Protection from Wishful Thinking

Forrest Shull

ONE OF THE thorny problems of software development is accurately assessing progress and the cost remaining to completion. Without a good sense of where the project is and how far it still has to go, it’s just not possible to consistently manage people and resources well. And miscommunicating progress to stakeholders is among the surest ways to lose trust and buy-in.

Despite a plethora of technologies and processes that aim to help teams better assess progress, many software projects still struggle. The complexity of modern systems and the many other day-to-day pressures requiring attention just add to the difficulty. The temptation to interpret progress measures overoptimistically seems near-universal. Optimism is a virtue in many cases, but there is a fine (yet important) distinction between optimism and wishful thinking. Finding ways to distinguish the two is an important skill for every member of a development team.

Even with the best of intentions and good processes in place, it’s possible for projects to end up in serious trouble as a result of misassessing or miscommunicating progress. To illustrate, let’s look at three examples drawn from real software and systems engineering projects. In each, processes were in place and followed that seemed eminently reasonable for avoiding the worst pitfalls. The teams seem to be playing by the rules, although that might not have been what the projects actually needed. I find that these cautionary tales illustrate some key misconceptions—and can perhaps be useful for reflecting on whether our own projects are falling into the same traps.

Confusing Activities with Results

Today’s development teams apply a wide array of processes, each with their own proscriptions about the activities to be executed to obtain the promised benefits and sometimes with competing objectives. An obvious danger of being faced with so many process guidelines is that teams can end up focusing on completing the activities rather than on achieving the end results they’re meant to provide.

Perhaps the clearest example I’ve seen of this misguided focus was a software development effort at a government contractor several years ago. I was doing data analysis to help provide an independent check on software progress metrics. To drive good behavior and an accurate assessment of true project status, the company had instituted an earned value management approach, where the contractor received...
40 percent of the credit for each module when coding was completed, 20 percent after the module passed peer review, and the final 40 percent when unit testing was successfully completed. The intention was certainly admirable: to recognize that we make important progress during development when we take the time to review and improve code quality.

However, in practice, the results left something to be desired. As the data came in, we realized that many of the inspections had been done the day before testing was started for the module—sometimes dozens of inspections were done on the same day. Of course, there was no rule against scheduling the activities this way. However, we could only interpret this as the contractor rushing through inspections because they had to be done (because the activity had to be completed to get credit) rather than because they would provide in-process feedback that would help improve development (because the benefit was desired for the project’s sake).

Although this might seem like a story that could only occur on a process-heavy, big-iteration government project, I think it generates a bit of healthy self-reflection on even small and agile projects: Have we gotten into the mode of doing things because we want to check off a to-do item rather than because we’re trying to get an actual end result? Can we honestly say we’re getting the desired benefits out of, for example, daily standup meetings (do they help make sure we’re still on track and remove obstacles before they impede too much progress)? Or from retrospectives (can we point to examples where we’ve used the information uncovered to actually improve things on the next iteration)?

Unfortunately, these software processes were defined without taking into consideration the connection to upstream considerations. Thus, while those key system requirements were “TBD,” there was a long period during which work on the software requirements proceeded apace, many software requirements were shown as “green” status, and the software team felt that they had everything under control. This was all true right up until some of the system requirements were undefined for an extended period of time while stakeholders and engineers decided what was desirable and feasible. The software component was itself an ambitious undertaking and was developed using well-defined and rigorous processes. Software requirements were distinct from system requirements and were tracked to completion. At first glance, this all seems like an eminently suitable approach.

Good project managers recognize the need to focus on success within their system component but not at the expense of ignoring the rest of the system.

The clearest example of this misconception that I recall was a highly innovative software-intensive system that had more than its fair share of challenges and that was cancelled and re-configured a few years ago. The highly innovative nature of the endeavor meant that key system requirements were undefined for an extended period of time while stakeholders and engineers decided what was desirable and feasible. The software component was itself an ambitious undertaking and was developed using well-defined and rigorous processes. Software requirements were distinct from system requirements and were tracked to completion. At first glance, this all seems like an eminently suitable approach.

I’m happy to announce that Jane Cleland-Huang has joined our editorial board and will be sharing responsibility for the software requirements coverage area with Neil Maiden. Cleland-Huang is an associate professor at DePaul University in Chicago, as well as North American Director for the Center of Excellence in Software Traceability. Her research work includes a focus on collaborative requirements engineering processes and software traceability. Cleland-Huang has an impressive track record of research publications, and her work has earned her five distinguished paper awards at highly ranked conferences (including ICSE10, ASE10, and RE 2002, 2006, and 2007). In addition, her energy and devotion to service to the research community has been exemplary, including (to name just a few positions) associate editor of IEEE Transactions on Software Engineering; program committee member and publicity chair of ICSE; and steering committee member, chair of the doctoral consortium, program chair, workshops chair, and publicity chair of the IEEE International Requirements Engineering Conference.

Cleland-Huang’s prior experience with IEEE Software includes serving as a guest editor in 2008. In addition, she was coauthor on a paper that we selected as a “top pick” for IEEE Software’s 25th anniversary. These top picks highlighted research results that made the most impact, as measured by number of citations in the literature and expert opinion.

**Working in a Vacuum**

Divide-and-conquer is an effective strategy for tackling large development projects. However, an associated risk is that it becomes harder to take a system-wide view, and issues get lost at the boundaries between components. Good project managers recognize the need to focus on success within their system component but not at the expense of ignoring the rest of the system.

The clearest example of this misconception that I recall was a highly innovative software-intensive system that had more than its fair share of challenges and that was cancelled and re-configured a few years ago. The highly innovative nature of the endeavor meant that key system requirements were undefined for an extended period of time while stakeholders and engineers decided what was desirable and feasible. The software component was itself an ambitious undertaking and was developed using well-defined and rigorous processes. Software requirements were distinct from system requirements and were tracked to completion. At first glance, this all seems like an eminently suitable approach.

Unfortunately, these software processes were defined without taking into consideration the connection to upstream considerations. Thus, while those key system requirements were “TBD,” there was a long period during which work on the software requirements proceeded apace, many software requirements were shown as “green” status, and the software team felt that they had everything under control. This was all true right up until some of the system requirements were finally nailed down, at which point a whole contingent of software requirements showed up as “red,” as the software functionality suddenly didn’t meet the newly defined system constraints. This created lots of rework in response to changes that everyone knew were coming.

It’s possible to argue that the
“green” and “red” status of those technical software risks adequately communicated the situation at those two moments in time, but we should ask ourselves whether this actually provided useful and accurate information for any of the stakeholders—most importantly, for the development team members themselves. Surely a more useful approach would have been to be aware of the true risks upfront and to try harder to stay flexible and assess the needs that undoubtedly were coming.

Useful protections against this kind of wishful thinking include being aware of the important interfaces—both to other components and to the larger system—and maintaining communication across them. Inviting stakeholders from interfacing systems to participate in appropriate reviews/inspections is a proven strategy for keeping lines of communication open and maintaining a bit of the system-level viewpoint.

Leaving Estimates Open to Misinterpretation

One of the most common problems I’ve seen is best exemplified in public documentation for a system that failed. The A-12 Avenger II, a stealth bomber developed for the US Navy, was cancelled in 1991 after a history of cost overruns and delays. Because the cancellation decision has been the subject of some controversy and much litigation, the A-12 might in fact be among the systems with the most publicly available analysis.

Before cancellation, the Secretary of the Navy ordered an administrative inquiry to investigate the program’s true status. Although this investigation identified several contributing factors, one in particular has always stuck in my head. Many good practices were in place on the program to keep development on track, including an independent cost analyst who monitored the Cost Performance Index (or CPI, which represented the budgeted cost of the work performed so far over its actual cost). As calculated by the analyst in this case, the CPI was essentially a best-case estimate of the work progress, as she reported. Unfortunately, when the CPI was communicated to the program manager, he chose to interpret it as a worst-case estimate, assuming that the number reported was only a lower bound on the work’s true progress (www.suu.edu/faculty/christensen/evms/beach-a-1.pdf).

Although this is now a 20-year-old example, my observation has been that the human capacity to interpret data optimistically remains undiminished. As project managers, many of us—no matter the project size—still need to fight the temptation to think that our teams will pull together and get the job done, and that estimates of progress and costs are just rough guidelines that we can easily surpass. The temptation to “spin” an estimate is powerful, no matter whether it describes a project, an iteration, or a sprint. However, it’s also true that the more fine-grained the unit of analysis, the less damage is done by inaccurate estimates and the quicker you can notice mistakes and recover from them.

Some of the best ways to fight these temptations are hinted at earlier: it’s better to construct at least two estimates (best and worst case), in order to have some confidence of the range in which the true estimate will really lie. If for some reason only one estimate is produced, be very clear about which end of the spectrum it really lies on. Finally, an advantage of any estimate, no matter how far off, is that it provides a mechanism for analyzing mistakes and doing better next time. If you don’t trust an estimate, go back and find where the bad assumptions were made—just throwing it out because it doesn’t match your gut feeling is not only dangerous, it...
All of these stories contribute to an overarching point: no matter how many innovations and new technologies are introduced for software development, it’s always possible to trip up a project by losing sight of engineering fundamentals. Many engineers, having read these short vignettes, will accuse me of grossly oversimplifying incredibly complex systems and boiling all of their problems down to a too-pat explanation—and they’re right. But I would argue that there’s value in accumulating stories that help illustrate important and fundamental lessons, to help keep them in focus despite all the other day-to-day project pressures. I hope my stories are of some interest to you, but I suspect you all have stories of your own that help explain the key lessons of what it means to be a software engineer.

If you have a short story with a compelling moral that you’d like to share, I’m always happy to learn from them. Feel free to send your stories my way at fshull@computer.org.

**LISTEN TO GRADY BOOCH**

“On Architecture”

podcast available at [http://computingnow.computer.org](http://computingnow.computer.org)