

Last Updated: 16-August-2003

Joe Befumo,
Joe@befumo.com
Shakti Development, Inc.

Doing More With Less

Background

In the late 1990s it often seemed as if Information Technology could do no wrong. Generous investors with deep pockets were only too willing to shovel venture capital into anything with a "dot-com" on the end of its name, rarely peering too deeply into issues of efficiency. Lavish office spaces, often populated with swollen staffs, were the norm. "Burn rate" was the term of the day, profits were a non-issue, and efficiency was a concept so foreign as to never even be considered.

Then came the millennium.

Staffs were cut, workloads increased, and those golden dot-com's started folding so fast it was hard to keep track. Beleaguered IT managers everywhere now face the same problem: how to do more with less. The good news is that there is definitely an answer.

The Methodology Dilemma

To most of us, the terms "Methodology," or "Development Process" invokes images of voluminous tomes of detailed procedures, high-price consultants, and prohibitive reporting procedures guaranteed to send our most valuable developers packing. After all, if we're already having difficulty meeting deadlines, how will adding *more* overhead possibly make things better? Fortunately, there *is* an answer.

Back in 1980, Philip B Crosby wrote a book called *Quality Is Free* (New York, 1980, Penguin Group). In it, he espoused a philosophy of simultaneously decreasing production costs while increasing quality by avoiding costly rework. A cursory review of your current development situation should be sufficient to validate the basis of this philosophy. Typically, we find that the reason we have difficulty meeting deadlines with our new projects is due to the resources we're expending on support of our *old* applications. The obvious difficulty is how and where to slam on the brakes without causing an catastrophic pile-up.

Attitude

The first major obstacle to overcome is our attitude toward a defined process. Purveyors of process are only too willing to tout their own particular offering as the latest *silver bullet* for software development. "Use *our* process and all your IT problems will be resolved . . ." Hogwash! A methodology is *not* about telling you what you *must* do. It's only purpose is to help you to avoid getting blind-sided by unanticipated events. Think of it as a meta-checklist. Sure, you're free to follow or ignore any particular step, just so long as you're sure to evaluate the potential impact of doing so. Of course, some procedures assume a level of *maturity* that can only be supported by more elementary capabilities, and hence, there are some procedures that must logically be implemented before others. In this sense, a well designed process is more than a mere checklist, but rather, represents a hierarchical structure of practices that, if thoughtfully implemented, will streamline your development environment, effectively allowing you to do more with less.

One of the best guidelines for improving the efficiency of any development environment is the *Capability Maturity Model*, developed at Carnegie Mellon's Software Engineering Institute (<http://www.sei.cmu.edu/cmm/cmm.html>). CMM evaluates an organization's maturity into five levels, based on the practices they implement. These levels are:

1. Initial
2. Repeatable
3. Defined
4. Managed
5. Optimizing

I won't be going into the details of these levels in this paper, other than to observe that the *Initial* stage is the one which corresponds to the vast majority of IT shops. Basically, you get things done, but you don't always know how, nor can you predict with any certainty how any particular project is going to progress. Basically, success depends on the heroics of one or two gifted developers who, somehow, always manage to muddle through. So, what's the problem with this?

Well, for one thing, do you want the fate of your company to hinge on a few individuals, loyal though they may be? I'm reminded of a cartoon I saw years ago, which depicted the scene at a graveside funeral service. Two well-dressed management-types had approached the grieving widow, and were saying: "I know this is a bad time, but did he ever mention anything about *source code*?"

More importantly, code wizards almost always have highly optimistic visions of their own capabilities, which generally translate poorly into real-life delivery schedules.

The Typical Software Disaster Cycle

Where do project disasters originate? After nearly thirty years of working with software projects, I believe I have identified the essentials of a universal disaster cycle. It goes something like this:

1. Marketing, or senior management, or some other higher-up approaches the project manager with a list of functions to be delivered, and a time frame in which they must be completed.
2. To the project manager, this appears unreasonably ambitious, but he dutifully goes to his top technical guru for an opinion.
3. "I don't know," replies the guru. That looks like a lot of work. *Maybe* I could get it done, if I could devote myself to it 100%, but as you know, I'm in the middle of bug fixes for Project-X, and enhancements for Project-Y, and we just committed to finish that report generation facility for Mr. Enduser."
4. The project manager goes back to management, and points out that everyone already has a pretty full plate, so it's unlikely that they'd be able to meet that schedule.
5. Management says something like: "We have faith in you. Just do your best.", to which the project manager utters those fateful words:
6. "I'll try."
7. The "I'll try," immediately becomes "I absolutely guarantee that this project will be 100% done at the hour, minute, and second identified.

Before we go on, let's stop for a moment, and take a closer look at what happened here.

First of all, as is often the case with software gurus, the off-the-cuff "maybe I could get it done" probably means "I could probably hack together something that might run under ideal circumstances." That is, the guru thinks in terms of writing code. There was probably no consideration of making sure that the functionality requested was realistic and consistent, and certainly no allowance for testing and rework.

Secondly, the project manager was never in a position to *realistically* respond to upper management, because he was unable to produce a detailed plan of what the project would entail. Hence, arguing with management would undoubtedly result in some response along the lines of: "Oh, you guys *always* whine that it can't be done, but you always manage to do it. . . ."

What nobody has noticed thus far is the reason why the software guru is now mired down in bug fixes and "enhancements" for Project-X and Project-Y; that is, at some time in the past, those projects were declared "done," when they were anything but.

Let's follow the cycle a bit further.

7. The "I'll try," immediately becomes "I absolutely guarantee that this project will be 100% done at the hour, minute, and second identified.
8. Due to the unrealistic schedule constraints, no time is "wasted" on analyzing the requirements, or ensuring that the design will integrate properly with existing applications or the overall environment; in other words, the guru simply starts writing code.
9. Somehow (late nights, weekends, many pizza deliveries, and gallons of black coffee), something resembling the desired application is delivered on time, however . . .
10. The application contains numerous bugs which will have to be resolved,
11. Since the requirements were never fully verified, additional functionality will inevitably be required,
12. Because no detailed design was ever completed, enhancements will have to be "hacked" into the code, which will result in additional bug fixes.

Hence, the developers plate becomes increasingly loaded with nebulous overhead, thereby impairing the ability to approach new projects in a more rational manner.

Breaking the Cycle

So how can this disastrous cycle be broken? To answer this, let's go back to our defined process. Most methodologies include some kind of framework project plan that includes *all* of the steps inherent in a generic project. Will *all* of these steps be applicable to every project? Of course not, however, they *will* provide a way of ensuring that nothing is inadvertently forgotten or glossed over. Working with the development guru, as well as key members of the prospective project team, the project manager can now come up with a preliminary detailed project plan. With this, he can go back to management with something more in his hand than intuition and generalities.

A Risk-Based Approach

Of course, the project manager can't expect to simply shove this list of tasks in front of his superior and expect immediate and unconditional capitulation. Nevertheless, he is now in a position to say "yes, we *can* eliminate this step, however, *these* will be the ramifications of doing so. . . ." It's important, at this point, to emphasize that the point of this exercise is *not* simply a matter of CYA (cover your ass). Presumably, we're all on the same side, and the intent is to enable everyone involved to make logical, informed decisions. For example, I was once project manager for a product being developed by a venture-capitalized start-up company. We had precisely enough money to last us for another four months. We also had several key customers who had stated that

they would purchase the product if it had certain added functionality. There was no way that we could "follow the rules," and still get that functionality added in the time available. We either had to cut some corners, or we'd be out of business. This type of situation is not at all atypical, and if your methodology can't accommodate it, then it's not a real-world methodology. The key point in this situation was that we were walking in with our eyes wide open. We *knew* that today's shortcuts would have to be paid for in the future, and consequently, we'd have to allow for it in our future development plans.

Talk is cheap

One of the most important purposes of any methodology is to ensure that consensus is employed in a timely and effective manner. Consensus is the most powerful tool available in any development environment—when used properly. Conversely, if applied at the wrong time it's a sure way to drag any project down an interminable rathole. As a simplistic example, consider a scenario in which a development team has a basic set of requirements, but no code has been written. The team begins to throw out ideas about how the system will be created. For the purposes of this discussion, it really doesn't matter whether you're using object oriented techniques, or old fashioned structured programming. The real issue is that you're talking about the right things at the right time; namely, the time *before* anyone is heavily invested in one approach over another. Talk, as they say, is cheap.

"We can have a module that formats widgets," one engineer ventures.

After some discussion, they conclude that this will actually turn out to be a hierarchy of six modules (or classes, or whatever.)

The team then begins looking at each class, or module, and discussing the size or complexity, and hence, the development effort involved. Similarly, the brainstorming effort begins to identify dependencies in the construction order of the modules, what test harnesses will be needed to exercise them, and so forth. No estimates have yet been given to management, so there are no expectations to control yet. If, when the final estimate is presented, those in charge wish to balk, the team leader will be in a position to lay out the plan and ask "Okay, which parts do you want to cut out." The team leader, being more confident of the plan and estimate, is less inclined to wince, squirm, and say "okay, I'll try. . . ."

This is an example of a project getting off to a good start.

The alternative, of course, is to hold the same discussion two months into the project, when customer expectations have been improperly set, the project is behind schedule, and everyone has their own body of code to protect. Typically, the team sits down in the same room, in front of the same whiteboard, and asks the same questions. The results, however, are predictably different. The ideas that flow are selfish in nature, and aimed at protecting individual interests, not the integrity of the project as a whole. The estimates

that emerge can not be justified to management, because to do so would be to admit how much time has been wasted, and how much effort must be directed to rework that could have been prevented.

I would argue, therefore, that the purpose of a process is *not* to tell you what to do, or how or when to do it, but rather, what to discuss, what to consider, who to confer with, and when to hold those conversations.

Nuts and Bolts

The above discussion is admittedly philosophical in nature. There are, however, numerous practices that can be implemented in any organization without adversely affecting projects already underway. Specific practices, of course, will depend on what your organization already has in place, however, a good place to start looking is CMM's key practice areas. For example, key practices for CMM Level II are described as :

"The key process areas at Level 2 focus on the software project's concerns related to establishing basic project management controls. They are Requirements Management, Software Project Planning, Software Project Tracking and Oversight, Software Subcontract Management, Software Quality Assurance, and Software Configuration Management."

If I had to select just *one* area for initial emphasis, it would be Software Configuration Management. Personally, I use a configuration management tool (Microsoft SourceSafe) for *everything*, not just software development. When I write, every change in my in-process manuscript goes directly into configuration control. Of course, the full benefits of a Software Configuration Management system are only realized in a multi-contributor environment. Time saved by not chasing after consistent versions of every module translates directly to your bottom line. It's like finding free resources you didn't even know you had. Moving up the scale, every improvement measure should be approached in the same way--that is:

Quality (and Process Improvement) is FREE

The result of implementing any process improvement measure should be immediate and definitive. Every measure should *pay*, not *cost*, otherwise there's no point in doing it. Doing More With Less, therefore, is simply another way of working *smarter* instead of *harder*.

Biographical Sketch

Joe Befumo is a Senior Managing Partner and Chief Technology Officer at Clarity Development, LLC. After working in software development since 1974, he

became interested in process improvement issues in 1989, while at Digital Equipment Corporation's Advanced Semiconductor Development Group . As a methodologist and head of AT&T Solutions' Center for Estimating and Metrics he contributed to the development of software processes geared toward the large-scale distributed development projects. He has been Chief Technology Officer of Clarity Development since 1997.