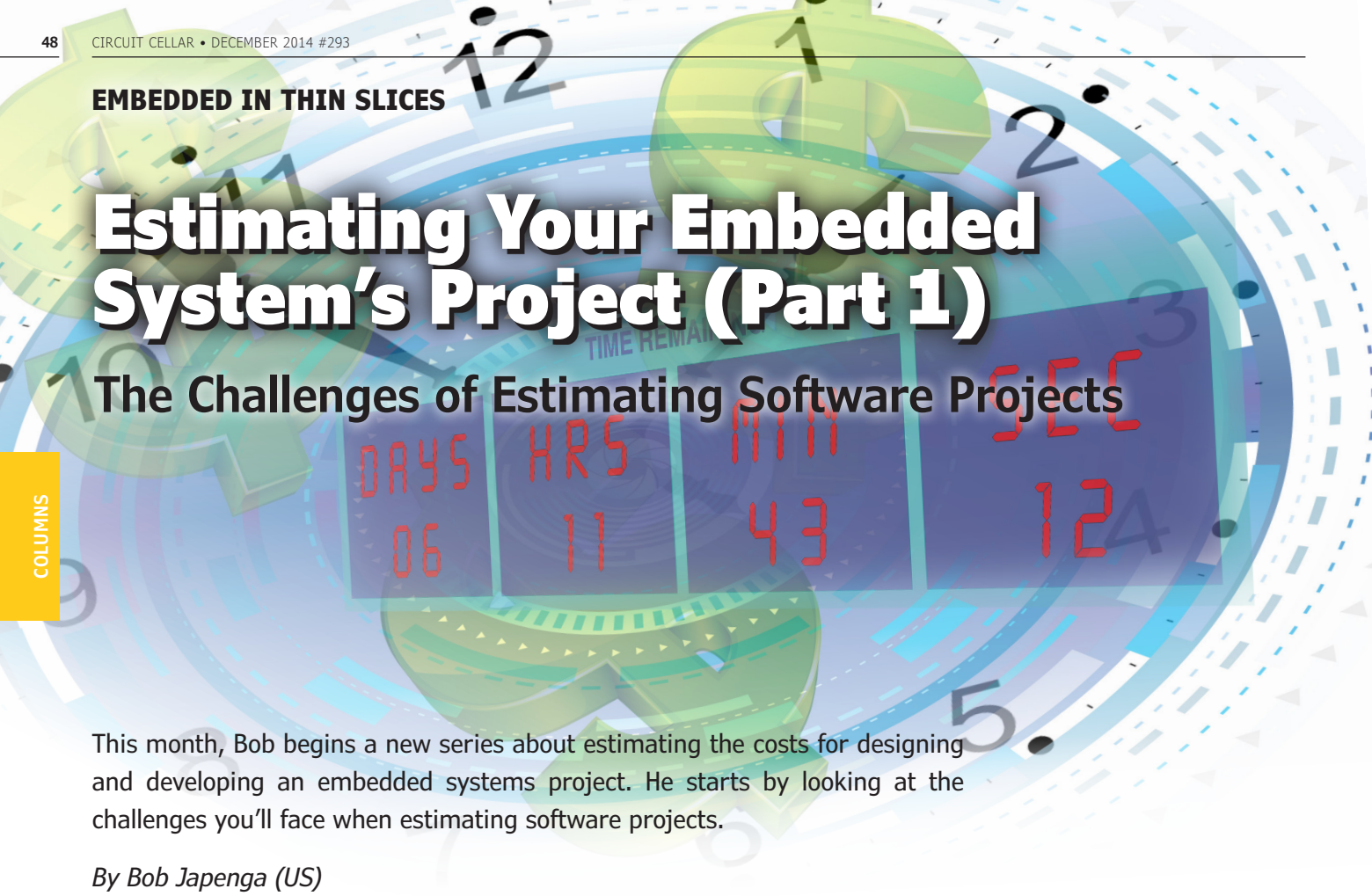
EMBEDDED IN THIN SLICES

Estimating Your Embedded System's Project (Part 1)

The Challenges of Estimating Software Projects



This month, Bob begins a new series about estimating the costs for designing and developing an embedded systems project. He starts by looking at the challenges you'll face when estimating software projects.

By Bob Japenga (US)

This morning at our 8:30 AM staff meeting, I asked one of the project leads if we were still planning on shipping a software update for an embedded system by 11:00 AM. In essence he reported that barring the sky falling or the final test failing, it should be released. The release was shipped at 11:14 AM. His response reflected the challenge we all face in making estimates for completing tasks we are assigned.

Accurately estimating my arrival time for dinner can be challenging. Accurately estimating embedded software development is almost impossible. Don't let anyone fool you into thinking that this is achievable in the real world. But we are not without hope. This article begins a three-part series on estimating embedded software systems. This month we will look at some of the general problems associated with estimating software development. In the next article, we will look at specific problems unique to embedded software development. We will close out the series discussing what we can do to continuously improve our estimating skills.

Learning how to estimate the time and dollar cost of designing an embedded system is very important to our business. We make our living by designing and building embedded systems for other people. We do this as fixed-priced projects. We live and die by our estimates. Perhaps we are like one of Nassim Taleb's black swans and the fact that we are still in business is just a statistical anomaly. But I think we have learned enough about estimating that, as imperfect as it is, we are able to estimate accurately enough to stay in business. Learning to estimate the time it takes to design embedded systems may not be that critical to you staying in business, but it is an important skill even if you work for someone else. Let's dive in by trying to identify some of the problems.

THE GENERAL PROBLEM

Asking you to estimate software development time is like asking you to estimate the number of pens I have in my desk. You could probably bound the problem on the lower side. "Well it is not less than zero." You could probably put an upper bound on it based on the estimated size of the average desk drawer and the size of an average pen. You could probably assign some reasonableness factors to it. "Bob would not have 1,000 pens in his desk." But how accurate can you be with so little information?

In software, without actually doing the design, you are working with even less data when you estimate. Software systems are the most complex things we design on the planet.

And the complexity increases exponentially as the systems get bigger. So our first axiom is that estimating the cost to develop software is extremely difficult.

As engineers we are well aware of an old axiom when applied to electronic systems design: "*You cannot control what you don't measure.*"

If you are asked to control the temperature of a room but cannot measure the temperature you will fail. Tom DeMarco, in his book *Controlling Software Projects*, applied this well-worn chestnut to software estimating. It was 1983. He introduced me to the concept of software metrics. Unless I was measuring what it actually took to design and develop software, I would not be able to estimate what it would take to design and develop my next project. This was a pivotal step in my professional growth. Two of the major points of his book were that we poorly estimate software development time because: we don't develop estimating expertise and we don't base our estimates on past performance. I set out to correct that. I wanted to develop expertise in estimating. I wanted to understand past performance.

With DeMarco's principles as my mentor, over the next seven years, my organization generated all kinds of metrics (some of which we will touch on in the last article) and used them in estimating software development costs. But there was a problem. After all that, I wasn't getting any better at estimating even though I had a lot of data. I developed my own corollary to DeMarco's axiom: "*And even if you do measure it, it doesn't mean you can control it.*"

I needed a new mentor. I found one in W. Edwards Deming, who arguably was the father of statistical quality control who is credited with single handedly bringing Japan's economy out of its World War II disaster to become the third largest economy in the world. Concerning quality control, he taught that: *"The most important things cannot be measured."*

I began to perform post-mortems on projects (a highly desirable practice for anyone who wants to continue to improve their software development process—including estimating). I tried to understand where the estimate went wrong. I had lots of examples. What I found was that Deming was correct and that the things that were sinking the schedules and making the estimates look bad were, for the most part were things I could not measure.

Deming was also famous for saying that in quality control: *"The most important things are unknown or unknowable."*

Let's look at just a few of the unmeasurables, unknowns, and unknowables that caused my estimates to go wildly wrong.

NOT CLARIFYING REQUIREMENTS

In 1993, we started the largest software project in our history at that time. There was one little line in the statement of work that said: "Provide reports for the data." This was an embedded system running on an Intel 80188. It had 512 KB of EEPROM. How many reports can this be? We estimated that it would take us 80 hours to create these reports. It took us over 200 hours to generate the 100 different reports that the customer had in mind. Here was a requirement that was knowable, but we didn't clarify it. That made it unknown to us.

MISESTIMATING MEMORY REQUIREMENTS

If you have to start playing games to shoe-horn your code into the memory allowed, you can blow your schedule completely out of the water. In 1974 a professor in one of my classes said: *"In the very near future, you will never have to worry about memory constraints again."*

Now this professor was smart. But he was dead wrong. Hardly a project goes by my desk that still doesn't concern itself with either RAM or ROM memory usage. In 1997 we were porting an embedded system to a DOS PC. Because we misestimated the memory requirements, we were forced to convert the program to use overlays (basically keeping only some of the program in memory at a time—something that happens automatically now). That part was easy. The problem was that the overlays didn't work in a multitasking OS that we put on top of DOS. It took two engineers a month working close to 80 hours per week to solve. The memory required for a software project is an unknowable unless you have designed it or something similar before.

THAT ELUSIVE BUG

I have more stories than there are pages in this magazine about one tenacious bug that took almost as long to find and fix as the entire estimate. This happens a lot. In 2006, we were about to release an embedded system on time and within budget estimates. Then during the last weeks of testing, a problem surfaced in the Linux C library that was difficult to duplicate and even more difficult to fix. We spent more than a calendar month and several man-months to correct this bug. How could we have estimated for this unknown?

THE OTHER GUY

There is a class of estimate defeaters that

ABOUT THE AUTHOR

Bob Japenga has been designing embedded systems since 1973. In 1988, along with his best friend, he started MicroTools, which specializes in creating a variety of real-time embedded systems. With a combined embedded systems experience base of more than 200 years, they love to tackle impossible problems together. Bob has been awarded 11 patents in many areas of embedded systems and motion control. You can reach him at rjapenga@microtoolsinc.com.

I call "The Other Guy." The other guy's API doesn't work the way we expected. The other guy's web server doesn't respond quickly enough. The other guy's device doesn't meet specifications.

On a 2010 project, we were using a flow sensor to measure oxygen content and flow. This sensor worked in most cases but sometimes didn't. Some sensors worked flawlessly. Some did not. This was unknowable to us when we performed the estimate for the proposal. We spent many times our original estimate in getting it to work.

RECOMMENDED READING

I'd like to recommend the following books to anyone interested in becoming a better software engineer:

- "Better Sure Than Safe? Over-Confidence in Judgment-Based Software Development Effort Prediction Intervals" (*Journal of Systems and Software*, Volume 70, February 2004): This article by Magne Jørgensen, Karl Halvor Teigen, and Kjetil Moløkken provides research into how our over-confidence affects our ability to estimate.
- *Controlling Software Projects* (Prentice Hall, 1986): I cut my teeth on Tom DeMarco's book. It is still useful today.
- *Dr. Deming: The American Who Taught the Japanese About Quality* (Millennia Management Associates, 2010), by Rafael Aguayo and W. Edwards Deming
- *Smart and Gets Things Done: Joel Spolsky's Concise Guide to Finding the Best Technical Talent* (Apress, 2007), by Joel Spolsky
- *The Black Swan: The Impact of the Highly Improbable* (Random House Publishing Group, 2010): Nassim Taleb's book is essential reading for anyone who wants to be able to predict the costs of developing embedded systems
- *The Mythical Man-Month* (Addison-Wesley Professional, 1975): Frederick P. Brooks, Jr.'s classic book is a must read for anyone who wants to grow in their ability to estimate software projects.
- *The Soul of the New Machine* (Atlantic-Little Brown, 1981): Tracy Kidder's classic book, although outdated in its context, teaches a lot about developing electronic systems. It describes how a team of software and hardware engineers at Data General developed the Eclipse in 1980.
- *What the Dog Saw* (Bay Back Books, 2010): Malcolm Gladwell's book introduced me to the concept of "creeping determinism," which is better called "hindsight bias" (http://en.wikipedia.org/wiki/Hindsight_bias).

HIDDEN COMPLEXITY

Very often when an estimate is made, there are lurking under the surface a vast array of hidden complexities that can only be uncovered by implementing and testing. Fred Brooks puts it this way in *The Mythical Man Month*, "the incompleteness and inconsistencies of our ideas become clear only during implementation."

This week I was reviewing the specification for an interface to a new cell modem module we were including in one of our designs. All of our designs work on very low cost data plans. This requires us to use very little data bandwidth every month. One of the cell carriers has a very strict and very clear set of requirements about handling retries. Retries can be very costly in data plans. My specification called out for the software to meet these very strict and very clear requirements. It should be easy to estimate how long it will take to implement that algorithm. But during the review, one of the designers asked: "How do we manage the retries handled by the TCP/IP stack underneath us?"

We all knew that the TCP/IP logic in our embedded Linux systems performs retries. How will we make that work with the carrier's retry requirements? This is an unknown with lots of hidden complexity and could easily add an order of magnitude to the effort required to implement this. In this case, we happened to think about this complexity but that doesn't always happen.

PROGRAMMER EFFICIENCY

We all know that different people take different amounts of time to do the same job. Joel Spolsky in his book *Smart and Gets Things Done* claims that his research shows that there can be more than an order of magnitude difference is programmer efficiency.

OK, so you take that into account when you estimate a job based on the programmers you have. But a programmer leaves or gets sick. You have to use your least efficient programmer. Now you could easily see your estimate go out the window by an order of magnitude. The most important things are unknowable and unmeasurable.

OPTIMISM AND HUBRIS

These nonidentical twins are one of the reasons we fail to accurately estimate our software projects. We always think we can get better. Because of our pride we can easily fall prey to wishful thinking. I know I can do it better this time. Research has demonstrated that this is a pervasive problem in software estimating. Related to this is what is called "hindsight bias." When combined with

optimism and hubris, we just think that we did better in the past than we actually performed.

STEP 1: RECOGNITION

Recognition of the problem is the first step at correcting it. We have named a few of the trees in the forest of problems with estimating software development costs. Tom DeMarco noted: *"An estimate is the most optimistic prediction that has a nonzero probability of coming true."*

The problem for us is that the probability of most of our estimates of coming true is very close to zero. We have to learn how to do it better. Next time we will again focus on identifying the problem—but this time specifically from an embedded design perspective. Of course, only in thin slices.