acmqueue

The Hyperdimensional Tar Pit

Make a guess, double the number, and then move to the next larger unit of time.

Poul-Henning Kamp

When I started in computing more than a quarter of a century ago, a kind elder colleague gave me a rule of thumb for estimating when I would have finished a task properly: make a guess, double the number, and then move to the next larger unit of time.

This rule scales tasks in a very interesting way: a one-minute task explodes by a factor of 120 to take two hours. A one-hour job explodes by “only” a factor 48 to take two days, while a one-day job grows by a factor of 14 to take two weeks.

The sweet spot is a one-week task, which becomes only eight times longer, but then it gets worse again: a one-month job takes 24 times longer when it is finished two years from now.

There is little agreement about what unit of time should follow *year*. Decade, lifetime, and century can all be defensibly argued, but the rule translates them all to *forever* in practice.

Intuitively, the rule makes sense, in that we tend to overlook details and implications for small tasks, and real life—in the shape of vacations, children, and random acts of management—frustrates the completion of the big tasks. The usability problem is that it is a one-person rule that talks about duration and not about effort.

We do have another old rule to help us: in the very first figure in chapter one of *The Mythical Man-Month*, Frederick P. Brooks Jr. makes what I think is the most important point in his classic masterpiece. The figure illustrates that delivering “a programming product” is three times more effort than just making “a program,” and that making “a programming system” is also three times the work of “a program,” but that delivering “a programming systems product” is nine times—almost a magnitude—more work than just writing “a program.”

The terminology is a bit dated—not unexpected from a 1975 book, which is probably unread by many younger readers—so let me spend a moment translating it to modern terms.

A program is a program is a program. We are talking about the code you write to do something. It will only ever have one user, you, and will probably never be run again, certainly not next month or next year. This program is our yardstick for the following discussion.

To turn your program into a product, you must make it possible for other people to use it without your presence or help. You need to document it, add error checking to input values, and make sure the algorithmic assumptions cover the intended use and warn the user when they don’t.

What Brooks calls “a programming system” is something we use to program with—class libraries, programming languages, debuggers, profilers, operating systems, etc. Here the extra effort will be spent on generalizing the domain of applicability, handling corner-cases sensibly, and generally implementing the Principle of Least Astonishment throughout.

If you want to make your class library or programming language usable for strangers, then you get to do all of the “productization,” which, as Brooks points out, does not add to but instead multiplies the necessary effort.

But Brooks was lucky.

Back in 1975, life was a lot simpler. Men (and they were mostly men at the time) were real programmers, computers stayed put where the forklift put them, “standards compliance” was about the width of your tapes (whether paper or magnetic), and “internationalization” was about how well your customers wrote and (if you were unlucky) spoke English.

I worked in a company where the whiteboard read, “Internationalization er et problem vi har mostly styr på,” with the Danish and English words written in black and green, respectively. A wry commentary on the difficulty of a problem so intractable is that we have even given up on its proper name (internationalization) and, instead, simply record the inordinate amount of letters therein: i18n.

To Brooks’s two Cartesian coordinates we must add internationalization as the third, and while we are at it, make the jump into hyperspace by adding a dimension for standards compliance, as well. Complying with standards means that in addition to your own ideas and conceptual models of the subject matter, you must be able to cope with whatever conceptual models were imagined by the people who wrote the standard, while having something entirely different in mind.

Tracy Kidder relates an example in his book, *The Soul of A New Machine*. You may think you build computers, but you ignore the relevant standards for European freight elevators at your peril.

Before anybody gets carried away, let me make it clear that security is not the next dimension in this software geometry. Security is neither a choice nor an optional feature. Lack of security is just an instance of lack of quality in general.

What makes these four dimensions different from other attributes of software is that like pregnancy, they are binary attributes. A software property such as quality is a continuous variable. You can decide how much quality you want and see how much you can afford, but making your program a product or not is a binary decision. There is no way to make it a little bit of a product.

Not that the world isn’t littered with products lacking documentation, libraries doing 37 percent of what’s needed, internationalization of all but the “most tricky dialog boxes,” and code that complies with only the “easy” or superficial parts of standards. There is plenty of such software—I have written some of it and so have you. But those shortcuts and shortcomings are invariably perceived as lack of quality, not as fractional dimensions. We don’t think, “21 percent of product”; we think, “nowhere near done.”

Once we embrace this way of thinking about software, we can put a price tag on marketing-inspired ideas such as this thinly disguised example from the real world: “Make it do XML and we will make a fortune *selling it* as a *module* to all the social media sites in the *world*.” If the program took a month to write and Brooks’s 1975 estimate of a factor of 3 still holds, I personally think of it only as a lower bound. We can confidently say that is not going to happen in less than:

- “XML” = standardization; now it’s 3 months.
- “selling it” = product; now it’s 9 months.
- “module” = programming; up to 27 months.
- “world” = internationalization; make it 81 months.

Less the one month already spent: 80 months of extra effort.

Our rule of thumb tells us that if we expect one programmer to do it, he or she will never get it done: 160 years.

To Frederick P. Brooks Jr.: Thanks for the best computer book ever.

To everybody else: Now go read it (again!).

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

POUL-HENNING KAMP (phk@FreeBSD.org) has programmed computers for 26 years and is the inspiration behind bikeshed.org. His software has been widely adopted as “under the hood” building blocks in both open source and commercial products. His most recent project is the Varnish HTTP accelerator, which is used to speed up large Web sites such as Facebook.

© 2012 ACM 1542-7730/12/0100 \$10.00