CONTENTS

1

How Is It That Some Computers Only Tell Time but Others Can Launch Rockets?

It's May 2021, deep in the era of pandemic-scrambled schooling. I'm talking to a classroom over Zoom, and this is a *great* question.

I'm a college professor, though not the absent-minded type baffled by day-to-day concerns of either life or technology. My subject is computer science, and I'm usually decent at things like Zoom, or at least as decent as the average technically savvy person can be at Zoom.

But, that May afternoon, I wasn't speaking to a classroom of college students. Although my campus had returned to a highly constrained form of in-person instruction, I, mercifully, was released from teaching for a maternity leave. I was speaking instead to a group of seven-year-olds as a favor to my sister. She's an elementary school teacher, at the time doing the heroic work of pandemic-teaching the second grade. Her district had

implemented a bonkers system where half the class rotated into school while the other half called in, swapping weekly.

. . . So at least they had a genuinely pretty nice A/V setup to support remote participation. My sister could include the kids at home in her classroom, and I could call her classroom from my home office. She had the idea to use that fancy and frankly unexpectedly functional Zoom Room to stage virtual career visits. I, College Computer Science Professor, was her test particle.*

Seven-year-olds ask great questions. "Are bugs real bugs?" (At least once!) "Why do my headphones break sometimes and why does turning them off and on again fix them?" (We'll cover this.) "Were you always good at computers?" (No.) (No one's good at computers.) (I'm serious.)

And, the question that opened this introduction: "How is it that some computers only tell time but others can launch rockets?"

The answer I gave, except longer and using seven-year-olds' words, was this: Any computer can, theoretically, do anything that any other computer can do. Your iPad could launch a spacecraft; the computers that guided Apollo 11 could theoretically both tell time and play Fortnite. Very, very, very slowly.

I'm not fully convinced that the kids believed me, but it is true.

Their skepticism was reasonable. That all computers are fundamentally the same, and that the moon landers and a fancy new laptop and the dumbest of smart watches are all (in

* Banking on the activity going well, she had already lined up a paleontologist for the following week. I'm glad I went first; I'm no match for dinosaur bones with that (or any, probably) audience.

theory) equally computationally capable sounds like the kind of tidy platitude adults give when avoiding more complicated explanations. "The tooth fairy needs teeth to build her castle!" Seven-year-olds are still young enough to be trusting, but getting old enough to Have Questions. I respect this.

And yet, while the phone in your pocket has more computational power than all of NASA did in 1969, by a lot, they're all fundamentally the same. Your car's entertainment system could theoretically run the software that sequenced the human genome. Your printer could model climate change. I'm not being metaphorical; I'm recounting basic facts that mathematicians started proving back in the 1930s, before transistors were invented, let alone real computers. *Practically* speaking, the computer's speed matters; the printer would take too long to stave off sea level rise. But there's something universal about computation, and all programs are fundamentally doing the same thing: following instructions that manipulate information.

Software's Really Nifty, I Swear

I'm outing myself as a nerd really early in this book, but this is honestly just so cool. Software is such a cool thing that humans invented, mostly out of math, plus some electricity. It's an engineering material, something humans use to build things, but one that can be transformed into almost any electronic thing almost instantly. It's like we invented a type of metal that could be a car door in the morning, a bridge in the afternoon, and a sewage plant's bacterial ecosystem the next day, an idea so weird it's hard to describe. The same electrons running through silicon can be a word processor, a flight simulator, a game, one moment to the next.

This is neat. No wonder it's totally transformed human society in just under sixty years.

. . . And yet . . .

And yet, it can be very difficult to remember how amazing computers are in light of all the inexplicable things they do. Like those headphones that break and then unbreak. Or video calls that echo sometimes, but not always. Or printers that act possessed. I'm not immune to computer-based bewilderment just because I profess the stuff. Consider: A colleague down the hall once needed to print an entirely unexceptional shipping label. He selected the printer in his office (sitting right next to him!) from a standard Print . . . menu dropdown and then hit Print. For reasons neither of us ever managed to track down, the shipping label came out of the printer in *my* office. This happened three times. The fourth time he did the exact same thing, hoping (the definition of insanity!) that this time something different would happen, the printer sitting next to him straightforwardly roared to life and produced the shipping label.

Why does software that can theoretically do anything so often fail to do the thing you actually want it to do? Why is it all so *baffling*?

As a college professor of computer science, especially one who studies software and software quality, I get a lot of good questions, and not just from my own students, or my sister's second-graders. Friends, family, neighbors, fellow parents at the kindergarten meet-and-greet who casually discover what I do for a living. As a professor, a major component of my job is research, which I do in collaboration with my PhD students as I train them. But people by and large don't ask me about that, which is understandable; the nitty-gritty of analysis techniques for software quality assurance is (tragically?) not as interesting to everyone else as it is to me. Instead, people ask why their

printer hates them. Or why their phone gets slower over time. What Bitcoin is, and whether they should buy some. Cousins corner me at family weddings to demand to know why their phone sometimes thinks they live in Newark, when they, citizens of France, have never set foot in New Jersey. And is the "smart" TV spying on me? Could it be, though?

I love these kinds of questions. I mean, I'm not always equally enthusiastic about debugging someone's specific Excel weirdness over meet-and-greet doughnuts, but the questions themselves are often great, because the answers are often pretty revealing about how software works and why it sometimes doesn't. I like bringing this kind of clarity to people's lives. Despite the fact that it's become so ubiquitously ingrained into our daily existence, most normal, nontechnical people don't really have a good sense of what software actually *is* or how it works.

This is entirely fair. I mean, I don't know how most of the other kinds of engineered stuff I run into every day works, either. It mostly just doesn't come up, though, because roads and roundabouts and electrical systems and so on blessedly mostly just work. But software's just so *weird*, you know? Roads don't pull nonsense on the level of a phone app. They get potholes sometimes, but mostly they just lie there. We have this idea that code is fundamentally logical or mathematical, but software's constantly doing profoundly perplexing things.

That paradox, and those questions, might seem like they should have strictly technical explanations, given that we're talking about computers and all. And there are certainly technical factors at play. But software's not physical, unlike many things we engineer (bridges! roundabouts!), which means there are no neatly practical laws of physics keeping anything in line. Most of the constraints around the software we build

and how we use it come from the people doing the building and the using. And people are, frankly, pretty weird.

That all means that the real answer to "Why does my phone do that?" often lands at the intersection of technical and human concerns, like economics, incentives, history, (mis)communication, assumptions . . . That kind of thing.

So, I'm going to tell you a story about how software works and why it sometimes doesn't. This will involve explaining some of the technical parts—what software is, how code works, why it can be weird to build with—because they're both relevant and (in my admittedly biased opinion) genuinely interesting. But it's also a story about people, because understanding how software works is less about code alone than it is about how code and people interact. I'll touch on the history of certain big technological advances, including decisions that we're now stuck with whether or not they were once reasonable, and the various economic pressures and human disagreements that got us here. After sixty years, we've ended up with an engineering medium that can land spacecraft and also can't reliably print a shipping label. The wonder and the weirdness come from the same place.

That's what this book is about.

A Quick and Painless (I Promise!) Programming Primer

I'm writing this book assuming you don't know how to write computer programs. It's also OK if you do. Maybe you did a little bit of programming in a statistics class twenty years ago, or maybe you're in a field where you have specialized programming skills. Maybe you're a full-fledged software engineer here to see how wrong I am. Honestly, and this is a bugbear of mine,

many more people have exposure to programming than popular discourse would lead you to believe. Spreadsheets (the kind in Microsoft Excel or Google Sheets) provide really complicated formula and data manipulation tools, and using them to process financial data or produce reports or do all sorts of similar things is *absolutely* programming. There's a programming language underlying the formulas, equivalent to any other programming language (because, as mentioned, computation is universal). If you do that kind of work, you're a programmer, no matter what anyone else calls you.

But I digress. In case you're a reader who's never looked at a line of code in your life, I'll give a quick and not-so-difficult primer on how computer programs work. It will help somewhat with the rest of the book.

(That said, if your eyes start to glaze over, I give you explicit permission to skim. The next section is titled "Don't Worry, This Book Isn't About Programming Anyway.")

A program is a set of instructions describing how to solve a problem. In some ways, it's like a cooking recipe. A cook is like the *central processing unit* (CPU) in a computer, in that they follow and execute the instructions. Most modern computers have more than one processing unit or *core*; that's not unlike having more than one cook in a kitchen.

A chef can't produce food just by reading a recipe. They need ingredients. Similarly, a CPU needs information to process, and so a major thing that programs do is keep track of data, using *variables*. A variable is like a labeled jar that stores something; the label gives it a name that we can use to get access to whatever it's storing.

For example, imagine we're writing a breakfast recipe program. Before starting to cook, we might have the program note how many eggs we're starting with:

```
1.  eggs_in_fridge = 12
```

This is an instruction, not a statement of fact, because = (equals sign) means "store this here," not "these things are equal."[†] It's not saying "there are 12 eggs in the fridge"; it's saying "take the number 12 and remember it using the name eggs_in_fridge." I wrote the number 12 in this (fake) program, but it could also come from the keyboard or hard drive (like a saved spreadsheet) or memory (or the fridge, to abuse the analogy).

Having stored the number with the name, we can then reuse it:

```
1.  eggs_in_fridge = 12
2.  eggs_needed_for_recipe = 3
3.  eggs_remaining = eggs_in_fridge – eggs_needed
```

That last line tells the program to do some math (12 minus 3, the numbers stored in the names) and store the result (9) in a variable named eggs_remaining. Textbooks often use single letters for variables in programming examples, like in algebra (x, y). But variables can mostly be named whatever. Using words, like I did just now, can make it easier to figure out what a piece of code is doing (egg inventory management, here).

Computer programs don't just put data in jars for no reason; they also do things with that data, like arithmetic (subtraction,

[†] I *swear* there's not very much code in this book. The code that you will see is what's called *pseudocode*, or fake code that resembles many real programming languages without actually being one. Computer scientists and programmers use it all the time to sketch out ideas without getting bogged down.

That said, in many programming languages, actual equality, like from math class, is denoted with == (two equals signs), while the single = indicates assignment. The answer to "uh, wait, what, why???" is "historical reasons," a.k.a., it made reasonable sense once Because Reasons, and then it stuck.

like we just did, but also addition, multiplication, and so on) or logic. Logical operations are things like AND, OR, or NOT. So if we have two instructions like:

1. A = true
2. B = NOT A

. . . the variable B after that (even more pointless) program is executed stores the value false.

Even though the core operations that a computer can perform in its electronics are ultimately fairly simple, you can do most math by combining them. That is, you can turn a deficient calculator that only adds into a calculator that also multiplies by adding repeatedly (3 * 3 is the same as 3 + 3 + 3). CPUs do almost *exactly* that to build complicated math out of basic electronic operations. It's as though recipes could only be written as a series of instructions like "add one gram of ingredient to bowl" or "stir once" or "bake for 1 minute," repeating as necessary. Tedious, but doable.

Programs can also make choices. "If you have parmesan cheese, grate it. Otherwise, skip to the next step." For programs, those choices are usually based on the results of arithmetic or logic. Sticking to cooking, this might look like:

1. IF (allergic_to_nuts):
2. . . . code to add chocolate chips . . .
3. ELSE/OTHERWISE:
4. . . . code to add walnuts . . .

"if [condition], then do [thing]; otherwise do [some other thing]" lets you chain together instructions for cooking and computing alike based on whether something is true or not. In this case, the chef-CPU does something different based on whether the variable allergic_to_nuts is true or false. Your

phone runs code like that to see if it should buzz or ding based on whether you've silenced notifications or not.

Finally, programs can do things repeatedly. This is called a *loop*. Thus "while you still have eggs, break an egg into a bowl" becomes something like this:

```
1.  eggs_in_bowl = 0
2.  WHILE (eggs_remaining > 0):
3.      eggs_in_bowl = eggs_in_bowl + 1
4.      eggs_remaining = eggs_remaining – 1
```

. . . which keeps going until you're out of eggs. This lets programs process millions of pixels without someone having to write an instruction for each pixel individually by having code that loops over all the pixels on the screen.

These basic building blocks sound pretty removed from anything anyone actually does on a computer. But it turns out that you can get surprisingly far by combining math and logic. Consider the game Candy Crush. A game board is data stored in variables that describe which candy is in which spot. When you tap a candy to do a swap, the game does logical checks on the data to see if they match and should "disappear." If so, a loop runs over the pieces on the board updating the data and doing arithmetic to change your score, also stored in a variable. Meanwhile, programmers have agreed on ways to match colors to numbers, so "drawing" a purple candy piece in colorful graphics boils down to looking up the number for purple and telling the correct pixel variables to store that value.

None of the individual operations are complicated. There are just *a lot* of them, organized reasonably carefully. The "a lot of them" part is one way my programs-as-recipes analogy is misleading, because even uninteresting programs are *much* longer

than a recipe. Something like Candy Crush requires logic and instructions for playing the game, generating the levels, tracking your score, and so on, amounting to at least a few orders of magnitude more instructions than a recipe.

One way programmers keep this manageable is by organizing code into reusable pieces. Just like my favorite pressure cooker cookbook says "see page 363 for chicken stock recipe," code can be written once, given a name, and then used again somewhere else. These pieces of code are referred to in different ways, including as *procedures*, *methods,* or *functions*.

This name *function* is inspired by programming's mathematical roots, where a function is a relationship between a set of inputs and a set of outputs. That's one way that functions are more powerful than cookbook cross-references. A recipe for chicken stock is always the same. A function can be more like "make chicken stock for soup for X people," such that you could shout how many people are coming for dinner at page 363 and have it adjust accordingly. Even better, functions can do something and hand you back a result, like the right amount of finished stock.

Let me switch from cooking to geometry for a moment. Remember right triangles (Figure 1)?

If you want to know the length of the hypotenuse (that's c, the long side opposite the right angle), the Greek mathematician Pythagoras worked out a formula:

$$c^2 = a^2 + b^2$$

If you want to figure out what c is, you can take the square root, which gives $c = \sqrt{(a^2 + b^2)}$. c is the "output," or the thing you want to compute (the hypotenuse), so this formula explains how to compute it from a and b. Those are the other two sides, or the *inputs* (or *arguments*) to the formula.

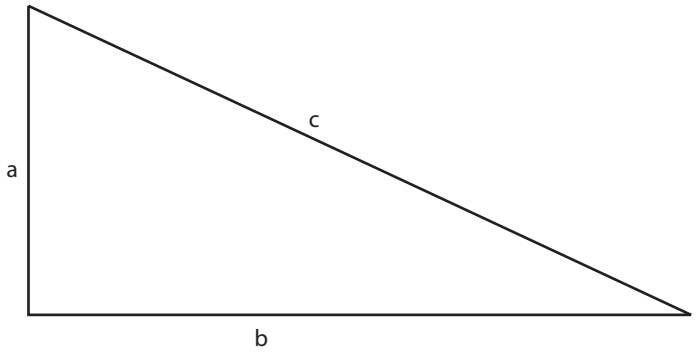

FIGURE 1. A right triangle, with sides *a* and *b*, and hypotenuse *c*, just as Pythagoras intended. It's possible you remember this from high school geometry, but your computer definitely doesn't.

Computing hypotenuses is a thing that can come up while programming, like in video games (where it can be useful for physics simulation or collision detection) or graphics software (where distances or angles are often relevant). If you're writing a program like that, you don't want to write this calculation every time you need a hypotenuse. Instead, you'll write it once and name it:

```
1.  FUNCTION ComputeHypotenuse(a, b)
2.      c_squared = (a × a) + (b × b)
3.      c = square_root(c_squared)
4.  RETURN c
```

Now, anywhere in a program that you need a hypotenuse, you can just write `result = ComputeHypotenuse(3, 4)`. If you had variables `triangle_height` and `triangle_width` saving the lengths of the other sides, you could even write `result = ComputeHypotenuse(triangle_height, triangle_width)`. The function does the work and hands back the answer,

which is saved in the variable `result`.[‡] It's like having a sous chef. You don't need to know how they make their perfect hollandaise, just how to ask them to make more.

This keeps things organized and easier to read. If I see `ComputeHypotenuse` in a program, I can immediately guess what's going on. I don't have to step through the individual instructions to figure it out: "OK that's `c_squared`, hmm, why would I take the square root? Oh! . . ."

Speaking of, I snuck something in there on you: `ComputeHypotenuse` uses another function called `square_root`. I didn't show you how `square_root` works, but it didn't matter for the example, because we can guess based on the name. `square_root` in turn probably calls other functions, which can call other functions, and so on. There's effectively no limit, and given that any of those functions can be given input to customize what they do, this whole situation ends up being a lot more complicated than recipes (even if, in both cases, we're following instructions to do things to ingredients/data).

I'll probably never remember how the code for `square_root` actually works (it's surprisingly complicated!), and that's fine. I can still use it. Even better, and very much unlike sous chefs, code is just text saved on a computer, which means it's extremely easy to copy. Once someone has written `square_root` (or `ComputeHypotenuse`), there's very little reason to bother writing a new version. Because this is incredibly convenient, programmers organize useful functions into what we call *libraries* to reuse and share, like a shelf full of cookbooks. I don't know how much code is in Candy Crush, but a lot of it is for functionality that's been done a million times before in other games or

[‡] Not all functions need arguments. A function that checks what time it is doesn't need any input, it can just go look.

apps—drawing letters on the screen, paying for microtransactions, posting updates to Facebook. Most of that isn't particularly Candy Crush–specific, and most of it is probably reusing code from a library.

So, that's as much as you need to know for now. Computers execute programs, which are made of instructions that boil down to math and logic. You can build surprisingly complicated things out of simple pieces by chaining instructions together, controlling when and how they run with concepts like if/then or loops, and organizing them into functions. Programmers organize useful functions into useful collections and then share and reuse them everywhere, avoiding the need to reinvent the hypotenuse. And because programs are just text on a computer, that's really easy to do, an idea we'll very much get back to.

Don't Worry, This Book Isn't About Programming Anyway

Now that I've covered the basics, I can explain why you don't need to know more than that for the purposes of this book. Namely, programming and software engineering are not the same thing, and this book is *mostly* about the second one: software as a funny material that we engineer *with*.

Programming is writing code, or the act of typing out instructions for a computer, like the ones we just talked about. These are written in specific languages that are designed to express those kinds of instructions.

Software engineering is a group of people building and maintaining Gmail, a giant collection of code that does a very large number of things and needs to interact with the reality of billions of users all across the planet in many different time zones and languages.

It's the difference between being able to write grammatically correct sentences and writing (or analyzing or understanding) a novel. Writing nice sentences is a legitimate skill, and you need to know how to compose reasonably nice sentences in some reasonable order to write a book. It's not as though the skills are *completely* unrelated. But writing a novel also involves things like pacing, narrative, thematic coherence, and so on. Book engineering includes and relies on sentence engineering, but also goes beyond it.

The same is true here. Programming is the sentences; software engineering is the novel construction. So software, and its engineering, and how it works and why it's weird, are about a very large number of things, many of which are not actually about programming. Not knowing much about the latter doesn't preclude understanding things about the former.

I'll end up using the words "software" and "program," or "software engineering" and "programming," somewhat interchangeably, the same way you might say you're writing when you're really "crafting a novel" or "desperately sprinting toward a [nonfiction, in my case] book deadline." The boundaries are blurry and not very important. When I talk about the weirdness of software, I'm really talking about the whole ecosystem of code, systems, engineering practices, social dynamics, economics, and only sometimes the actual act of programming.

That said: Software, made of endlessly malleable code, tells computers, lovely universal computing machines, what to do. It's a material that we build things out of, and so we often call the activity surrounding the building "engineering," as in "software engineering."

Engineering is derived from the Latin *ingenium*, meaning "cleverness," and *ingeniare*, meaning "to contrive, devise." *Merriam-Webster's Collegiate Dictionary*—a physical copy of

which I have sitting on the bookshelf in my office because, again, I'm a nerd—defines it as "the application of science and mathematics by which the properties of matter and the sources of energy in nature are made useful to people." I personally prefer a somewhat more nuanced definition[§] as "the application of codified (ideally scientifically informed) knowledge, under constraint and in the face of uncertainty, to advance the human condition (or otherwise solve human problems)."

I want to unpack that definition a bit, especially to distinguish "engineering" from either science or craftsmanship. Scientists, broadly, discover how the world works: Force equals mass times acceleration. Engineers use that knowledge to build bridges that won't collapse when trucks drive over them. They may in the process discover or develop new ways to do that, doing science along the way, but the discovery of new things isn't the point. The point is to solve problems. Craftspeople solve problems too, but using individual skill, intuition, and experience, like a master blacksmith who can judge the temperature of steel by its color, or a chef who can tell the salmon is done by feeling how it responds to pressure from the spatula.

Engineering doesn't *require* scientific knowledge; Roman engineers built roads and aqueducts that still stand today using rules of thumb, accumulated experience, and systematic approaches that had been passed down through generations. What made it *engineering* rather than *craft* is that this knowledge was understood systematically and had been refined into

[§] I learned approximately this, and especially though not exclusively the distinction between engineering and craftsmanship, from my genuinely esteemed Carnegie Mellon colleague Mary Shaw, who has forgotten more about engineering than I will ever know.

repeatable methods that could be applied to new problems (or roads).

The bits about constraint and uncertainty are important, too, because engineers don't work in the frictionless vacuums of physics textbooks. Bridge builders are constrained by the properties of steel and concrete, local geology, and the local township's budget. They face uncertainty when it comes to wind, weather, and contractor timelines. Codified knowledge helps because it lets them make informed predictions about those uncertainties. Steel has known properties; gravity exerts a certain force. Engineers use that kind of information when designing and building solutions to human problems, out of all sorts of engineering materials.

Humans have long been discovering and inventing useful and often fascinating materials as well as ways to solve problems with ("engineer") them. Consider roads, perhaps humanity's oldest engineering project. The basic idea of making the ground easier to walk on arose very early, approximately when humanity realized that trudging through mud is suboptimal. The ancient Mesopotamians started paving things around 4000 BCE, presumably because cart wheels and sand don't mix well. We've refined both the materials (the Romans had layers of progressively sized stone; the Industrial Revolution brought us macadam roads, named after their Scottish inventor; in the late 1800s, we developed asphalt-based paving) and the processes for putting them together, so now we have quite a well understood process for road-building, and a general understanding of how those roads are likely to hold up under time and weather.

Humans also build impossibly tall buildings. We make all kinds of useful chemicals in very large quantities. We've invented airplanes, giant machines that weigh more than 30 elephants and can fly through the air at 550 miles per hour. We've built

bridges over 20 miles of bay and trains that can travel at fantastic speeds between Tokyo and Osaka, 320 miles apart, in less than 2.5 hours. We've discovered new ways to grow giant silicon crystals and then etch tiny patterns in them (really!) to create the powerful processors that form the basis of our often-irrational computing machines. This doesn't even begin to touch on medicine, biomedical engineering, robotics, the works! Human engineering ingenuity is really just terrifically impressive.

So, What's "Software Engineering"?

Software is also something humans nominally engineer; there were something on the order of 2.5–3 million people with a job title like "software engineer" in the United States as of 2022, and many more with adjacent titles. Software systems are built out of code to solve human problems. Programming itself has its roots in mathematics, suggesting it's informed by science. There's certainly plenty of codified knowledge about how to engineer software. I mean, I'm a professor who teaches whole classes on this, with course names like "Foundations of Software Engineering," where we cover things like systemic knowledge for engineering software systems.

But it's also a funny thing to talk about in this way, at least compared to everything *else* we engineer. For one, software isn't physical in the way that silicon or asphalt or airplanes are. You can't touch it or watch it rust. It can be copied a million times without degrading or changing. It can vanish if you forget to "save" it, whatever that means, but it can also exist in a million places simultaneously. It's something like engineering with information, or thought. This is both lovely and exhausting. Thoughts can be exceedingly chaotic compared

to physical things. I mean, a piano doesn't sometimes stop being a piano for a few minutes and then suddenly remember that it's a piano again after being opened and closed a few times. Asphalt doesn't become a bridge, and when a road fails, it's because physics or weather happened, not due to some kind of existential crisis. Physically engineered artifacts aren't manifestly bewildering the way software so often is, day-to-day.

A (Brief) History of Software Angst

Calling software "engineering" is pretty fraught, historically. Software is also, historically, a nightmare; these two facts are somewhat related. If it makes you feel any better about our current circumstances, the software situation has never really been under control.

The name "software engineering" was first officially put forward in 1968, at a conference held by NATO—yes, the military alliance—in Garmisch, Germany, convened specifically to talk about software, and specifically about the fact that it was a disaster.

The disaster had developed as follows: Computer hardware was advancing so rapidly that software development wasn't keeping up. Computers in the 1950s weren't very powerful, so single humans could still understand an entire programmed system. If something went wrong, someone could trace through and fix it in an afternoon. One reason engineering disciplines are useful is that they give us tools for managing process and risk even when the thing we're building is really complicated; no one understands the entirety of a skyscraper. The initial computers and their software were simple enough to not meaningfully require engineering.

But by the 1960s, the computer hardware was getting much faster, meaning the software was getting a lot more complicated. Programmers started writing code for things like airline reservation systems, air traffic control, banking, or military command and control. This meant that the software's complexity very rapidly outstripped our ability to deal with it. Projects were routinely late, over budget, and just generally broken. The IBM System/360 operating system, one of the most ambitious software projects of the era, had been budgeted to require twelve "program designers" leading a team of sixty programmers. It ultimately consumed 5,000 person-years of effort and shipped more than a year late, with bugs. Programs would work fine in testing and then fail catastrophically in the field. The Mariner 1 spacecraft was destroyed shortly after launch in 1962 due to what amounted to a typo in its guidance software (though the actual error became somewhat mythologized; it wasn't *quite* as stupid as this makes it sound). Documentation was often nonexistent, or mostly wrong, or both.

The NATO conference report,[1] which is something like the account of a group therapy session, makes for pretty fun reading, if you're a certain kind of nerd (I am). It's full of politely despairing quotations about the state of affairs: "The basic problem is that certain classes of systems are placing demands on us which are beyond our capabilities." The conference taxonomized the dysfunction: poor project estimation, inadequate quality assurance, communication failures between teams, and what they delicately called "the problem of scale." Programming had been *fine* when programs were 1,000 lines of code. It was very much *not fine* now that programs were reaching 100,000 lines of code. Meanwhile, today's systems are often easily millions of lines of code, which the conference participants probably couldn't even imagine.

I can never decide if it's a profound statement about how difficult the problem is, or just a profound indictment of my field, that the situation they're describing sounds so familiar to users and researchers and engineers of software today. Modern software projects still often run late, go over budget, and fail.

The conference attendees explicitly debated whether the terms "crisis" or "software crisis" were accurate. Some participants considered it a touch melodramatic. Regardless, though it took a few years to catch on, the phrase became strongly associated with the conference. It was, in effect, branding via strategic catastrophizing. "Software crisis" ended up being sufficiently evocative to change the way many people (computer scientists and otherwise) thought about the challenges of building software.

The conference report also popularized a second phrase: "software engineering." The conference organizers surveyed the state of the (software) world, found it depressing, and argued that software should be treated as a legitimate artifact that is legitimately engineered. They weren't describing the world *as it was*: Software was *not* being constructed using codified, systematized knowledge. They were being *aspirational*, saying that software *should* be built in ways that paralleled other engineering disciplines. They wanted it to be something other than an art form, or afterthought. The report opens by explaining that they deliberately chose the term "software engineering" as "being provocative . . . in implying the need for software manufacture to be based on the types of theoretical foundations and practical disciplines that are traditional in the established branches of engineering."

This was in fact a provocative thesis: Attendees spent a nontrivial amount of time arguing about whether software development could ever really be engineering, given that programs

(at least at the time) lacked the predictable materials of "real" engineering. Corny as it sounds, this argument continues, more than sixty years later.

As in many things, a pause to give credit to the woman who said it first: Margaret Hamilton, who led the team that wrote the software for the Apollo missions, had been using the term "software engineering" informally since at least 1965. She'd been trying to get people to take programming seriously, while making sure the moon landing guidance software didn't accidentally crash the moon lander into the moon. She also wrote, decades later, about the thing I was talking about earlier wherein programming and engineering are different, and knowing words doesn't mean you can write a novel.

Anyway, the point is, the NATO folks put the term "software engineering" out there aspirationally. It's like calling your garage band "the royal philharmonic" to trick yourself into learning to play the violin. The goal was to convince everyone to take software seriously and develop ways to make it more like other, real engineering disciplines—the kinds that build things like bridges, buildings, chemical plants—with their established processes, validation methodologies, and reliability metrics. They also acknowledged that software itself has important differentiating properties from the other materials out of which humans engineer things. The fact that it's made out of nothing ultimately means that it challenges conventional engineering wisdom in any number of ways. The overall argument was that we needed to come up with new, different wisdom for this special new stuff we were starting to build with.

Naming the problem, unfortunately, wasn't enough to solve it. A follow-up conference in Rome in 1969 aimed to build consensus on potential solutions but ended up being, in the words of its report's editors, "far less harmonious" than the first,

dissolving into disagreement and miscommunication. Meanwhile, hardware costs continued to fall exponentially, and software costs were rising exponentially, and it was all tremendously unfortunate. In 1972, Edsger Dijkstra, a well-known computer scientist, said, pithily, "The major cause [of the software crisis] is . . . that the machines have become several orders of magnitude more powerful! To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming [has] become an equally gigantic problem."[2]

I regret to report that computers have only gotten bigger and faster since. No wonder we're still stuck.

Software Cannot Be Fixed by Math or Logic Alone

Dijkstra was brilliant and influential, but also famously (allegedly) unpleasant to deal with. He spent a lot of time writing a lot of papers (in fountain pen, numbered sequentially), many of which were centrally concerned with the fact that everyone else in computer science was doing everything wrong. He clearly agreed with the NATO report's diagnosis of the problem, based on his characterization of that same problem. Despite that, he was extensively disdainful of software engineering as a discipline throughout his long life and career.[¶] He thought that it was too focused on managing complexity rather than avoiding complexity in the first place.

[¶] Quoth EWD1036 (1988), verbatim: "As economics is known as 'The Miserable Science,' software engineering should be known as 'The Doomed Discipline' . . . software engineering has accepted as its charter 'How to program if you cannot.'"

I disagree with him, for whatever that's worth. One thing I think the report was too pessimistic about, though, is a thing I *expect* Dijkstra would have agreed with me on. Namely, the NATO conference attendees weren't sure software was engineering because it lacked the theoretical foundations of traditional disciplines. And . . . that wasn't *quite* true, even then. Researchers had already begun to work out the mathematical theory behind computation, they just hadn't fully connected it to programming yet. The fact that we use the word "functions" in both isn't a coincidence. The reason I think Dijkstra would agree with me is that he helped make those connections in his work. Among other things, he developed key ideas connecting programming to logic, including how to write programs and then mathematically prove (some) things about their behavior. That's all based in the kinds of theoretical foundations the NATO attendees ostensibly wanted.

Unfortunately, though, mathematical foundations aren't quite enough to call software "engineering." Sorry, both Dijkstra and NATO. Yes, we can prove things about programs, because computation has elegant mathematical properties. But none of that math explains why or how Facebook accidentally deleted itself from the Internet in 2021, nor why my colleague's print job showed up in my office three times in a row but not a fourth. The math says what's possible; it doesn't say what humans will actually do with that possibility. Reality is chaotic, no matter how beautiful the math is.

Software's problem is first that it's weird to build things out of ideas and electricity, and the math isn't enough to fully make it not weird. And second, building things entirely out of human ideas to solve human problems means that software engineering involves navigating human decisions, miscommunications, assumptions, and economics, entirely without any kind of

physics to rein it in. Most of the weird things software does can be explained by the intersection of those facts.

This Book: Long Answers to Good Questions

So the questions that people ask me about the offbeat thing their phone has been up to lately aren't strictly technical. They're usually at least a little bit technical, but they're hard to answer fully without getting into not just what software's made of, but how, and why. In short, the human decisions that created it, and the human world in which it exists. I'm a professor, so I'm not good at short answers anyway, but this one's not really my fault.

Each chapter of this book *starts* with a question of the sort people ask me at dinner parties. Questions like, "Is my phone spying on me?," "What do all those people who work for Google do all day, anyway?," and, most mysteriously, "Why is my printer acting possessed?" They're all real questions I've been asked, usually multiple times, though not exclusively at dinner parties. Then, instead of giving only the highlights to avoid monologuing at the dinner table, I try to answer them properly. The long answers explain not just what's *normal* about software as a thing we engineer with, but also the things that are weird about it as an engineering medium (how it works, and how we build stuff with it). And, they let me explain many of the *non*technical reasons why the software in your life is often kind of bizarre.

The high-level answer, which I'm going to give through nine more chapters of questions from my confused friends, family, and acquaintances, is that when you can build essentially anything out of essentially nothing, it's very easy to build the wrong thing, and hard to tell. Even without inherent physical constraints, you

INDEX