35

Estimate Honestly and Fairly

8. I will produce estimates that are honest both in magnitude and precision. I will not make promises without reasonable certainty.

In this chapter, we're going to talk about estimating projects and large tasks that take many days or weeks to accomplish. This is not the same as the Agile practice of estimating small tasks and stories, which I describe in the book *Clean Agile*.[1]

---

1. [Clean Agile].

Knowing how to estimate is an essential skill for every software developer; and one that most of us are very, very bad at.

The skill is essential because every business needs to know, roughly, how much something is going to cost before they commit resources to it.

Unfortunately, our failure to understand what estimates actually are, and how to create them, has led to an almost catastrophic loss of trust between programmers and businesses.

The landscape is littered with billions of dollars in software failures. Often, those failures are due to poor estimation. It is not uncommon for estimates to be off by a factor of 2, 3, or even 4 or 5. But why? Why are estimates so hard to get right?

Mostly it's because we don't understand what estimates actually are, and how to create them.

You see, in order for estimates to be useful, they must be honest. They must be honestly accurate, and they must be honestly precise. But most estimates

are neither. Indeed, most estimates are lies.

Lies

Most estimates are lies because most estimates are constructed backward from a known end date.

Consider Healthcare.gov, for example. The president of the United State signed a bill into law that mandated a specific date when that software system was to be turned on.

The depth of that illogic is nausea inducing. I mean, how absurd. Nobody was asked to estimate the end date; they were just told what the end date had to be—by law!

So, of course, all estimates associated with that mandated date were lies. How could they be anything else?

It reminds me of a team I was consulting for about twenty years ago. I remember being in the project room with them, when the project manager walked in.

He was a young fellow; perhaps 25. He'd just returned from a meeting with his boss. He was visibly agitated.

He told the team how important the end date was. He said: "We really have to make that date. I mean, we *really* have to make that date."

Of course, the rest of the team just rolled their eyes and shook their heads. The need to make the date was not a solution for making the date. The young manager offered no solution.

Of course, estimates in an environment like that are just lies that support the plan.

And that reminds me of another client of mine who had a huge software production plan on the wall—full of circles and arrows and labels and tasks.

The programmers referred to it as *The Laugh Track*.

In this chapter, we're going to talk about real, worthwhile, honest, accurate, and precise estimates. The kinds of estimates that professionals create.

Honesty, Accuracy, Precision

The most important aspect of an estimate is honesty. Estimates don't do anybody any good unless they are honest.

The most honest estimate you can give is: "I don't know." But that estimate is not particularly accurate or precise. After all, you do know *something* about the estimate. So the challenge is to quantify what you do *and don't* know.

First, your estimate must be accurate. That doesn't mean you give a firm date—you must not dare be that precise. It just means that you name a *range* of dates that you feel confident about.

So, for example, "Sometime between now and 10 years from now" is a pretty accurate estimate for how long it would take you to write a hello world program. But it lacks precision.

On the other hand, "Yesterday at 2:15 a.m." is a very precise estimate, but probably not very accurate if you haven't started yet.

Do you see the difference? When you give an estimate, you want it to be honest both in accuracy and in precision. To be accurate, you name a range of dates within which you are confident. To be precise, you narrow that range down to the level of your confidence.

And for both of these operations, brutal honesty is the only option.

Lessons from Me

To be honest about these things, you have to have some idea of how *wrong* you might be. So let me tell you two stories about how wrong I once was.

Story #1: Vectors

The year was 1978. I was working at a company named Teradyne, in Deerfield, Illinois. We built automated test equipment for the telephone

company.

I was a young programmer, 26 years old. And I was working on the firmware for an embedded measurement device that bolted into racks in telephone central offices. This device was called a *COLT*—Central Office Line Tester.

The processor in the COLT was an Intel 8085—an early 8-bit microprocessor. We had 32K of solid-state RAM and another 32K of ROM. The ROM was based on the Intel 2708 chip, which stored 1K. So we used 32 of those chips.

Those chips were plugged into sockets on our memory boards. Each board could hold 12 chips; so we used three boards.

The software was written in 8085 assembler. The source code was held in a set of source files that were compiled as a single unit. The output of the compiler was a single binary file somewhat less than 32K in length.

We took that file and cut it up into thirty-two 1K chunks. Each 1K chunk was then burned onto one of the ROM chips, which were then inserted into the sockets on the ROM boards.

As you can imagine, you had to get the right chip into the right socket on the right board. So we were very careful to label them.

We sold hundreds of these devices. And they were installed in telephone central offices all over the country.

So what do you think happened when we changed that program? Just a one-line change?

If we added or removed a line, then all the addresses of all the subroutines after that point changed. And since those subroutines were called by other routines earlier in the code, every chip was affected. We had to reburn all 32 chips for a one-line change!

This was a nightmare. We had to burn hundreds of sets of chips and ship them to all the field service reps, all around the world. Then, those reps would have to drive hundreds of miles to get to all the central offices in

their district. They'd have to open our units, pull out all the memory boards, remove all 32 of the old chips, insert the 32 new chips, and reinsert the boards.

Now, I don't know if you know this, but the act of removing and inserting a chip into a socket is not entirely reliable. The little pins on the chips tend to bend and break in frustratingly silent ways. So the poor field service folks had to have lots of spares of each of the 32 chips; and suffer through the inevitable debugging by removing and reinserting chips until they could get a unit to work.

So my boss came to me one day and told me that we had to solve this problem by making each chip *independently deployable*. He didn't use those words, of course, but that was the intent. Each chip needed to be turned into an independently compilable and deployable unit. This would allow us to make changes to the program without forcing all 32 chips to be reburned. Indeed, in most cases, we could simply redeploy a single chip—the chip that was changed.

I won't bore you with the details of the implementation. Suffice it to say that it involved vector tables, indirect calls, and the partitioning of the program into independent chunks of less than 1K each.[2]

---

2. In other words, each chip was turned into a polymorphic object.

My boss and I talked through the strategy, and then he asked me how long it would take me to get this done.

I told him two weeks.

But I wasn't done in two weeks. I wasn't done in four weeks. Nor was I done in six, eight, or ten weeks.

The job took me 12 weeks to complete. It was a lot more complicated than I thought.

So I was off by a factor of six. Six!

Fortunately, my boss didn't get mad. He saw me working on it every day. He got regular status updates from me. He understood the complexities I was dealing with.

But still. Six? How could I have been so wrong?

Story #2: pCCU

Then there was that time, in the early '80s, that I had to work a miracle. You see, we had promised a new product to our customer. It was called *CCU-CMU*.

Copper is a precious metal. It's rare and expensive. The phone company decided to harvest the huge network of copper wires that it had installed all over the country during the last century. They replaced it with a much cheaper, high-bandwidth network of coaxial cable and fiber carrying digital signals. This was known as *digital switching*.

The CCU-CMU was a complete rearchitecture of our measurement technology that fit within the new digital switching architecture of the phone company.

Now, we had promised the CCU-CMU to the phone company a couple of years before. We knew it was going to take us one person-year or so to build the software. But then, we just never quite got around to building it.

You know how it goes. The phone company delayed their deployment, so we delayed our development. There were always lots of other, more urgent issues to deal with.

So one day, my boss calls me into his office and says that they had forgotten about one small customer who had already installed an early digital switch. That customer was now expecting a CCU/CMU within the next month—as promised.

So now I had to create a person-year of software in less than a month.

So I told my boss that this was impossible. There was no way that I could get a fully functioning CCU/CMU built in one month.

Then, he looked at me with a sneaky grin and said that there was a way to cheat.

You see, this was a very small customer. Their installation was literally the smallest possible configuration for a digital switch. What's more, the configuration of their equipment just happened—just happened—to eliminate virtually all of the complexity that the CCU/CMU solved.

Long story short: I got a special-purpose, one-of-a-kind unit up and running for the customer in two weeks. We called it the *pCCU*.

The Lesson

Those two stories are examples of the huge range that estimates can have. On the one hand, I underestimated the vectoring of the chips by a factor of six. On the other, we found a solution to the CCU/CMU in one twentieth the expected time.

This is where honesty comes in. Because, honestly, when things go wrong, they can go very, very wrong. And when things go right, they can sometimes go very, very right.

And this makes estimating one hell of a challenge.

Accuracy and Precision

It should be clear by now that an estimate for a project *cannot* be a date. A single date is far too precise for a process that can be off by as much as a factor of six, or even twenty.

That variation means that every estimate is likely wrong. That's why we call them *estimates*. But even though an estimate is wrong, it may not be very wrong. So part of the job of estimation is to estimate how wrong the estimate probably is.

A single date is too precise to be honest, so estimates are ranges; they must be *probability distributions*. Probability distributions have a mean, and a width—sometimes called the *standard deviation* or the *sigma*. We need to be able to express our estimates with both a mean and a sigma.

My favorite technique for estimating the probability distribution of an estimate is to estimate three numbers: the best case, the worst case, and the normal case.

The normal case is how long you think the task would take you if the average number of things go wrong—if things go the way they *usually* do. Think of it as the date that you've got a 50% chance of making.

The worst-case estimate is the Murphy's law estimate. It assumes that anything that can go wrong will go wrong. It is deeply pessimistic. Think of it as the date that you've got a 95% chance of making.

The best-case estimate is when everything goes right. You eat the right breakfast cereal every morning. When you get into work, your coworkers are all polite and friendly. There are no disasters in the field, no meetings, no telephone calls, no distractions.

Your chances of hitting the best-case estimate are 5%: one in twenty.

OK, so now we have three numbers. The best case which has a 5% chance of success. The nominal case which has a 50% chance of success. And the worst case, which has a 95% chance of success, represents a normal curve —a probability distribution. It is this probability distribution that is your actual estimate.

Notice that this is not a date. We don't know the date. We don't know when we are really going to be done. All we really have is a crude idea of the probabilities.

Without certain knowledge, probabilities are the only logical way to estimate.

If your estimate is a date, you are making a commitment, not an estimate. And if you make a commitment, you *must* succeed.

Sometimes you have to make commitments. But the thing about commitments is that you absolutely must succeed. You must never promise to make a date that you aren't sure you can make. To do so would be deeply dishonest.

So, if you don't know—and I mean *know*—that you can make a certain date, then you don't offer that date as an estimate. You offer a range of dates instead. Offering a range of dates with probabilities is much more honest.

Aggregation

OK, so now let's say that you've got a whole project full of tasks that have been described in terms of Best- (B), Normal- (N), and Worst- (W) case estimates. How do you aggregate them all into a single estimate for the whole project?

You simply represent the probability of each task, and then accumulate those probabilities using standard statistical methods.

The first thing we want to do is represent each task in terms of its expected completion time and the standard deviation.

Now, remember six standard deviations (three on each side of the mean) corresponds to a probability of better than 99%. So we're going to set our standard deviation, our sigma, to Worse minus Best over 6.

The expected completion time (mu) is a bit trickier. Notice that N is probably not equal to the midpoint between W and B. Indeed, the midpoint is probably well past N. This is because it is much more likely for a project to go long than to go short. So it's probably best to use a weighted average, like this: mu = (B + 4N + W) ÷ 6.

Now you have calculated the mu and sigma for a set of tasks. So the expected completion time for the whole project is just the sum of all the mus. The sigma for the project is the square root of the sum of the squares of all the sigmas.

This is just basic statistical mathematics.

What I've just described is the estimation procedure invented back in the late 1950s to manage the Polaris Fleet Ballistic Missile program. It has been used successfully on many thousands of projects since.

It is called *PERT*—Program Evaluation and Review Technique.

Honesty

So we started with honesty. Then, we talked about accuracy and precision. Now it's time to come back to honesty.

The kind of estimating we are talking about here is intrinsically honest. It is a way of communicating, to those who need to know, the level of your uncertainty.

This is honest because you truly are uncertain. And those with the responsibility to manage the project must be aware of the risks they are taking so that they can manage those risks.

But uncertainty is something that people don't like. Your customers and managers will almost certainly press you to be more certain.

The only way to truly increase certainty is to do parts of the project. You can only get perfect certainty if you do the entire project.

So part of what you have to tell your customers and managers is the cost of increasing certainty.

Sometimes, however, your superiors may ask you to increase certainty using a different tactic. They may ask you to commit.

You need to recognize this for what it is. They are trying to manage their risk, by putting it onto you. By asking you to commit, they are asking you to take on the risk that it is their job to manage.

Now, there's nothing wrong with this. Managers have a perfect right to do this. And there are many situations in which you should comply. But—and I stress this—only if you are reasonably certain you *can* comply.

If your boss comes to you and asks if you can get something done by Friday, you should think very hard about whether that is reasonable. And if it is reasonable and probable, then by all means, say yes!

But under no circumstances should you say yes if you are not sure.

If you are not sure, then you *must* say no, and then describe your uncertainty as we've described. It is perfectly OK to say: "I can't promise Friday. It might take as long as the following Wednesday."

In fact, it's absolutely critical that you say no to commitments that you are not sure of. Because if you say yes, you set up a long domino chain of failures for you, your boss, and many others. They'll be counting on you, and you'll let them down.

So, when you are asked to commit, and you are sure you can comply, then say yes. But if you aren't sure, then say *no*, and describe your uncertainty.

And be willing to discuss options and workarounds. Be willing to hunt for ways to say yes. Never be eager to say *no*. But also, never be afraid to say no.

You see, you were hired for your ability to say no. Anybody can say yes. But only people with skill and knowledge know when and how to say no.

One of the prime values you bring to the organization is your ability to know when the answer must be no. By saying no at those times, you will save your company untold grief and money.

Pressure

One last thing: Often a manager will try to cajole you into committing—into saying yes. Watch out for this.

They might tell you that you aren't being a team player, or that other people have more commitment than you do. Don't be fooled by those games.

Be willing to work with them to find solutions, but don't let them bully you into saying yes when you know you shouldn't.

And be very careful with the word *try*. Your boss might say something reasonable, like, "Well, will you at least try?"

The answer to this is:

NO! I am already trying. How dare you suggest that I am not? I am trying as hard as I can; and there is no way I can try harder. There are no magic beans in my pocket with which I can work miracles.

You might not want to use those exact words, but that's exactly what you should be thinking.

And remember this: If you say "Yes, I'll try," then you are lying. Because you have no idea how you are going to succeed. You don't have any plan to change your behavior. You said yes just to get rid of them. And that is the most dishonest thing of all.