together, we'd write something different again. This, btw, is very cool, because it says something about human minds that fascinates me. BUT despite the differences in end result, both of us would use a similar heuristic to get there.



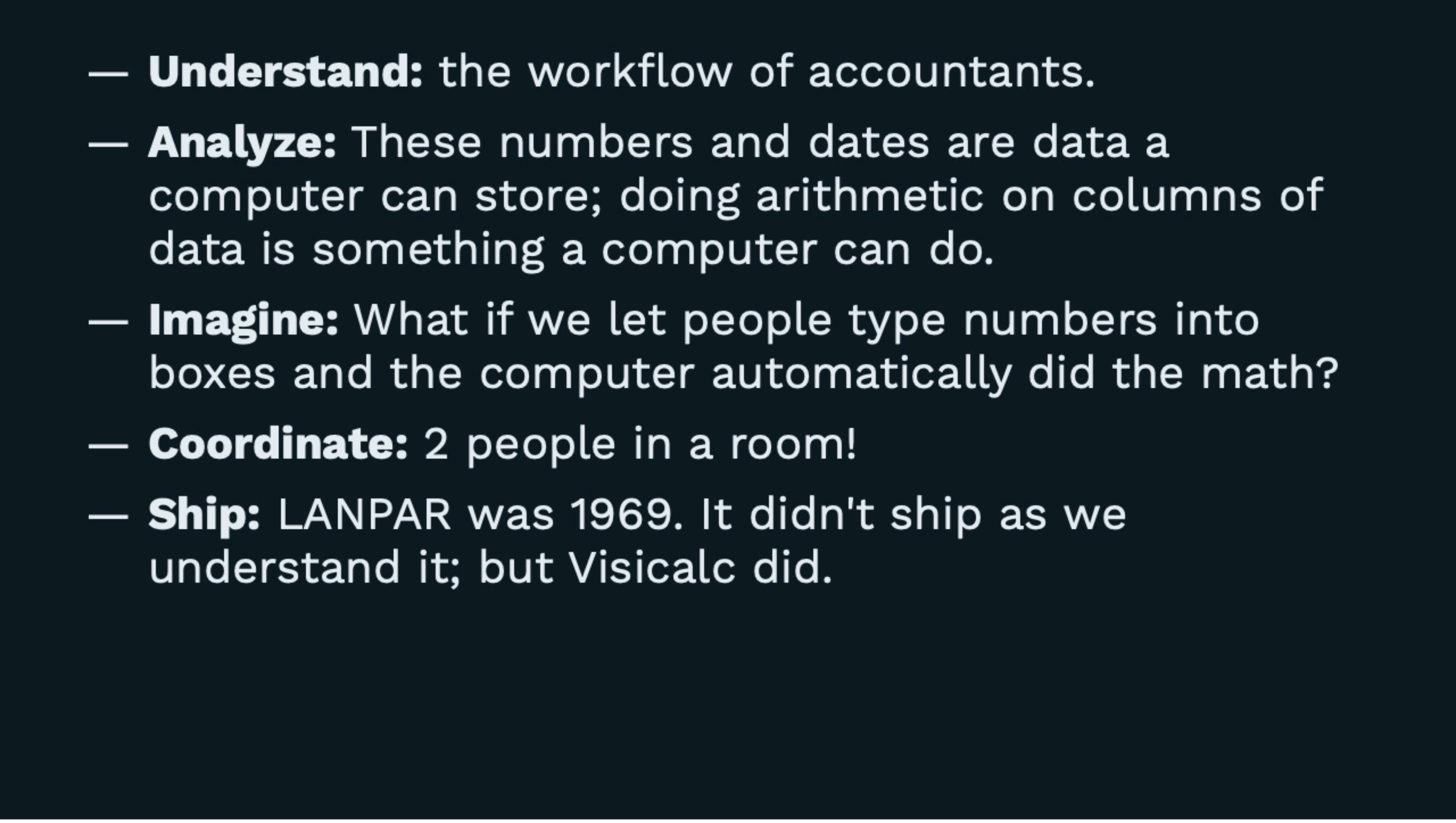
1. **Understand** the real-world problem.
2. **Analyze** it from a software point of view.
3. **Imagine** a solution.
4. **Align** a team on the problem, the solution, and the values that shape the solution.
5. **Coordinate** to express that understanding in code.
6. Get **feedback** and iterate.
7. **SHIP IT.**

There are no secrets here. It works this way for all problems in software, whether small or large. Some things are easier when you're a team of one-- it's easy to align with yourself. That might be hard with a team of 20, and very hard indeed when your team is is larger than Dunbar's number. But this is how it works. Let's look at a simple example.



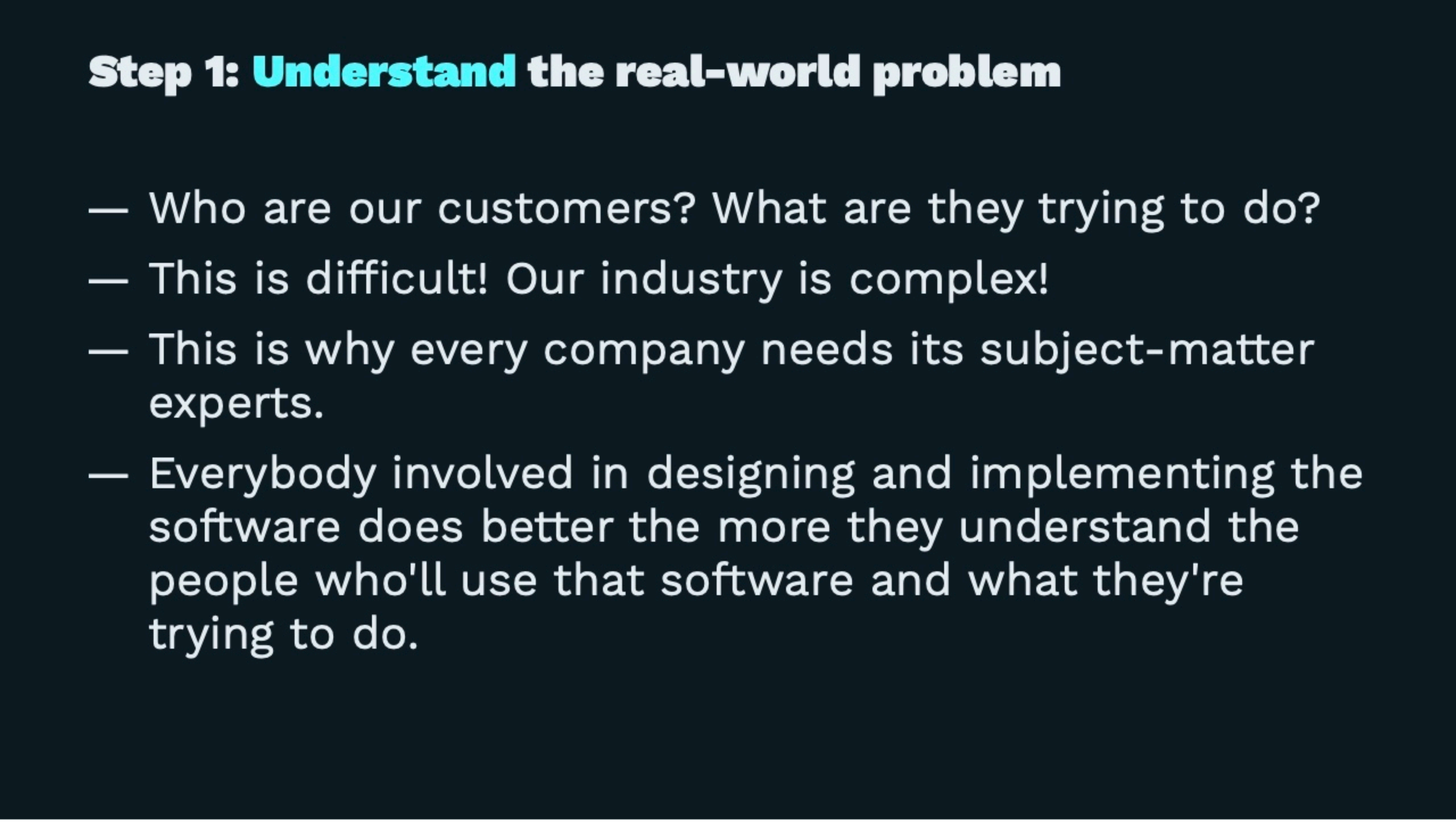
```
           A          B          C          D
 1       ITEM        NO.       UNIT       COST
 2       ----        ---       ----       ----
 3MUCK RAKE           43      12.95     556.85
 4BUZZ CUT            15       6.75     101.25
 5TOE TONER          250      49.95   12487.50
 6EYE SNUFF            2       4.95       9.90
 7                                  ----------
 8                    SUBTOTAL        13155.50
 9               9.75% TAX             1282.66
10                                  ----------
11                    TOTAL           14438.16
12
13
14
15
16
17
18
```

This is Visicalc, the first spreadsheet software anybody remembers. 1977. (The first one was LANPAR in 1969.) This one invention sold personal computers to millions of small businesses and is a huge part of Microsoft's revenue even today. Spreadsheets ate the world and run many businesses and are part of critical workflows everywhere. But somebody had to make the first one.



- **Understand:** the workflow of accountants.
- **Analyze:** These numbers and dates are data a computer can store; doing arithmetic on columns of data is something a computer can do.
- **Imagine:** What if we let people type numbers into boxes and the computer automatically did the math?
- **Coordinate:** 2 people in a room!
- **Ship:** LANPAR was 1969. It didn't ship as we understand it; but Visicalc did.

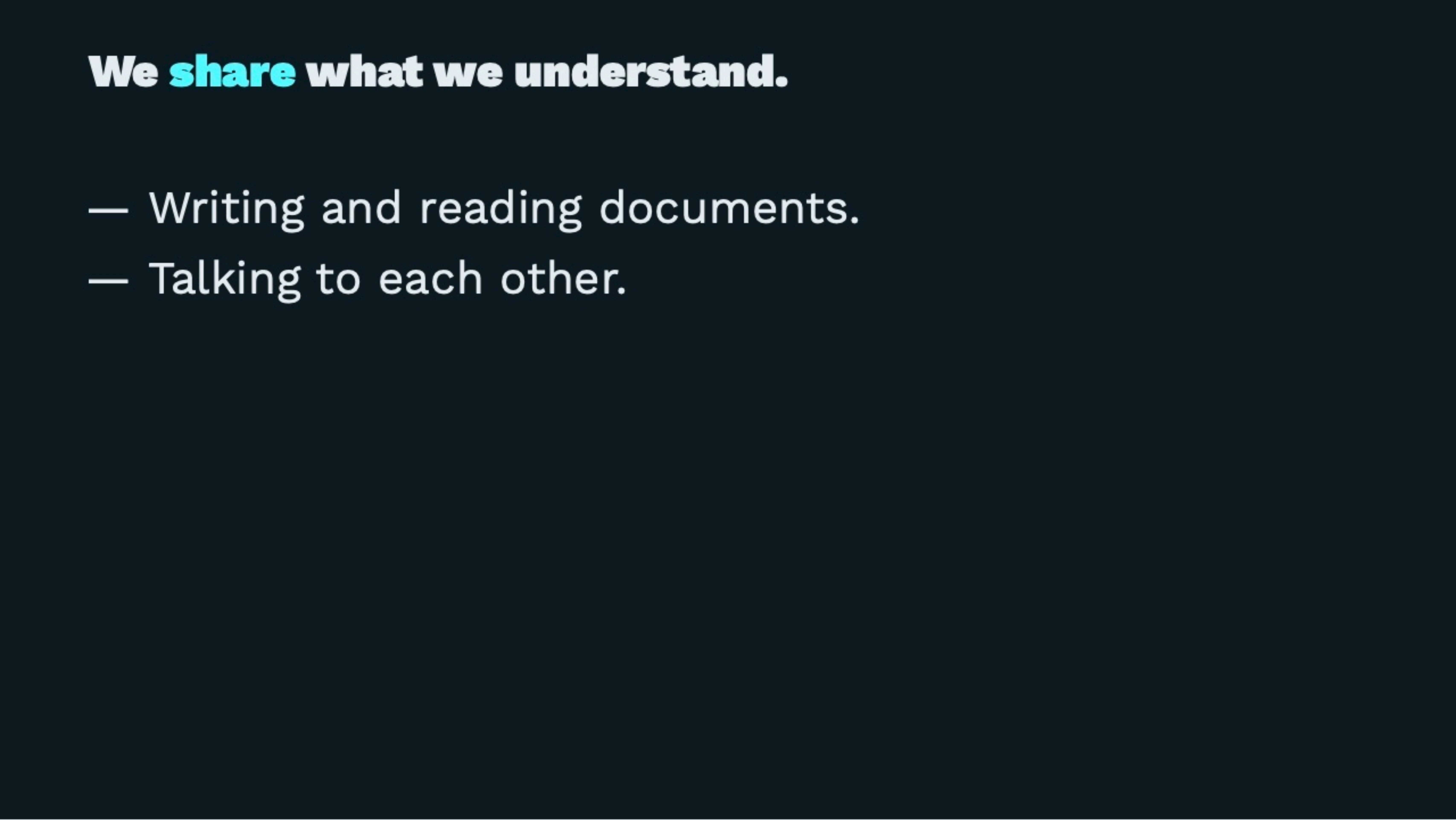
The word "spreadsheet" comes directly from accounting. Let's go broader, and apply the process to our shared endeavor.



Step 1: Understand the real-world problem

- Who are our customers? What are they trying to do?
- This is difficult! Our industry is complex!
- This is why every company needs its subject-matter experts.
- Everybody involved in designing and implementing the software does better the more they understand the people who'll use that software and what they're trying to do.

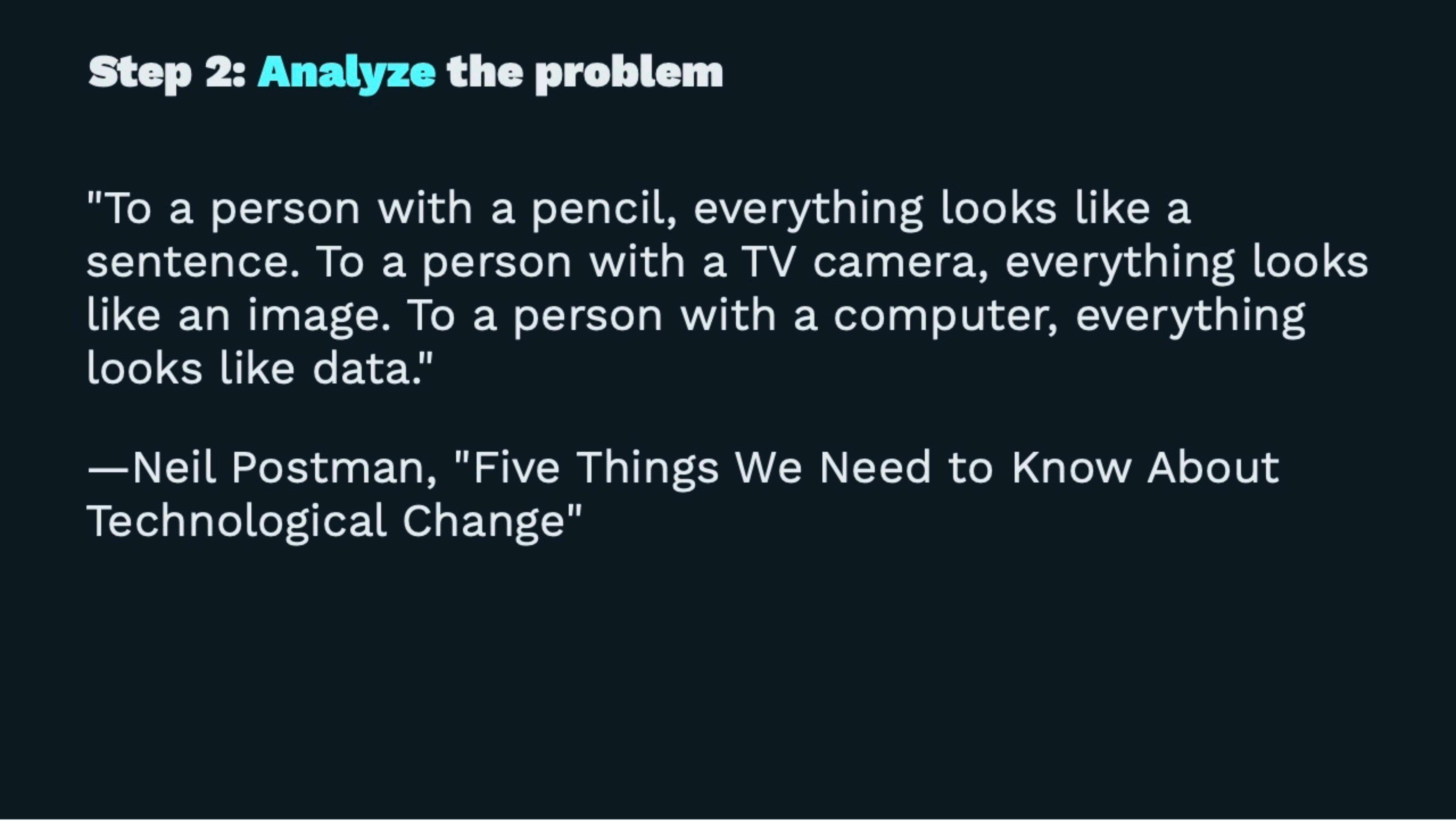
Our experts and our customer contact people keep programmers like me in touch with who we're making tools for. I believe I speak for every person on the engineering team when I say that we all desperately want more understanding of our customers. Please! Talk to us!



We share what we understand.

— Writing and reading documents.
— Talking to each other.

Once we understand something, we don't leap to writing code. Instead we share that understanding.



Step 2: Analyze the problem

"To a person with a pencil, everything looks like a sentence. To a person with a TV camera, everything looks like an image. To a person with a computer, everything looks like data."

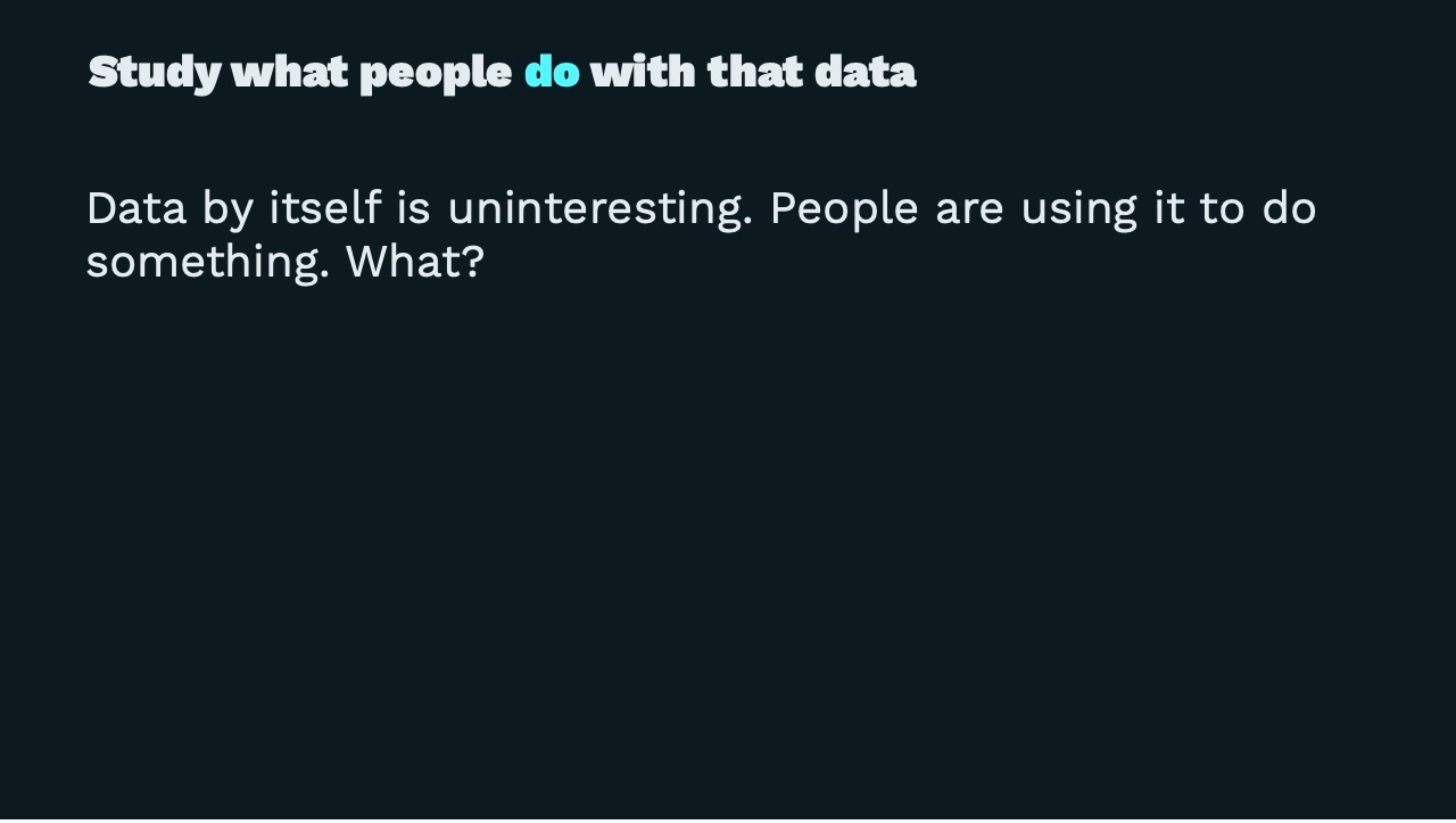
—Neil Postman, "Five Things We Need to Know About Technological Change"

Or more succinctly, the medium is the message, and the medium of software is data.

Study the data

The medium of software is information, or data. Software collects or generates data, then transforms that data via rules. The process of describing the data and writing the rules is what occupies us all day.

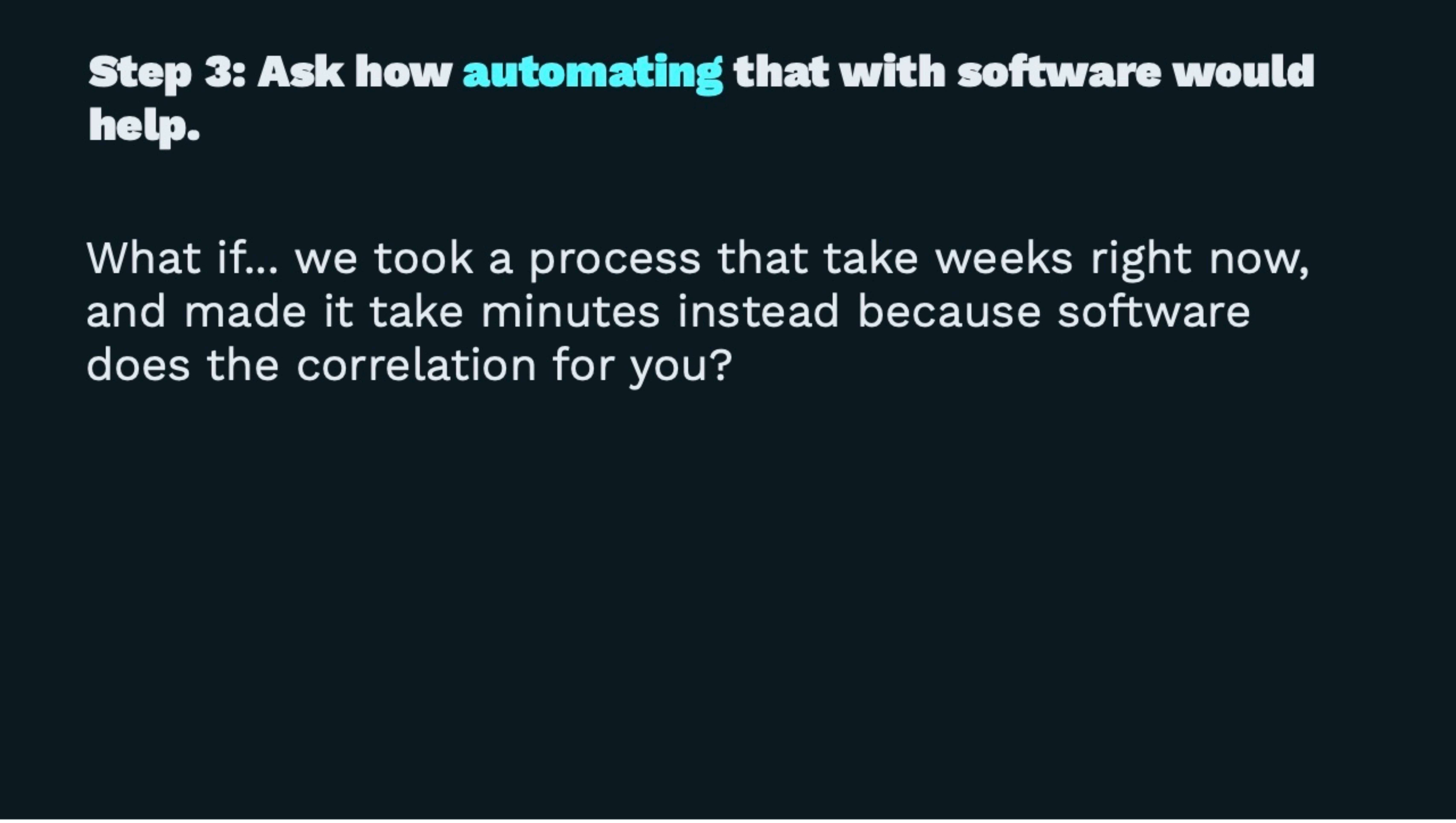
Call out some of the nouns we track in data.



Study what people do with that data

Data by itself is uninteresting. People are using it to do something. What?

Talk about how our customers use their data.



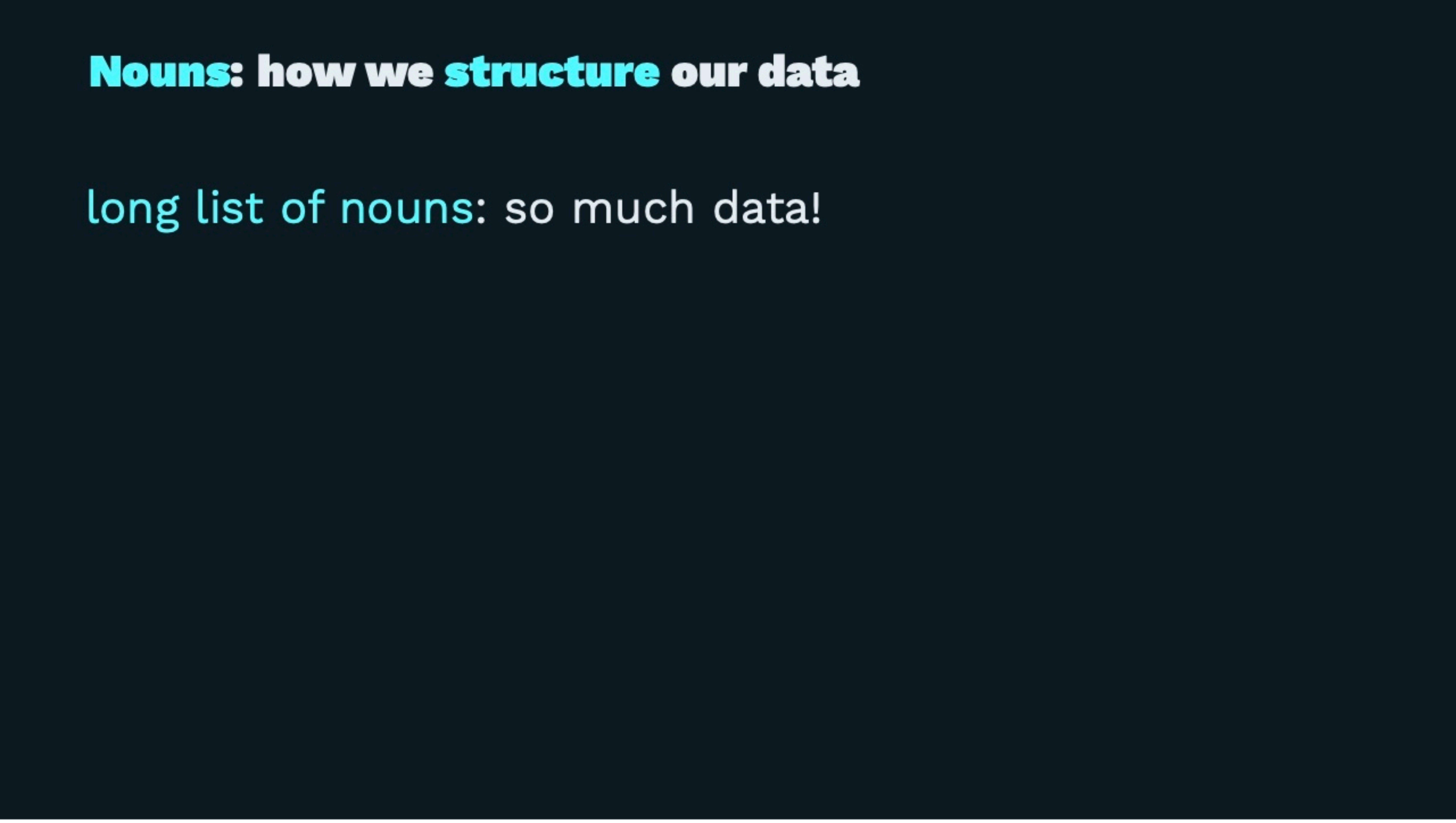
Step 3: Ask how automating that with software would help.

What if... we took a process that take weeks right now, and made it take minutes instead because software does the correlation for you?

Marc Andreesen described this as "software eating the world", and he should know. He invented the image tag, and that was enough.

Deepen that computer-focused analysis

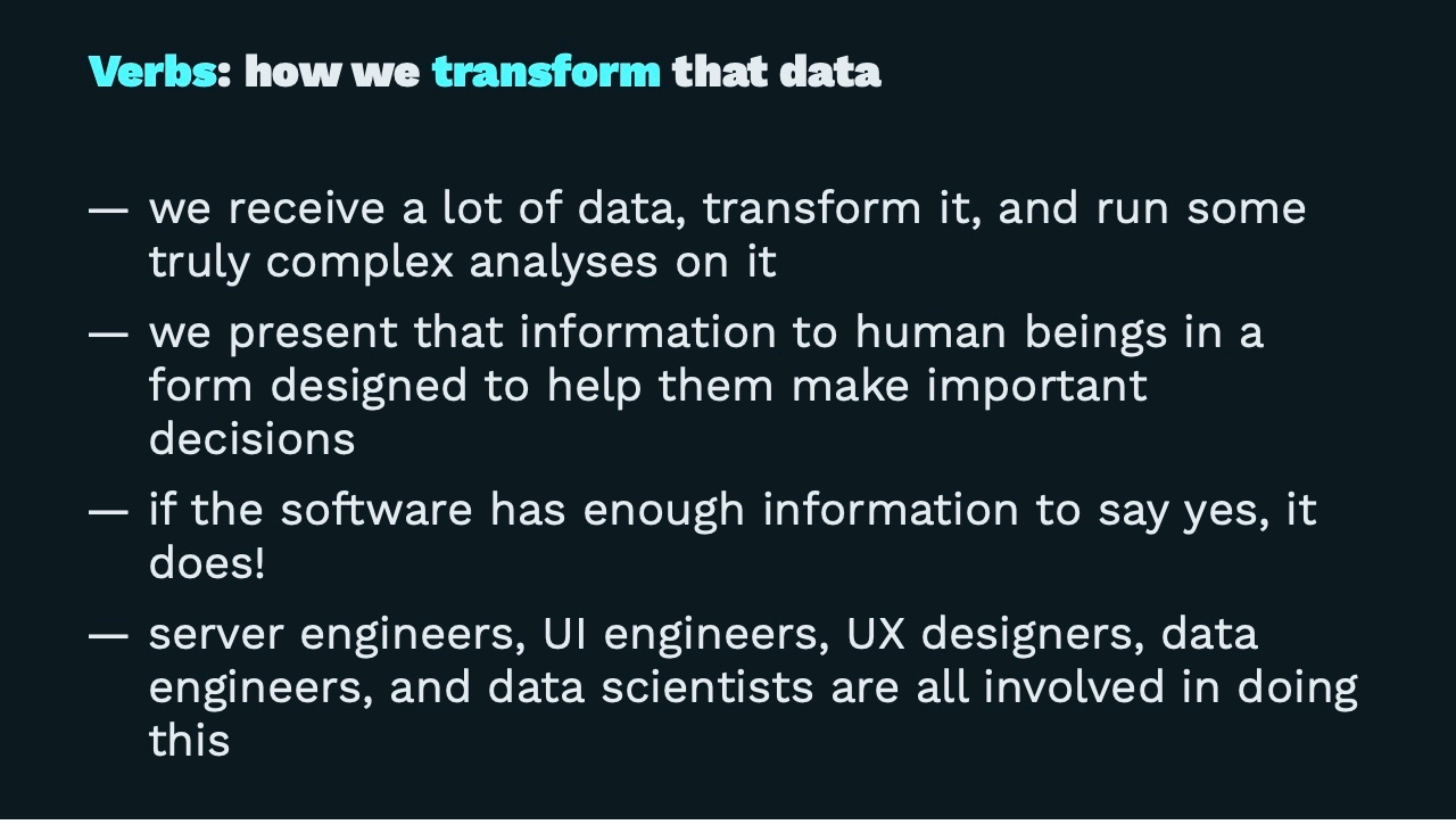
What data would the software need to have available? How will we get that data in a form we can use? What would we need to do with that data to present useful information to humans?



Nouns: how we structure our data

long list of nouns: so much data!

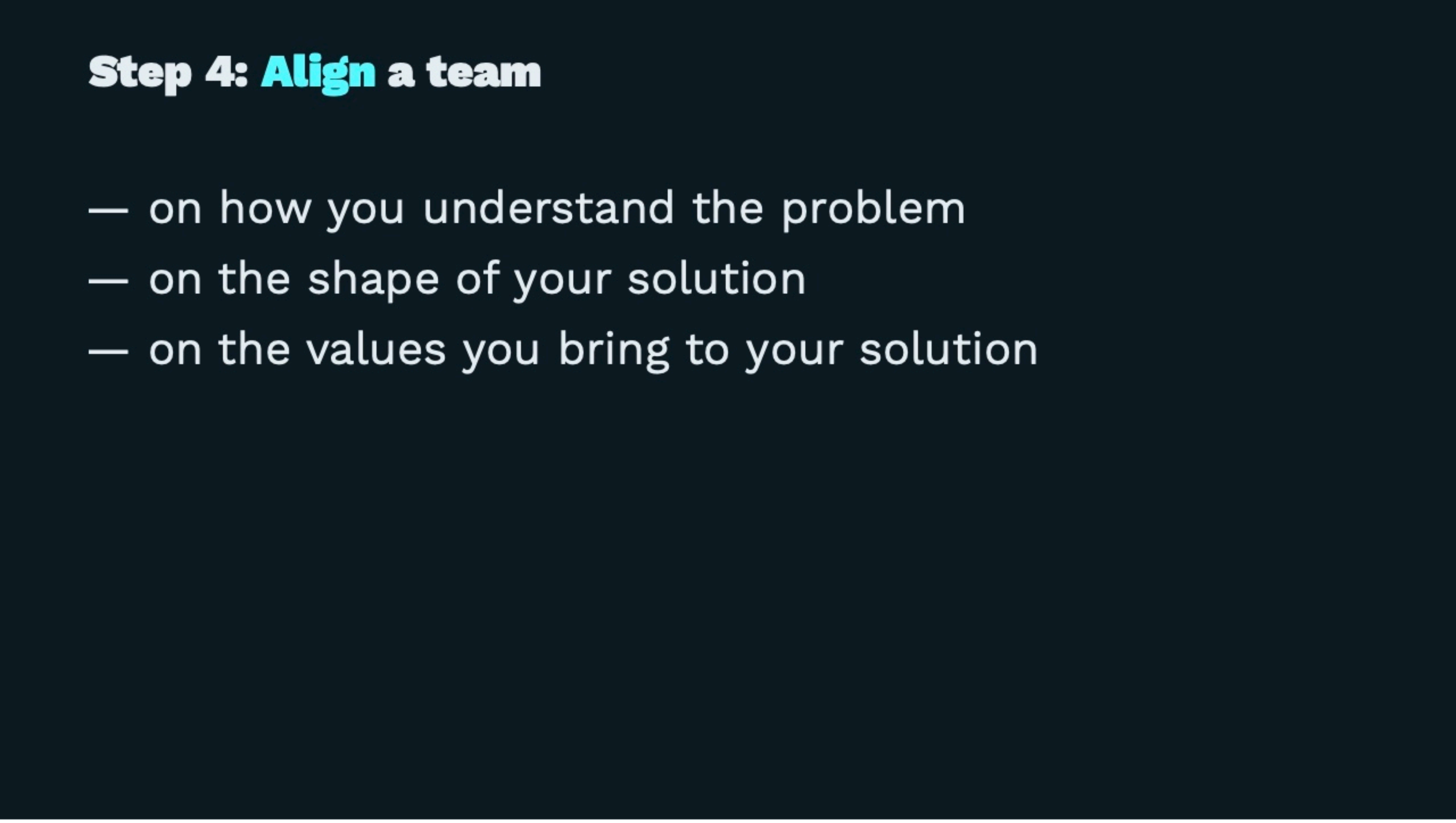
Talk about how subject-matter experts help us identify the data.



Verbs: how we transform that data

- we receive a lot of data, transform it, and run some truly complex analyses on it
- we present that information to human beings in a form designed to help them make important decisions
- if the software has enough information to say yes, it does!
- server engineers, UI engineers, UX designers, data engineers, and data scientists are all involved in doing this

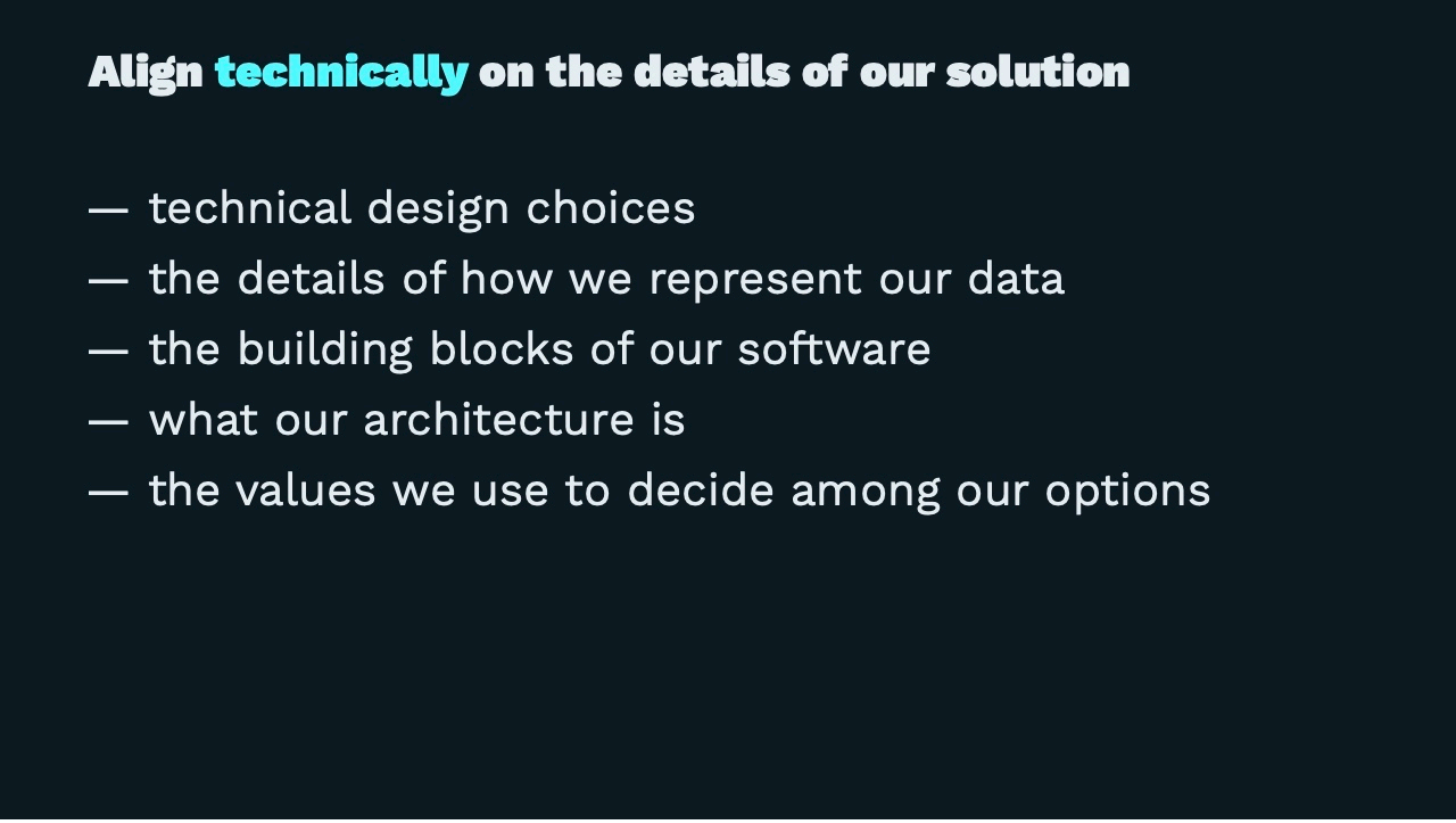
This is most of the work, right here. This is what the software *does*, its verbs.



Step 4: Align a team

— on how you understand the problem
— on the shape of your solution
— on the values you bring to your solution

This is what our company meeting does. Every week, we talk about what our customers are trying to do and how well we're solving their problems.


Align **technically** on the details of our solution

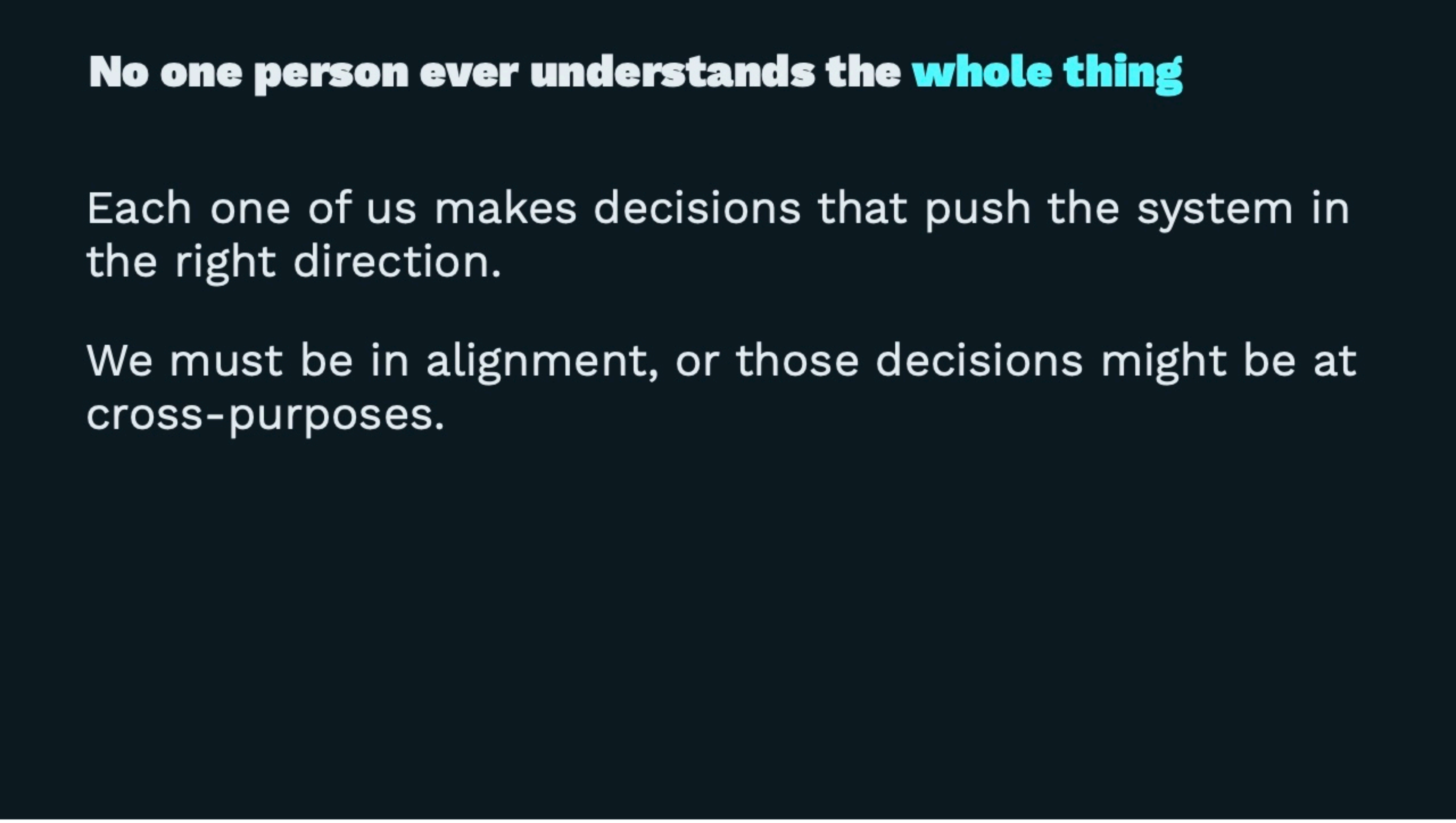
— technical design choices
— the details of how we represent our data
— the building blocks of our software
— what our architecture is
— the values we use to decide among our options

What programming languages are we using? How are we storing our data? Of the countless ways we might write this, which way are we picking?

Technical alignment comes from:

— Writing and reading documents.
— Talking to each other.
— Over and over (you don't stop).

Alignment is an ongoing task. We must constantly communicate in person and via design documents to make sure we all understand the direction we're going.



No one person ever understands the whole thing

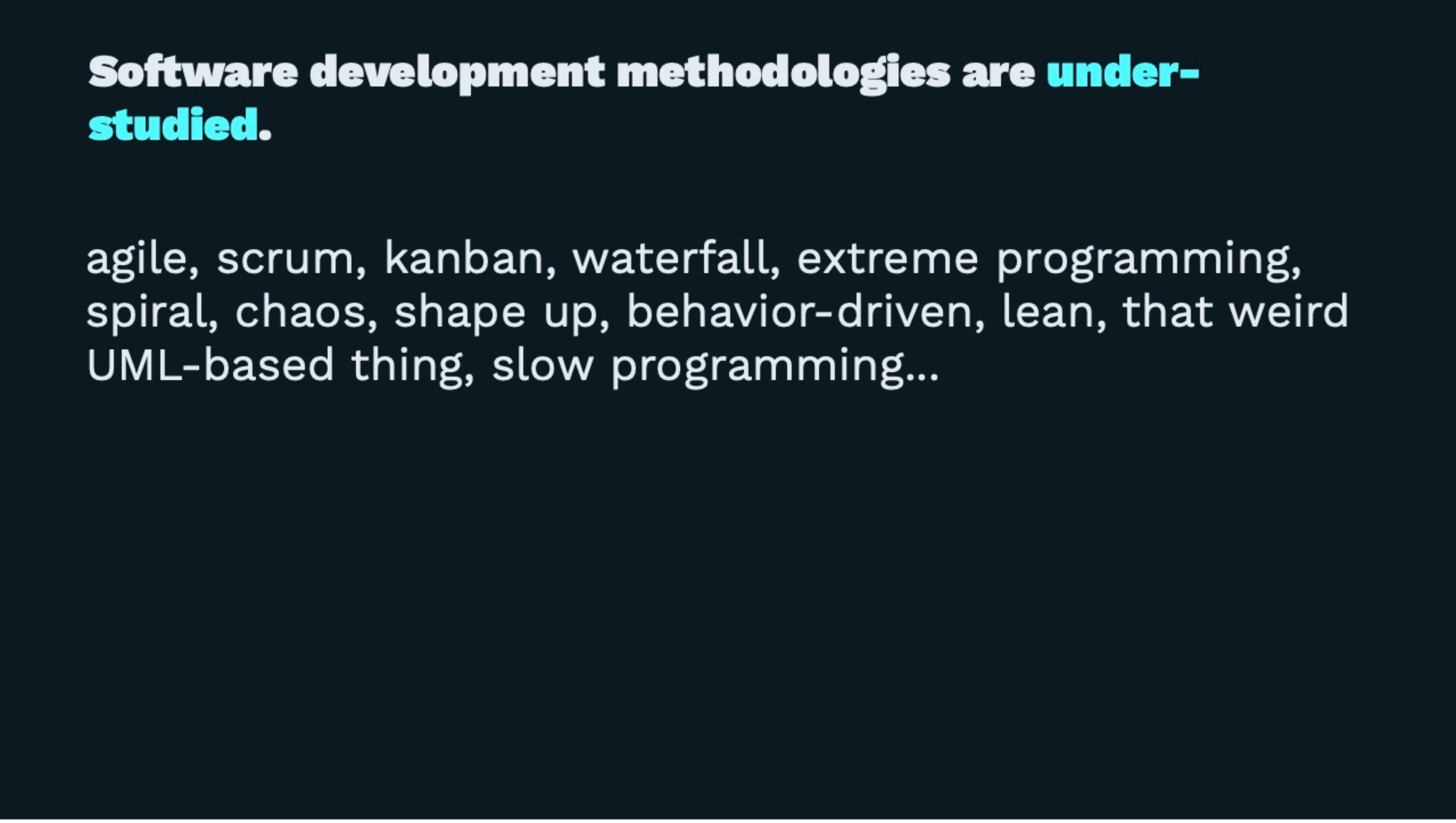
Each one of us makes decisions that push the system in the right direction.

We must be in alignment, or those decisions might be at cross-purposes.

Alignment is critical, because complex software is too big for any one person.

Step 5: Coordinate to write all those text files.

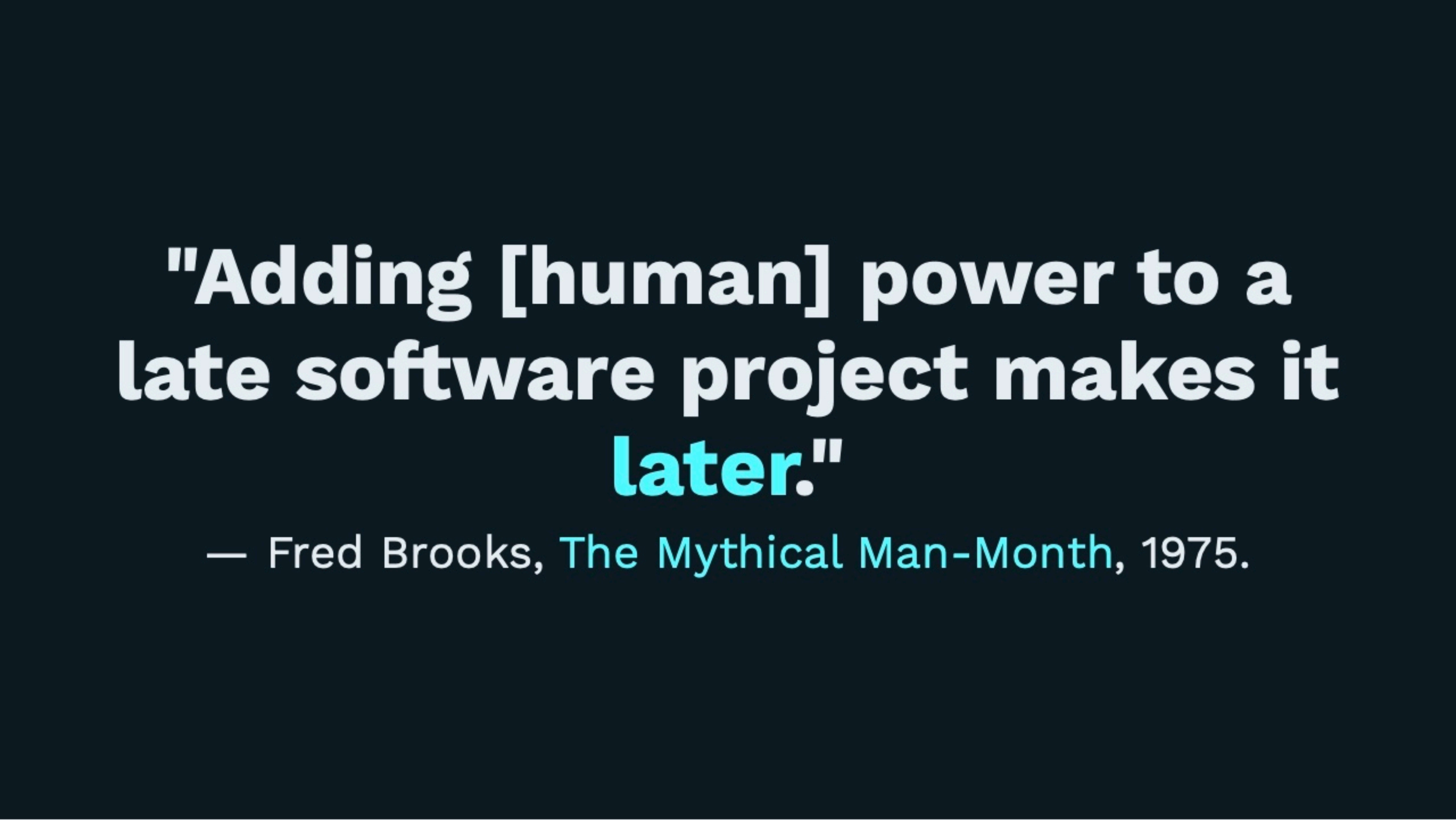
DEEP SIGH. This is where all the trouble is. I could give an entire presentation on what we know about this part of it, from books people have written about their face-plants through the years. Today I'll stick to sharing a couple of insights I hope will be useful.



Software development methodologies are under-studied.

agile, scrum, kanban, waterfall, extreme programming, spiral, chaos, shape up, behavior-driven, lean, that weird UML-based thing, slow programming...

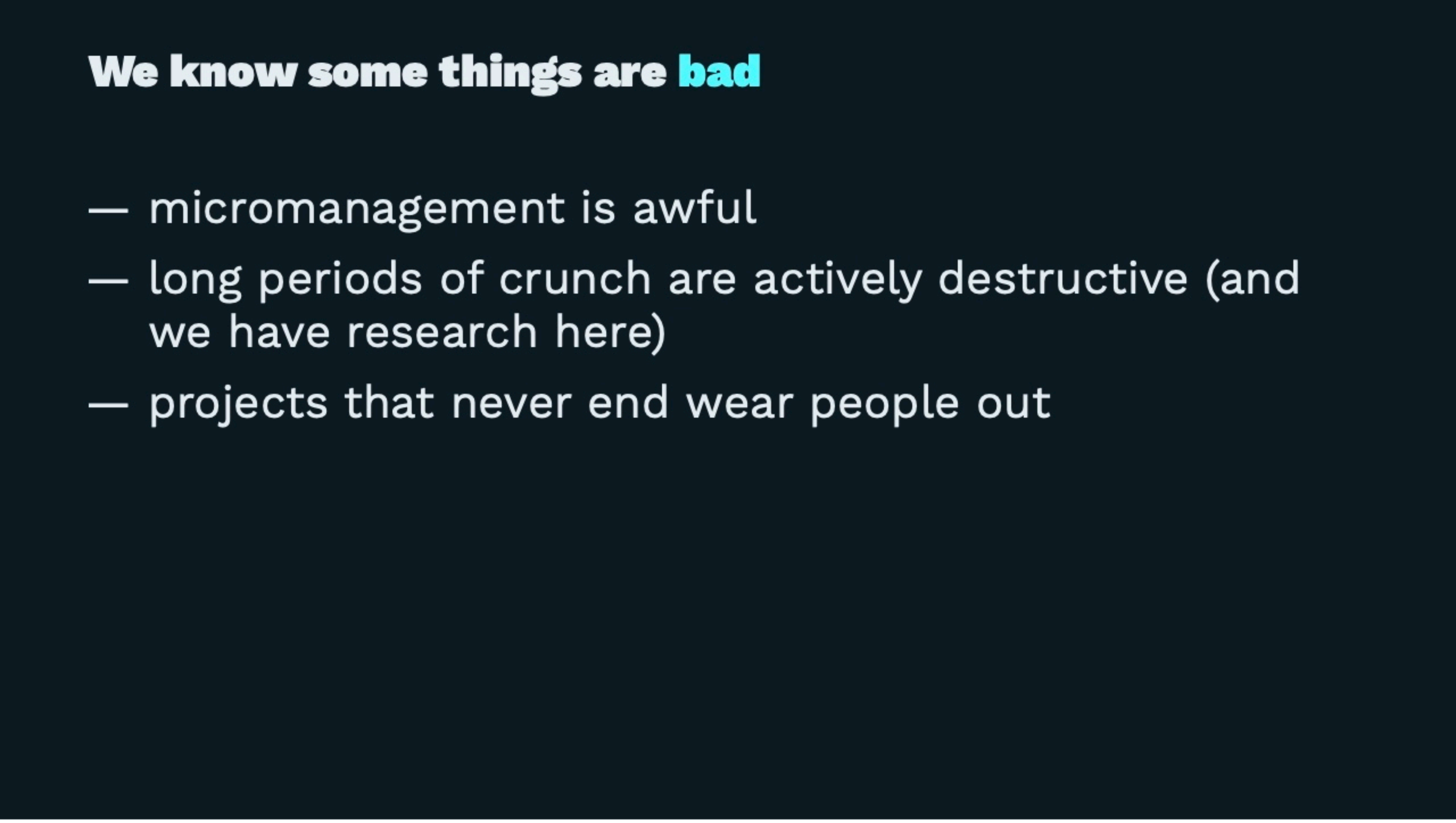
Which ones result in measurable, repeatable productivity improvements? No idea. Nobody has studied this. There are a few things we do know, from looking at past projects. We do know it's a team sport, and that communication is the core.



"Adding [human] power to a late software project makes it later."

— Fred Brooks, The Mythical Man-Month, 1975.

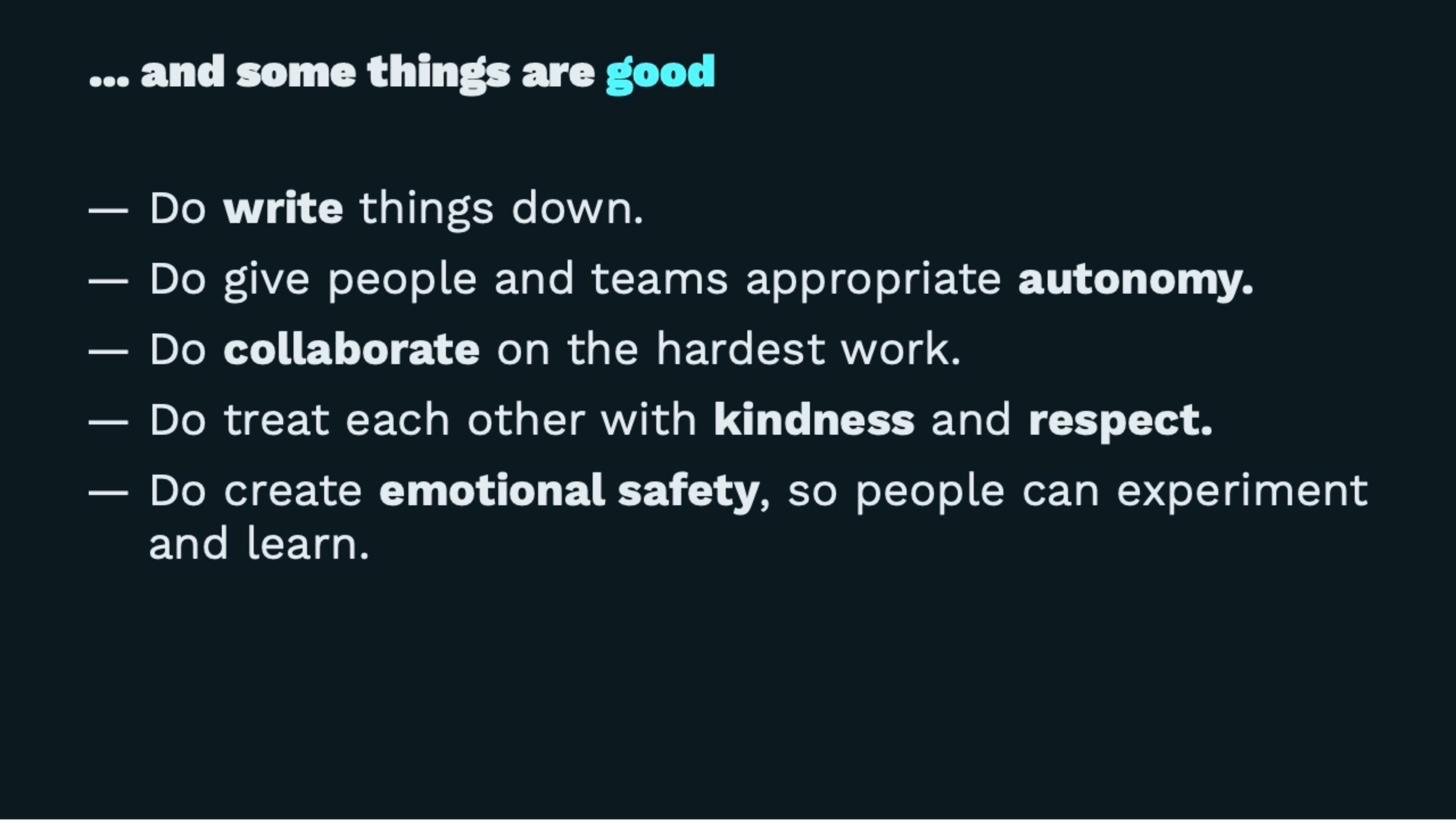
Why? Because communication is, as we nerds like to say, an order N squared problem. Adding the 10th person to a project team adds 9 new lines of communication to worry about to *everybody*. This is a great book with a lot of great project insight, including the nugget that if it takes one woman nine months to deliver a baby, it does not follow that it would take 9 women one month to do it. And yet this is something the software industry keeps trying to do...



We know some things are **bad**

- micromanagement is awful
- long periods of crunch are actively destructive (and we have research here)
- projects that never end wear people out

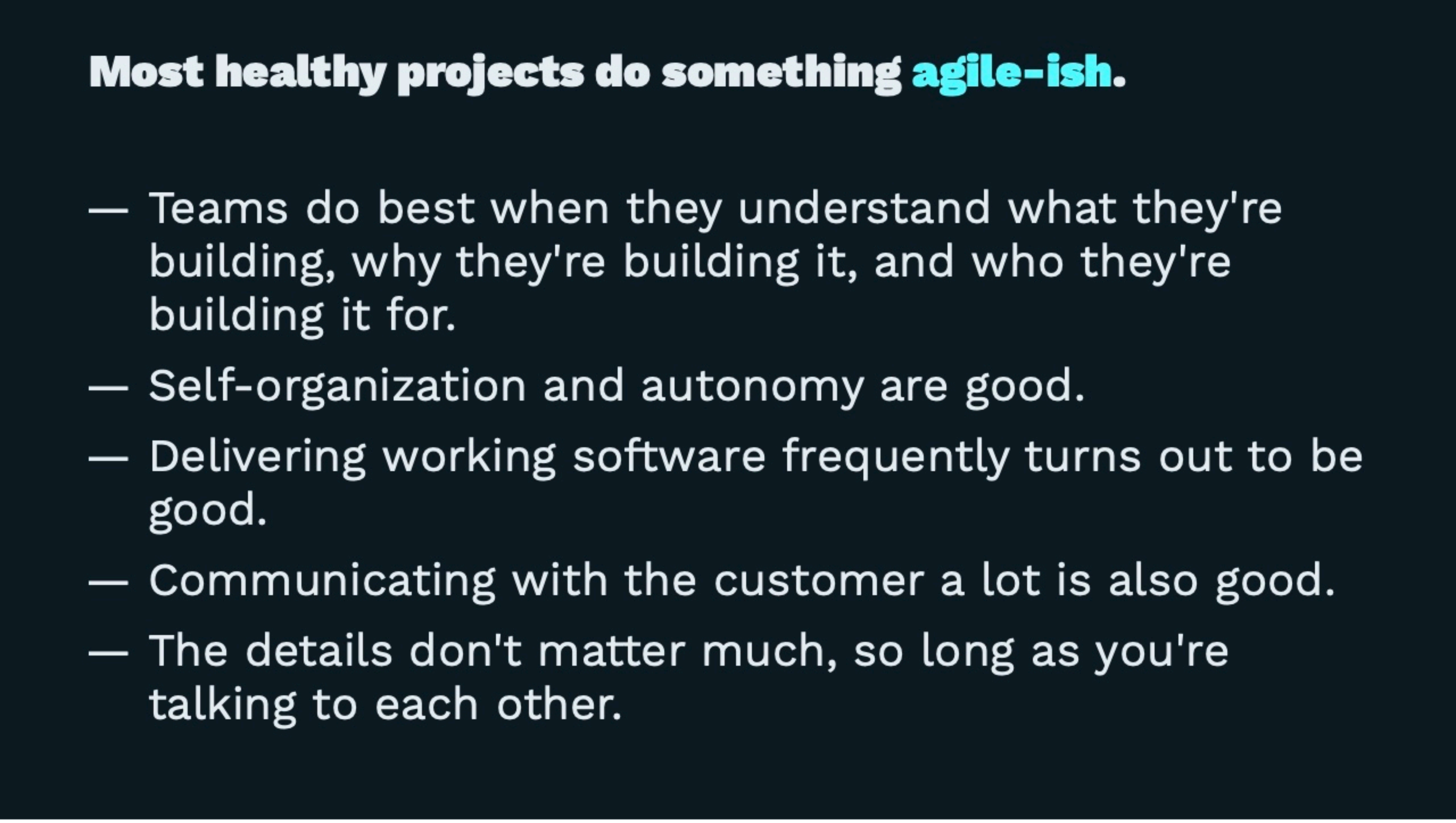
These things fall into the category of yeah, people are people.



... and some things are good

- Do **write** things down.
- Do give people and teams appropriate **autonomy.**
- Do **collaborate** on the hardest work.
- Do treat each other with **kindness** and **respect.**
- Do create **emotional safety**, so people can experiment and learn.

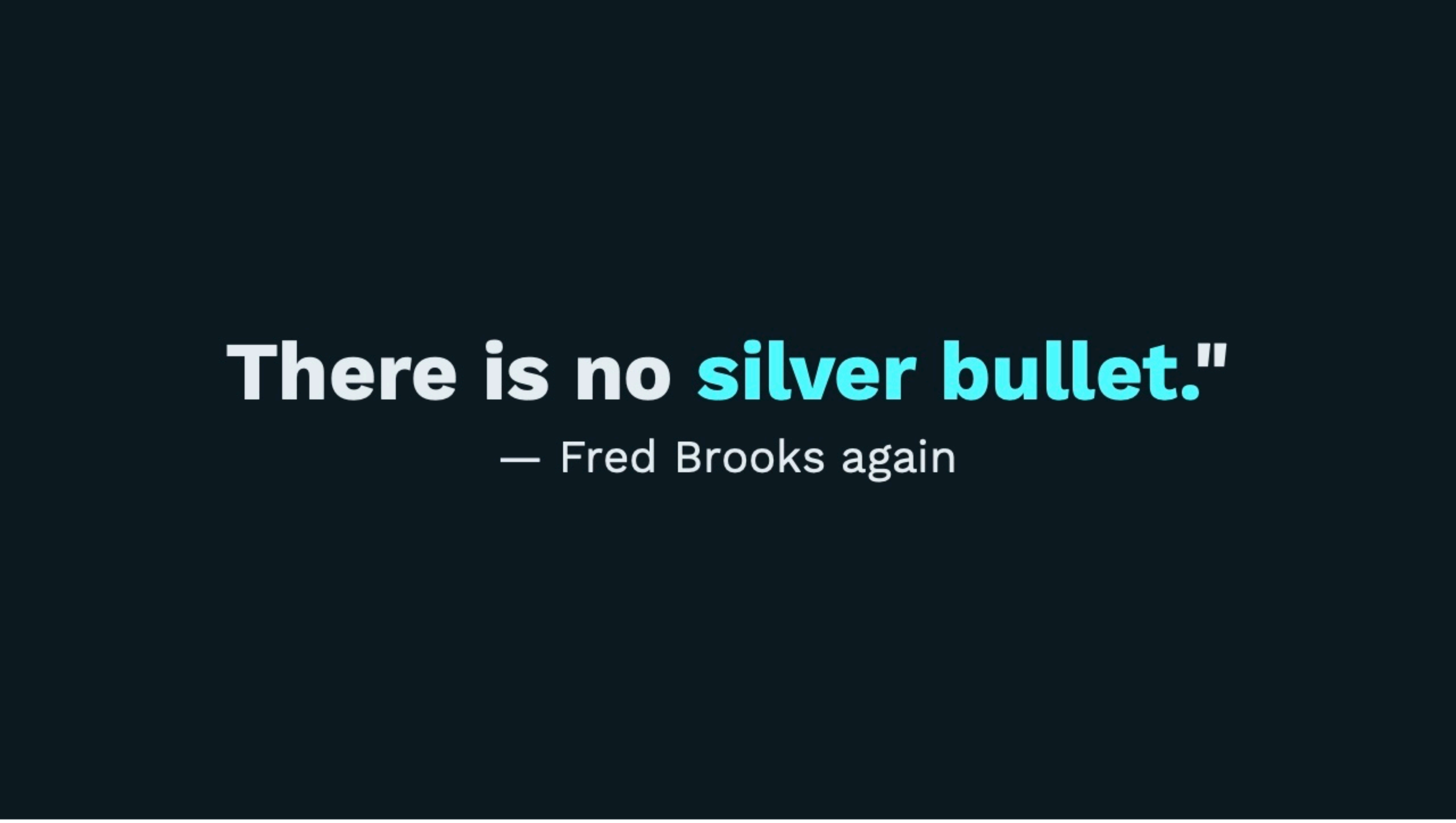
Huh, none of those things are about process meetings. All of these things are about enabling smart people to do their best work. Strange. Okay, let's talk process for two more slides.



Most healthy projects do something agile-ish.

- Teams do best when they understand what they're building, why they're building it, and who they're building it for.
- Self-organization and autonomy are good.
- Delivering working software frequently turns out to be good.
- Communicating with the customer a lot is also good.
- The details don't matter much, so long as you're talking to each other.

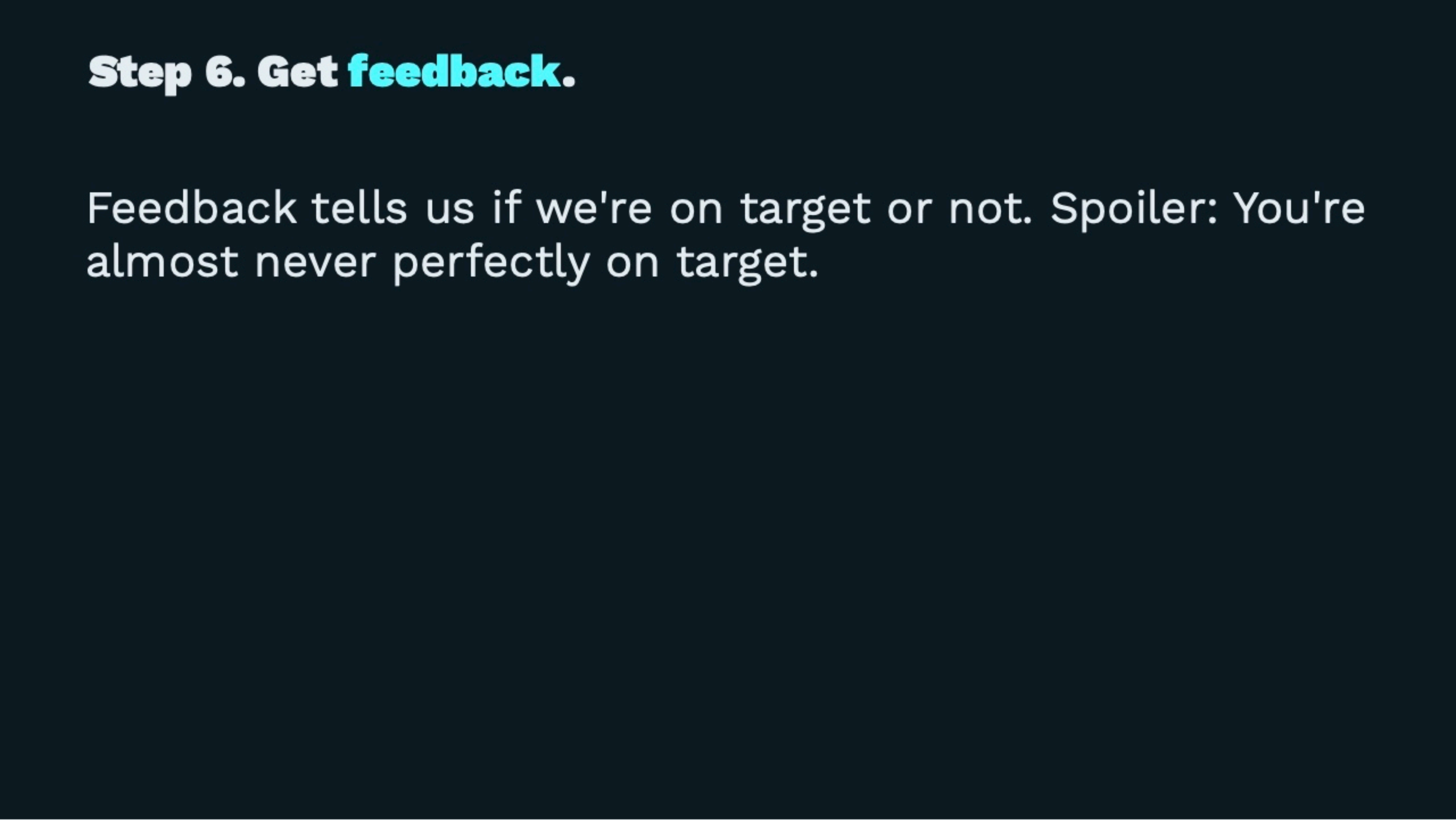
The Agile Manifesto is actually good.



There is no **silver bullet.**"

— Fred Brooks again

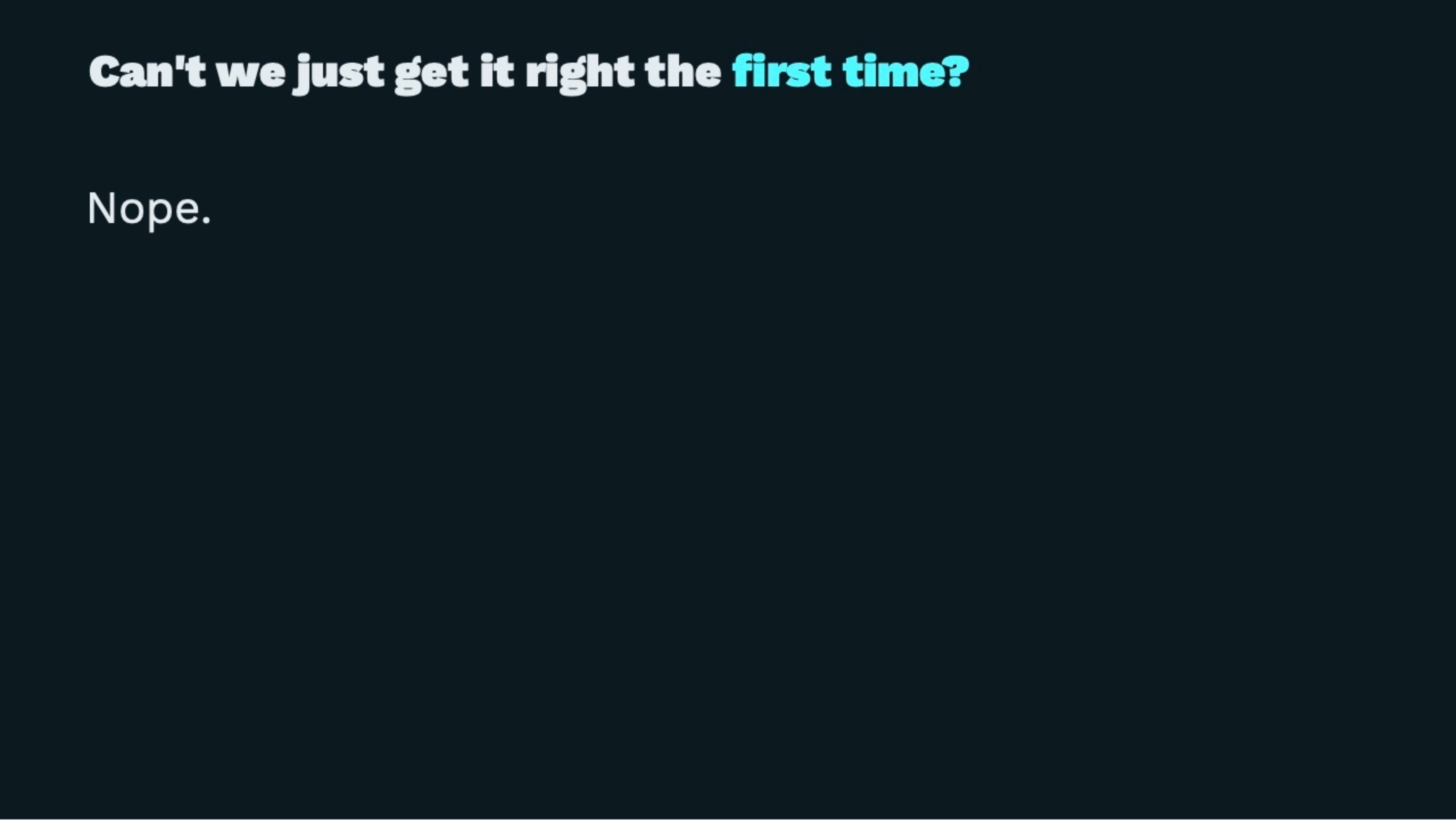
There is no single solution that works for every team in every moment.



Step 6. Get feedback.

Feedback tells us if we're on target or not. Spoiler: You're almost never perfectly on target.

Feedback loops are pretty important. We need to check on how we're doing. We run retrospectives on incidents and on projects to see how we're doing with our processes, and learn from our experiences. Do more of this? Less of that? Feedback loops are how learning happens.



Can't we just get it right the first time?

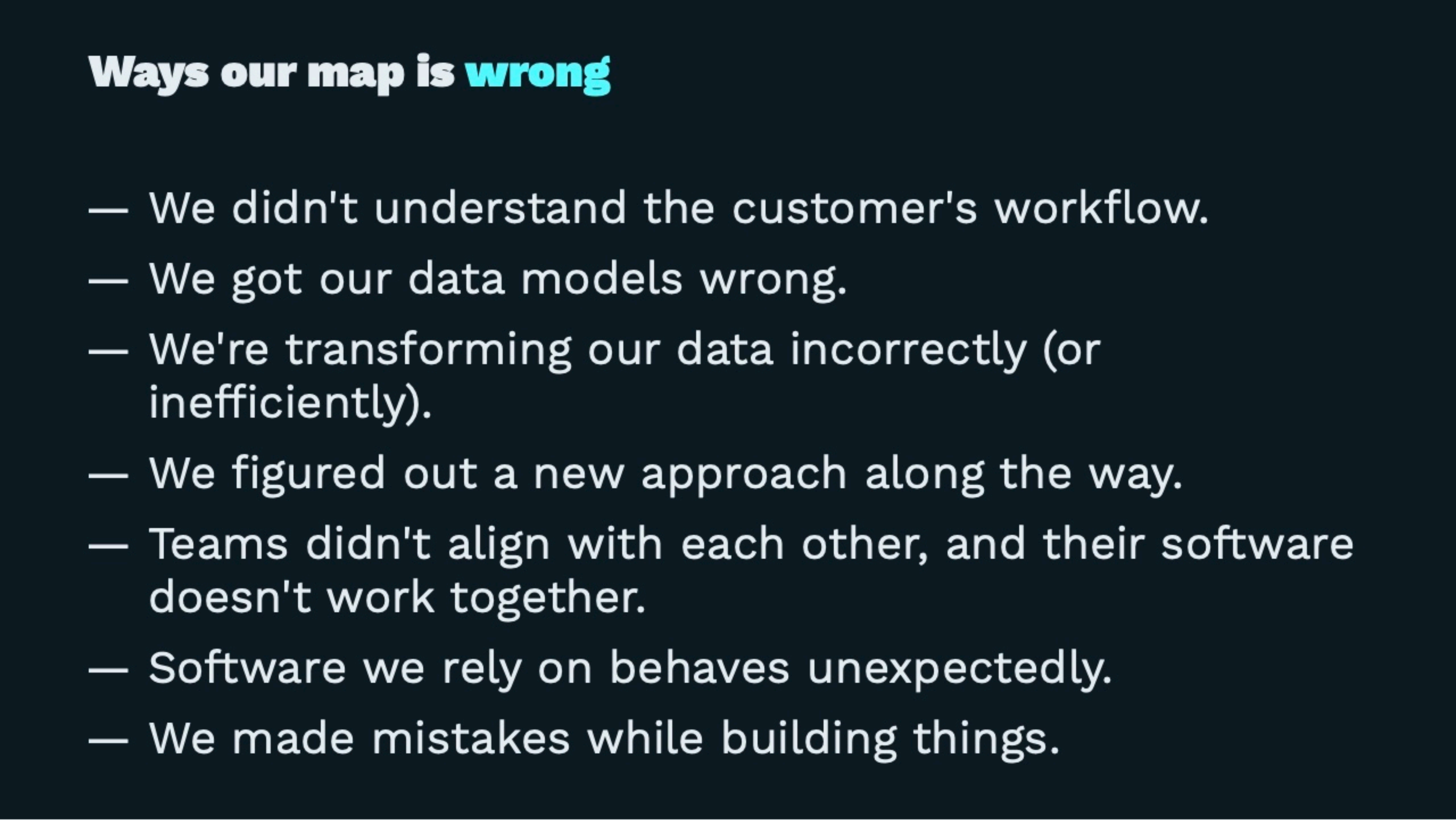
Nope.

And there's a reason why we can't.

"The map is not the territory."

— Alfred Korzybski

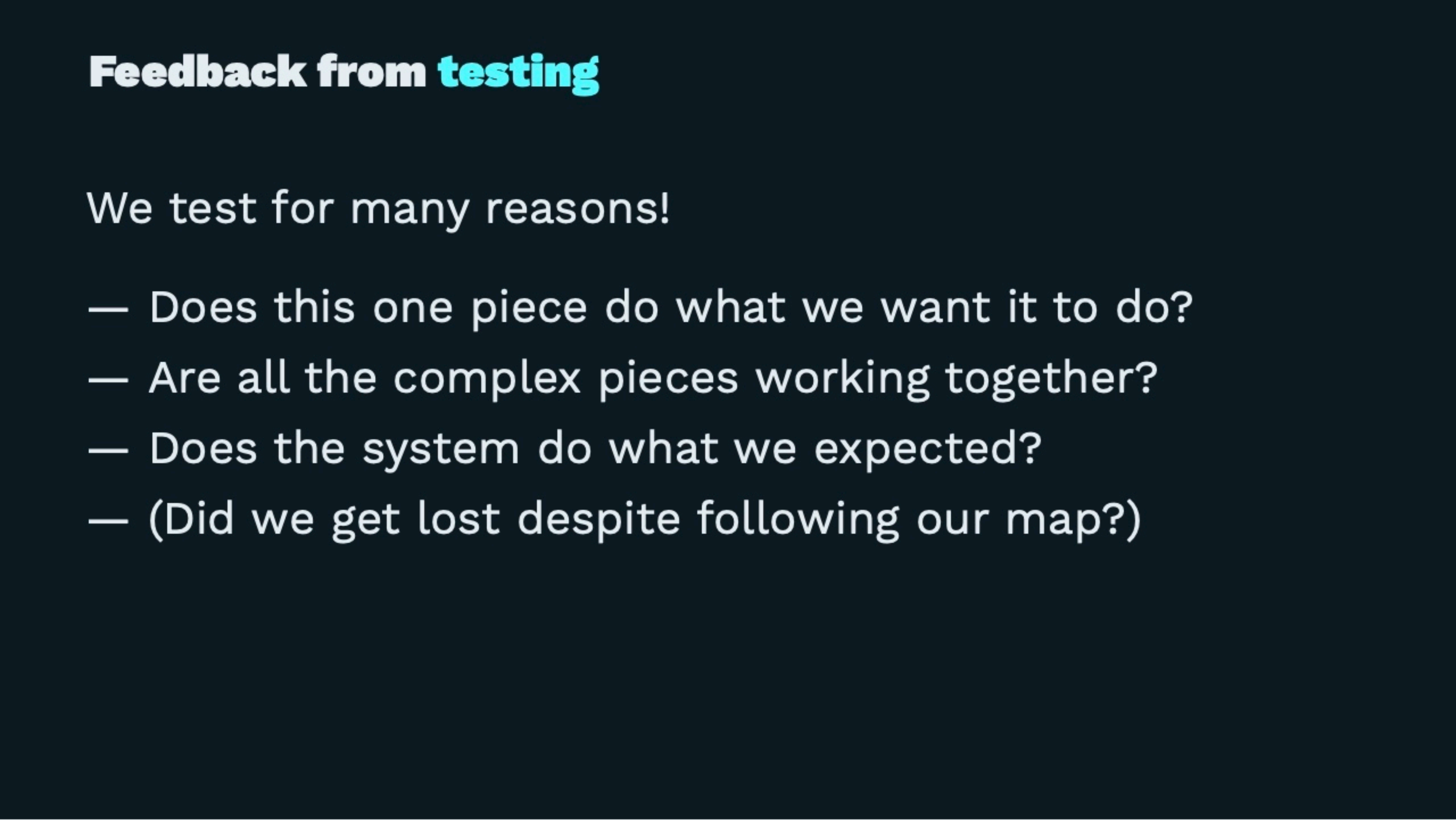
Your mental model is not reality. The map is a model of the real world-- the mountain and the terrain, and the trails across it. The map tells you a trail is there, but it does not tell you that the trail was washed out in a mudslide three days ago. We make our plans with the information we have, and then we learn from feedback how we're wrong.



Ways our map is wrong

- We didn't understand the customer's workflow.
- We got our data models wrong.
- We're transforming our data incorrectly (or inefficiently).
- We figured out a new approach along the way.
- Teams didn't align with each other, and their software doesn't work together.
- Software we rely on behaves unexpectedly.
- We made mistakes while building things.

All of these things are guaranteed to happen, mostly at a small level, but sometimes with very big concepts. So we need feedback and do things to get it.



Feedback from testing

We test for many reasons!

— Does this one piece do what we want it to do?
— Are all the complex pieces working together?
— Does the system do what we expected?
— (Did we get lost despite following our map?)

This is why we have QA.

Feedback from our customers

- Is our system doing what our customers need?
- (Did we reach our planned destination or did our map lie?)

Step 7. Ship it.

Get it into the hands of customers as soon as it would be useful to them. Get revenue as soon as you're able.

The reality of Silicon Valley style software companies is that we all go into debt immediately to be able to pay salaries and AWS bills. We want to get out of that situation as soon as possible, so the company can keep doing its thing.

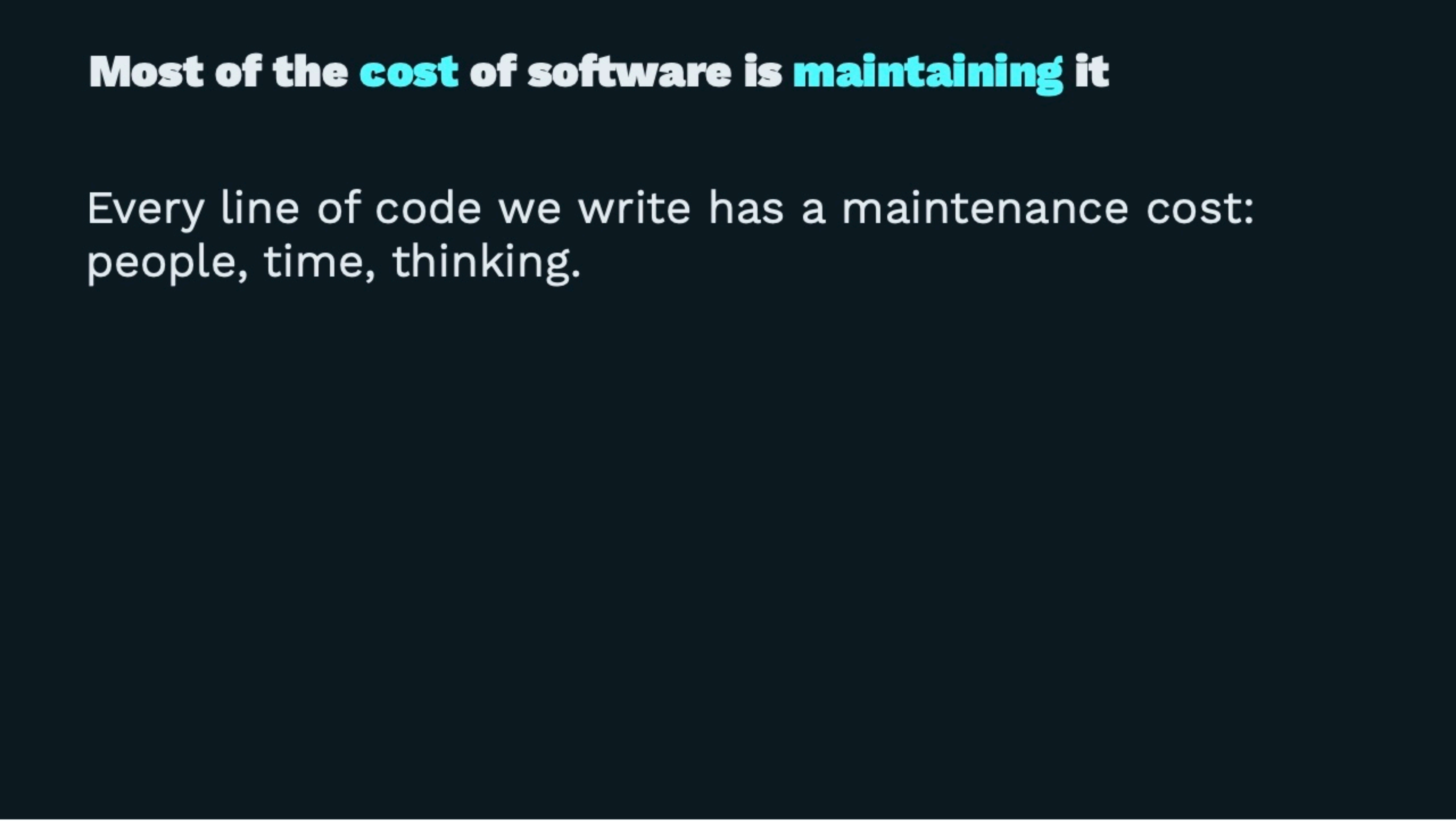
"Ship or die." — Danger, Inc, internal motto, 2002

Before the team shipped the first Sidekick in 2002, we said this often to each other. This over-dramatic motto came from a maniacal focus on shipping, getting our product done and out there into people's hands. But the catch is that you're not done when you ship.

What happens after you ship?

More software.

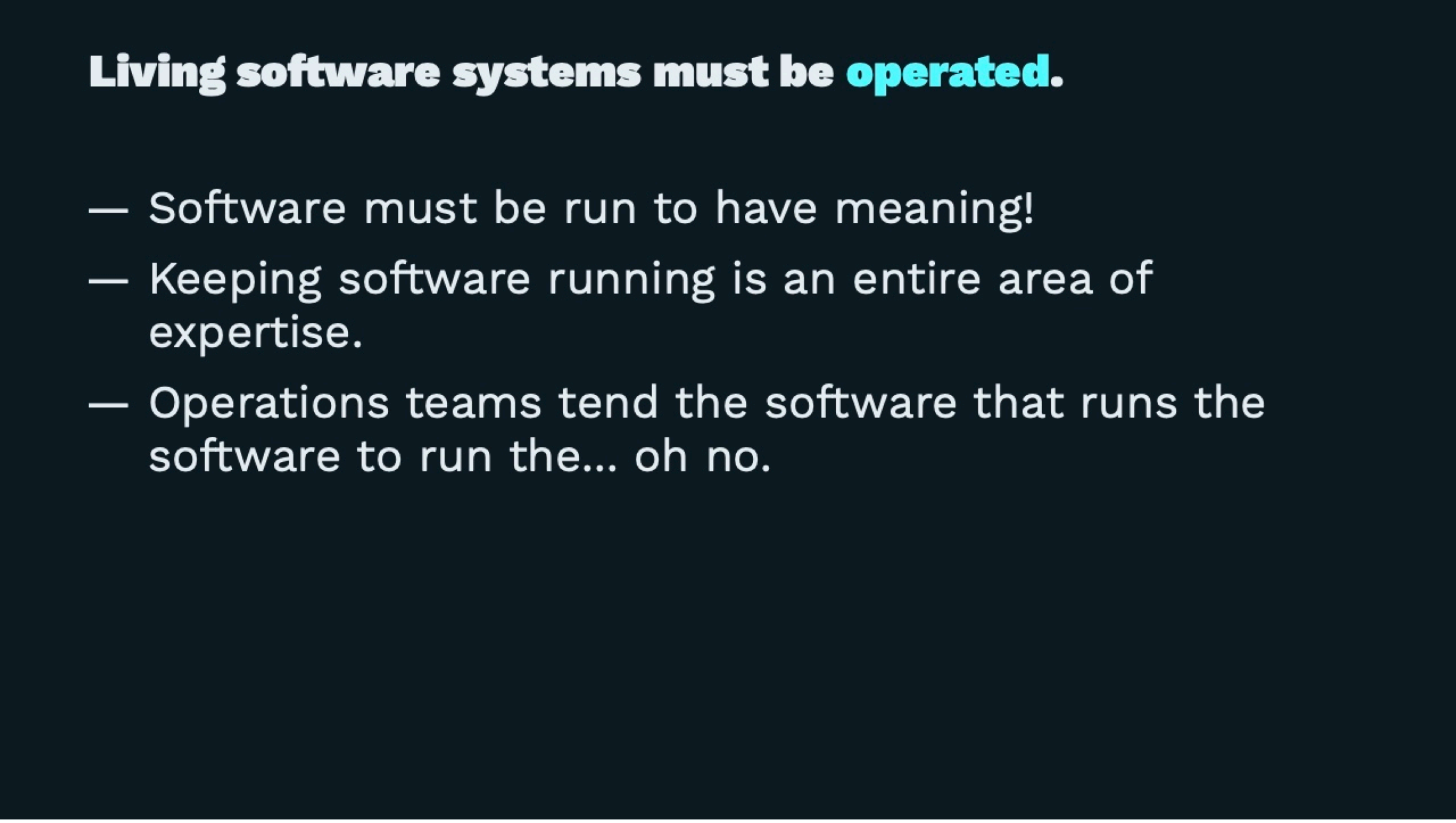
So it's great we shipped instead of dying, but now we gotta keep the software alive too. Software is never finished! We continue to modify it after we release it to the world.



Most of the **cost** of software is **maintaining** it

Every line of code we write has a maintenance cost: people, time, thinking.

Those half-million lines of code represent complexity that has to be understood.



Living software systems must be operated.

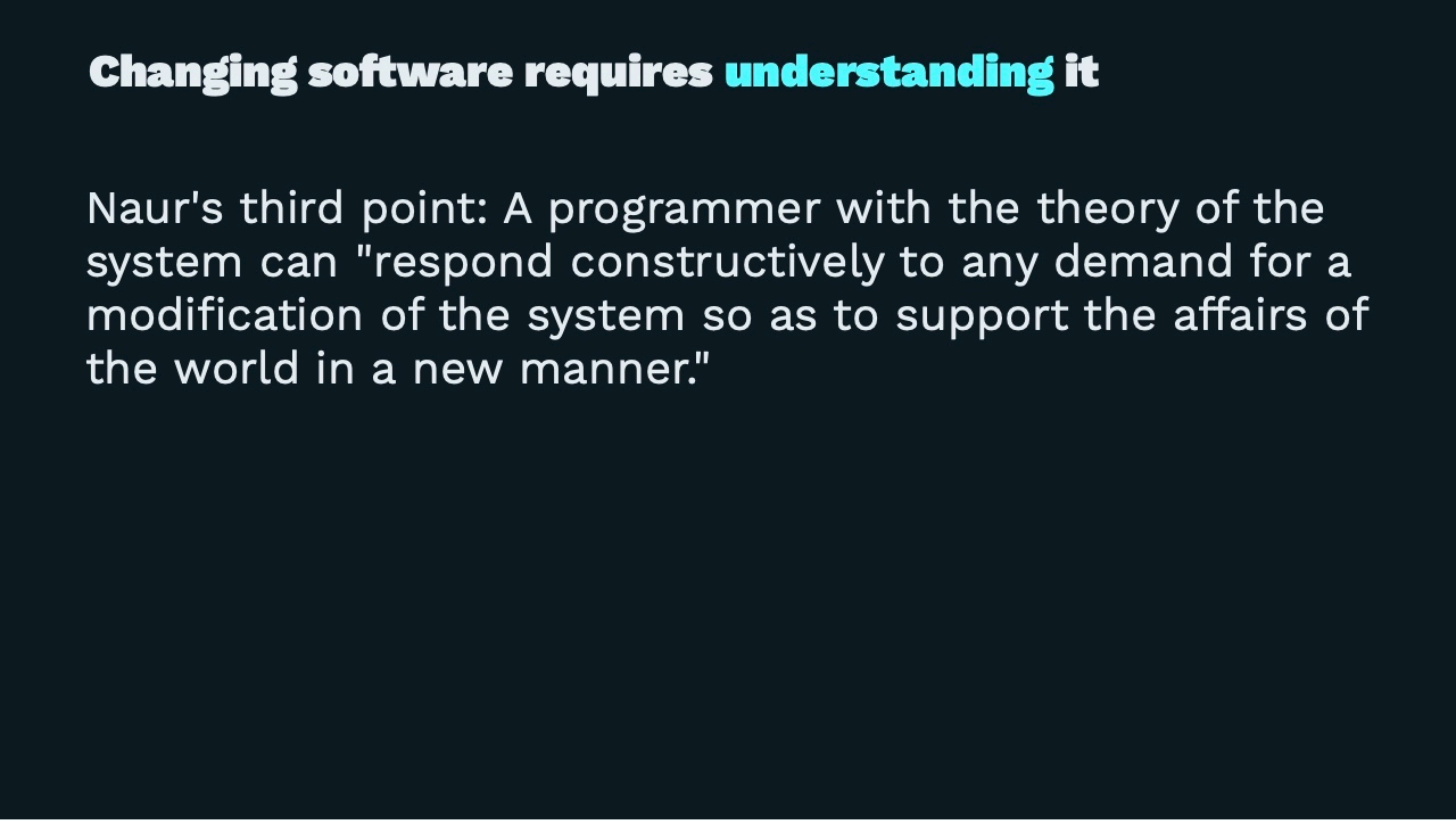
- Software must be run to have meaning!
- Keeping software running is an entire area of expertise.
- Operations teams tend the software that runs the software to run the... oh no.

Text files on GitHub don't do much by themselves.

Living software systems must be changed.

- the world around us changes
- new laws & regulations, new practices from our customers
- the context in which the software runs changes
- the team maintaining the software changes over time

The software must change in response.


Changing software requires **understanding** it

Naur's third point: A programmer with the theory of the system can "respond constructively to any demand for a modification of the system so as to support the affairs of the world in a new manner."

Let's call back to Naur again--changing software requires understanding it. The more complex and voluminous the software, the more there is to understand.

Success can be a catastrophe.

— we need to scale up from a few customers to many
— we learn where you need to be flexible
— we learn where our models were incomplete

A friend who was at Twitter during its early years describes implementing things that would get them through the next six months, by which time they'd have its replacement ready to go.

All software has a lifespan

- the changes made to it slowly build up like plaque in arteries
- the software in a big system usually gets replaced in pieces to keep the system itself working
- the system of software itself lives a long time

Congratulations.
Now do it all over again for the next product.

You figured out how to eat this thing with software. You shipped. Your customers grumble sometimes, but they're mostly happy. PHEW. Let's do a fast recap.

recap: what is software?

- software is, yes, text files with instructions to computers
- it's also an expression of our understanding of a real-world problem
- and an expression of our analysis from a computing perspective

recap: how do we build it?

- there's no perfect answer to this
- building software requires a team to
  - align on their understanding
  - plan an approach
  - coordinate with each other
  - iterate in response to feedback

recap: what happens after we ship?

— software lives on long after we build it
— most of its cost is maintenance
— you have to understand it to maintain it
— eventually we need to replace it

And that's how we turn a **napkin sketch** into something that affects the physical world.

Questions?

Stop sharing screen now.