How Solved Is the Cost Estimation Problem?

Negotiations on project costs and delivery dates between the developers of software and the managers who request it are often a lot like the negotiations between opposing factions in Bosnia, Northern Ireland, or the Middle East. Each side is only concerned with what it can win, even if the resulting loss to the other side may jeopardize the entire situation.

Larry Putnam and Ware Myers suggest an approach for establishing peace between software engineers and their customers. They argue that a few commonsense steps, coupled with historical data, can help every software team solve the cost-estimation problem and, as a consequence, establish an environment in which both parties win.

—Roger Pressman

“The SOFTWARE COST ESTIMATION problem is solved.” So spoke Tom DeMarco—an editorial board member and occasional guest editor for this magazine—at the European Software Control and Measurement Conference a couple of years ago. His view carries special weight partly because he wrote one of the early books on this subject, Controlling Software Projects, in 1982 (Yourdon Press).

Besides, we agree with him. We, too, have written many articles and books on the subject, dating back as far as 1978. We haven’t talked to DeMarco about cost estimation, so we don’t know exactly what he meant by that quote. We mean that “The problem of cost estimation has been solved...in principle.” In practice, the rather imperious human psyches that oversee software development don’t always get it right.

“Rather imperious?” we hear you scream. “We’re the people who march in those death marches that Ed Yourdon talks about. We’re the people who work 80 hours a week. We’re the people who sacrifice our families to cover the Earth with software. Imperious? Downtrodden is more like it!”

We know. You deserve credit—lots of it. You aren’t the problem, it’s those cartoon people, the ones with big bellies, big cigars, and big desks. Most of you still work on projects that are not very well thought through. Why? Because somewhere above the 80-hour-a-week people are those other folks who get you involved in efforts that veer between the technically impossible and the impossibly costly.

There are reasons why the cigar-wavers fall into these traps. The three main ones are:

- failure to establish that the project is feasible,
- failure to reduce the remaining risks of a feasible project to manageable proportions, and
- failure to plan the work in the construction phase and then control it according to the plan.

Establish Feasibility. Back in the early years of World War II the human race was coming off 200 years of remarkable accomplishment—the spinning jenny, the steam engine, the Bessemer converter, the telegraph, electric power, the telephone, the automobile, the airplane—the list goes on. Backing up these technological achievements was the science of Newton’s equations, Maxwell’s equations, Einstein’s relativity theory, and quantum theory: a body of mathematics that was unambiguous and provably correct. Then the atomic bomb capped it all. The people in charge and the public in general began to believe that science and technology could accomplish whatever someone could imagine—no matter how wonderful or terrible.

Since World War II, however, we have seen some imaginings that did not work. The atomic-powered plane of the 1950s was abandoned only after the US government spent billions of development dollars. The project was technically impossible. The US’s planned supersonic passenger airplane was likewise abandoned, while the European Concorde turned out to be technically feasible but too expensive for the marketplace. In 1983, the Strategic Defense Initiative, popularly known as Star Wars, embodied the US’s hope that it could deploy a spaceborne umbrella against nuclear missiles; it turned out to be in-
feasible. Currently, California has been mandating zero-emission electric-battery cars; unfortunately, they fail to meet market criteria, such as acceptable range on a single charge. Yet batteries have been researched for over 100 years, so the basic physics appears to be against this initiative.

To estimate the time and cost of next time, you must know and be able to repeat what you did last time.

All of which amply proves that not everything imaginable is feasible—not even in software, evanescent as it may appear to outsiders. On the contrary, software feasibility has four solid dimensions.

♦ Technology. Is a project technically feasible? Is it within the state of the art? Can defects be reduced to a level matching the application’s needs?

♦ Finance. Is it financially feasible? Can development be completed at a cost the software organization, its client, or the market can afford?

♦ Time. Will the project’s time-to-market beat the competition?

♦ Resource. Does the organization have the resources needed to succeed?

For some projects in established areas the answers are easy. You have done projects like this one before. After a few hours’ or sometimes a few weeks’ investigation, you are sure you can do it again.

Projects on the margins of your experience are not so easy. A team may have to spend several months discovering what the central, difficult-to-implement requirements of a new application actually are. Do some of these requirements pose risks that would make the project infeasible? Can these risks be overcome? The feasibility team should work on initial architecture and high-risk requirements design until they can answer these questions. In some cases, when they get negative answers, the team may be able to negotiate a require-

ment reduction.

Meantime, the cartoon people drum their fingers nervously on their large desks. Often, they wave their fat cigars in a lordly manner and yell impatiently through the smoke screen, “Enough. Do it!”

Many of the projects that appear in the newspapers a few years later as whopping failures got their start this way.

REDUCE RISKS. The feasibility stage establishes only that the proposed system could be done. It does not determine in any detail the four planning factors on which a software organization bases its construction plans:

♦ How much function is needed?
♦ How long will it take?
♦ How much will it cost?
♦ What level of quality will result?

If the organization has developed a similar system before, it may understand what it has to do. If the system involves some areas outside the organization’s experience, they almost certainly present risks. A project group must explore the new areas, assessing and reducing each risk so that it can, finally, estimate the construction work involved.

Now, in the course of establishing feasibility and reducing risks, the project team, still small at this point, has given a lot of thought to what the client actually needs. The early risk-assessment prototypes will have shown the clients more about what they really need. In short, the requirements will have been changing during these first two stages and should, as a result, be quite firm.

Software development rests on factors that are quite obvious, but which people often overlook anyway:

♦ Requirements: you must know what you are going to do.
♦ Process: you must know how you are going to do it.
♦ Peopleware: you must have people who can do it.

PLAN AND CONTROL. At the conclusion of the risk-reduction stage—better known as functional or high-level design—we know what tasks remain to complete the project. Knowing this, we could estimate the project’s development time and cost, its main build. We say “could” because many software organizations say they “can’t.”

One of the reasons they “can’t” make such an estimate is that they don’t have a repeatable process. The Software Engineering Institute, while assessing software organizations’ capability maturity level, has found that more than two thirds of those organizations were stuck at the first of five levels. One feature of Level 1 is, not surprisingly, lack of a repeatable process. To estimate the time and cost of next time, you must know and be able to repeat what you did last time.

Another reason organizations “can’t” estimate accurately is that they don’t have any measurements from previous projects. Metrics from your past projects are vital for planning and controlling your current one. A software organization needs at least five such metrics. More are often useful, but the following are crucial for use in planning and controlling future projects.

♦ Amount of function built or modified. This is usually measured in lines of source code, but often in function points, modules, subsystems, use cases, or other indicators of work done.
♦ Development time. This measures the number of months or years in the main build stage.
♦ Effort. This measures, in number of person-months, the effort applied. It’s roughly equivalent to cost:

Cost = person-months × labor rate

♦ Process productivity. This indicates the rate at which work is accomplished.
♦ Defect rate. This measures quality and reliability.

An organization can count four of these five metrics on completed projects. It can derive process productivity from the first three. It can store these numbers in an online database where managers and estimators can find them when the time comes to consider another project. With these five numbers, organizations can project the corresponding numbers for the next project and, with those projected numbers, the organization can bid the next project.

If it wins the project, the organization can use these numbers to control the project’s execution. That is, it can project the effort applied per month in staffing over the duration of the project. It can project the defect rate during the main build period. It can estimate the number of lines of code to
be produced each month. When the project is complete, managers can compare the actual numbers against those projected. Deviations of actuals from plan call for investigation and corrective action. This is actually a version of the statistical control made famous by W. Edwards Deming.

**QUANTIFY UNCERTAINTY.** The final reason that project teams say they can’t estimate is that all these numbers are uncertain. Source lines of code, for instance, is an inexact indicator of a product’s functionality. This uncertainty leads many software people to throw up their hands, look skyward, and bleat, “We just do the best we can down here in the boiler room.”

Statistics was invented to cope with uncertainty. As a software developer, you have enough on your plate without having to master statistics, but the basics are not that difficult.

If the counts of source lines of code, person-hours, and months’ duration are a little uncertain, then the process productivity, derived from these three, is going to fall within a range, not on an exact point. If the estimate of the next project’s size, even if you wait until the end of the risk-reduction stage, is imprecise, then the estimate for the next project is going to be a range, not an exact number of dollars. Of course, the people with cigars want a quote to the exact dollar. Not to worry.

This is where statistics comes in. It can tell you something like this. There is a 50-percent probability that you can complete this project successfully in $x$ months with $y$ person-months of effort applied. Bidding on this basis, over the next dozen or so projects, your organization ought to break even. Some would lose money. Others would make money. You can raise that 50-percent probability to 75 percent, or even higher, up into the 90s—but you cannot realistically get to 100 percent. If you must bid low sometime, you can lower your estimate. If you do that, you will probably lose money on that particular job, but then you expected to. In the longer range, you can submit most of your bids in the 60 to 70 percent range and confidently expect to make some money over the next dozen projects.

“All that’s a very tall order,” we hear you say as you peer anxiously through the cigar smoke.

“Sure it is,” we reply. “Still, a few companies make it work and get good commercial results. Do you want them to get all the business?”

Lawrence H. Putnam is the founder of Quantitative Software Management. He has been involved with software estimating and control and benchmarking measurement for the last 25 years. He can be reached at (703) 790-0055; QSMINC@compuserve.com.

Ware Myers is a contributing editor to several IEEE Computer Society publications. He has many years’ experience as an engineer and technical editor. He can be reached at (909) 621-7082; 73152.1762@compuserve.com.

---

**bookshelf**

Continued from page 104

**Class Act in Parallel Programming**

_Nan C. Schaller, Rochester Institute of Technology_


**READER SERVICE 72**

“A WEEK OF CODING CAN SOMETIMES SAVE AN HOUR OF THOUGHT.” This quote from appendix A of Greg Wilson’s new book is typical of the wry wit with which he approaches his subject. Well written and entertaining, the book covers the basics of practical parallel computing: terminology, architecture, history, and metrics. It also addresses four parallel programming techniques: data parallelism, shared variables, message passing, and generative communication. While exploring each topic, you are constantly reminded of performance considerations—indeed, one of Wilson’s stated goals is to explain how to develop parallel software and achieve good computational performance on today’s parallel hardware.

**GOLDEN THROUGHOUT.** Not only is the book’s main text packed with useful information, the appendices are a gold mine as well. They include a useful, annotated reading list, a glossary of parallel computing terminology, “a little bit of sarcasm,” an extensive reference list, information about the Fortran-K Programming Language, and a short history lesson (for a more detailed history see Past, Present and Parallel, Greg Wilson and Arthur Trew, eds., Springer-Verlag, 1991).

Fortran-K is Wilson’s own emulator (currently available through FTP from mitpress.mit.edu—sign in as “guest” with your e-mail address as the password), designed so that you don’t need access to a parallel machine to explore the concepts presented. Wilson gives good reasons for why he chose to write his examples using his own emulator rather than one or more “real” parallel programming systems: Besides providing general accessibility, his examples show how to minimize both the debugging effort and the problems that arise when moving from system to system.

**COURSE-WORTHY.** Practical Parallel Programming very nearly meets my criteria for a textbook I would teach from in my own parallel computing courses. The first two chapters reflect the contents of my course quite well. The examples used are the ones I use; the parallel programming techniques presented are for the most part the ones I emphasize. For me, the book’s drawbacks are a lack of student exercises and its focus on the Fortran-K emulator rather than on tools that run on real parallel systems. Although Fortran-K might be a plus for others, I like my students to deal with the very issues the emulator hides.

Still, I would recommend that any of you interested in the practical application of parallel computing—and, in particular, those of you who teach parallel computing—take a look at this fine book.