Phillip Armour

# Ten Unmyths of Project Estimation

## Reconsidering some commonly accepted project management practices.

According to the late Joseph Campbell a myth is not, as we may think, untrue. A myth is something that is extremely true. It is the essence of truth dressed up in an allegorical costume that helps us remember its lesson. If a myth is truth packaged in a fabrication, then something that appears perfectly reasonable but is, in essence, wrong would be an "unmyth." There are a few "unmyths" in project management and estimation. There are truths that, when we look closely at them are not, well, *true*.

When we estimate a project, we attempt to foretell the future. Historically, this has been a dangerous occupation. Oracles and soothsayers have been praised and lauded in the past. But they have also been stoned to death too—if they proved to be wrong and sometimes if they were right. And so it is with predicting the time and effort for software projects. Perhaps project estimation should come with a warning: "The Surgeon General has determined that project estimation can be harmful to your career." Anyway, as Woody Allen once remarked: the only thing I cannot accurately predict is the future.

There are several verities of estimation that are not very "correct." They may have elements of rightness, a veneer of truth, but they may also be worth a critical look. It might help understand how and why we can sometimes be so wrong in estimating, and maybe what we can do about it.

***The Accuracy Unmyth: We can have an "accurate estimate."*** Apart from being an oxymoron, there is a very simple reason why estimates cannot be "accurate"—we simply *do not have the data* necessary to be accurate. It is a sad fact that the earlier the estimate is made, the less data we have available, and therefore the less "accurate" we can be. The only time we have sufficient data to truly warrant the label "accurate" is at the very end of the project when all the variables are resolved. Unfortunately, no one will ever ask for an estimate at that stage. We can, however, have a "lucky" estimate. This is when all the things we didn't think of that would make the project run faster, and all the things we didn't think of that would make the project run slower, happen to be equal. This is luck, not accuracy, and since there are far more things that will slow down a project than speed it up (an application of the Second Law of

LISA HANEY

## There are several verities of estimation that are not very "correct." They may have elements of rightness, a veneer of truth, but they may also be worth a critical look.

Thermodynamics to projects), we almost invariably underestimate.

***The End-Date Unmyth: The job of estimating is to come up with a date for completion.*** This can be easily demonstrated using the recently revived SDI program (aka "Star Wars"). What is the likelihood of completing this program by, say, next Wednesday? Shooting from the hip, I would say almost zero. However, the likelihood of completing it by 100 years next Wednesday, we could reasonably assert, is close to 100%. If the probability starts off at 0% and a century later reaches 100%, its value must move. How it moves depends on the type of project. Specifically, it depends on the uncertainty (or amount of unknown knowledge at the time of estimating, which is the same thing). We can pick any date we want from between next Wednesday to the year 2102. Each date comes with some probability of completion from infinitesimally small to high confidence of success. Since we can pick any date, estimation does not give us a date for completion—what it gives us is a *probability* for completion.

***The Commitment Unmyth: The estimate and the commitment are the same.*** The End-Date Unmyth leads to another common assumption that is very dangerous in practice. This is our failure to differentiate between the estimate and the commitment to the estimate. Estimating generates a probability, not a date; but we cannot commit to a probability, we can only commit to a date. The commitment *process* is what turns a probability into a date. Using knowledge of the resources available, business goals, ROI, and so forth, the commitment process picks a date that optimizes the probability of a good return on investment and manages the risk. It is in misunderstanding this that large commitments of lots of money are made by big companies based on hunches and guesses by junior programmers.

***The Size Unmyth: A project estimate is dependent on the size of the final system.*** This is actually kind of true—but only as a gross average. Since software development is really a knowledge acquisition activity, rather than a code production activity, the size of the product may actually be irrelevant. If I can reuse a whole new system, the size could be large, but the effort very small. Equally, I can have a tiny embedded real-time system that is extremely complex requiring a huge effort. This is tacitly acknowledged in the calibration of some of the more popular estimation tools and methods—real-time embedded systems (which are typically "smaller" but more complex) have "productivity factors" much lower than IT systems. The effort is, of course, dependent on the amount of knowledge that must be obtained. The final size of the system is dependent on the amount of knowledge that can be reused from elsewhere *plus* the amount of knowledge that must be gained. It is the "knowledge-to-be-gained" that determines the effort on the project.

***The History Unmyth: Historical data is an accurate indicator of productivity.*** This is also somewhat true, but we use it mostly because we don't have anything better. The problem with using history is that it was gathered on earlier and different projects. The project being estimated must be different from previous projects. If we've kept the knowledge from the previous projects, the "knowledge-to-be-gained" on this project is the difference. But it is in acquiring this knowledge that the true effort lies. The productivity calibration was done on a different set of knowledge, so almost by definition it probably doesn't quite apply. Luckily, especially on larger projects, our ability to acquire knowledge we do not already have tends to be similar no matter what that knowledge

might be. But this is only as an average and not true, of course, if the knowledge is really different, or the developers are different, or the customer is different, or …

***The Productivity Unmyth: Productivity is an accurate indicator of project duration.*** Most productivity measures imply a rate of production, which implies we are building a product, which is not what we really do. Since much of software development involves acquiring knowledge we do not already have, we can neither accurately estimate how long it will take us (see the End-Date Unmyth), or even if we've got it all (due to Second Order Ignorance (2OI)). A simple example: we might be wonderfully productive in creating software modules; we build, test, and ship the system only to find there is some environmental knowledge we missed. There is some key fact in the customer's setup, company, business model, or some other area we did not acquire. Oops. Back comes the system to be fixed (have the new knowledge added). Big delay. We were certainly productive in a lines-of-code sense, but the project still took a lot longer than it should have.

***The LOC Unmyth: A Line of Code (LOC) count is a good way to size a system.*** Our job is not to produce LOC, it is to acquire knowledge; if the lines of code do not contain the "correct" knowledge, it doesn't matter how many there are. Herein lies the rub—the size of the system for estimation purposes is related to the quantity of knowledge that must be obtained. It's also related to the difficulty of getting that knowledge, which is a story for another time. The trouble with sizing is we are trying to estimate a quantity of knowledge and the human race simply has not found a way to quantify knowledge. After thinking about this for a couple of thousand years, we still haven't come up with a "knowledge unit." LOC may average out, as averages do, but using it as a measure of knowledge quantity is pretty much like weighing a book to figure out how much knowledge it contains.

***The Function Point Unmyth: Function Points are a good way to size a system.*** If LOC are not a good measure of size, how about Function Points, in one of their numerous guises? Many of the same issues arise, of course. Function Points have some advantages and some disadvantages over LOC. Most Function Point counting methods are not directly measurable in the system in quite the same way that LOC counts are, though they are usually measurable in some representation of the system. On the plus side, the Function Point approach counts elements, such as inputs and outputs to a system, which we can intuitively assert are related to the size, and presumably the knowledge content, of the system. That means if we increase the number and complexity of inputs, we would proportionally increase the size of the system and the amount of knowledge it contains.

Unfortunately, there are many aspects of system complexity (knowledge content) that are not directly, if at all, related to the things Function Points count. For example: the complexity of a telecommunications system may be almost entirely unrelated to its inputs and outputs. A rich variety of Function Point formulae have sprung up to try to address these issues, but they suffer from the same problems that beset LOC—they too are not "knowledge units."

***The More People Unmyth: We can get the system faster, by assigning more resources.*** This is a lesson we must learn over and over, it seems. While some estimating tools and formulae show a huge dependency of time and resources on delivery date, adding people to projects to get them done faster is still the most common management recourse. Since software development is a knowledge acquisition business, there are many issues involved. The first is that there is a finite rate at which people can accumulate knowledge—it cannot be increased past that limit. The second is that, when knowledge is intricately connected, dismembering and allocating bits of it to different people does not allow the whole picture to be seen.

Unfortunate revelations occur in the integration testing phases of such projects. Also, more people equals a lot more potential

communication channels; we may staff a project with people who have a lot more to learn (that is, are less experienced); the pace of a project may cause us to assume we do know something, without actually validating that assumption—which reality will do for us, of course, when we ship the system.

***The Defect-Free Unmyth: Given enough time, we can create a defect-free system.*** Or enough experience, computing power, the correct process, and so on. Defect-free is almost a Zen-like goal. Even understanding that we cannot achieve it, it is the only pure quality goal. We cannot achieve it, for the same reason we cannot achieve full enlightenment—we don't know what it is. A defect is a lack of knowledge, it is a misunderstanding, it is something we did not know. Lack of knowledge becomes a defect when it is made manifest by the real world (or some real-world surrogate such as testing). That moment when the bug is detected I call *an epiphany of knowledge.* It is the moment when the real world tells us there's something here we don't know. The only way that we could truly have a defect-free system would be to run the system against all conceivable sets of variables. Not only is the set of all possible inputs very large, but we also have a paradox related to the word *conceivable*. We can only test and inspect for things we know, or at least know we don't know. We cannot test thoroughly for things we don't know we don't know (2OI). Unfortunately it is 2OI that is the truly valuable knowledge, and the real purpose of the project.

## Mythtreating the Unmyths

***Accuracy.*** Estimates don't have to be accurate—they have to be accurate enough to fit within our business degrees of freedom. That is, the estimation process must only come up with estimates that allow us to not make really bad decisions, from which we can't recover. Well, at least most of the time.

***End-Date.*** We can define estimate outputs in terms of a *range of probabilities* associated with dates (and other resource factors) to help this.

***Commitment.*** Establish processes with discrete estimate and commitment stages, with defined inputs and outputs.

***Size.*** There is no real fix to this, but keeping history of size ranges might help understand the likely variance. We really need to perform a (lack of) knowledge assessment, rather than a size calculation.

***History.*** This is also somewhat intractable, but having an estimating process that modifies history for the current situation would help. Many estimating tools do a good job of this.

***Productivity.*** Also a perennial problem. Using historical productivity as an indicator, rather than an explicit predictor helps.

***LOC.*** Invent a real knowledge unit that can be empirically measured, can be calibrated against the environment, and peoples' individual and collective understanding and that can also be used to evaluate evident, projected, or assumed lack of knowledge. (Best of luck, and please call me when you're done.)

***Function Points.*** Ditto, though using Function Points only where there is a strong relationship between the knowledge types and the function point units would help.

***More People.*** This can only work where there is well-defined separation between the knowledge in functions, supported by the product architecture, the development methodology, peoples' collective knowledge base and knowledge acquisition methods, and the project management and task assignment. Otherwise it just doesn't work.

***Defect-Free.*** Good enough quality? That's what it always is, even if we don't want to admit it (and probably shouldn't intentionally work to it). Knowledge in software has one function—to deliver value to the customer. To allow the customer to do things the customer could not do without the software; to allow the user access to valuable knowledge he or she would not otherwise have. When the value delivered exceeds the cost of use (including compensating for defects) by a sufficient amount, the software is valuable to the customer. This value proposition is the only real criterion.

In the end, an estimate is just an estimate, it is not exact. After all, the process is called estimation, not exactimation. C

PHILLIP ARMOUR (armour@corvusintl.com) is a vice president and senior consultant at Corvus International, Inc., Deer Park, IL.