Copyright (c) 1997, IEEE Computer Society

Author Contact: james@satisfice.com

The Hard Road From Methods to Practice

James Bach, ST Labs

We who read this magazine and also work on real projects are caught between two worlds. In one, we absorb the advice of pundits on the proper development of software. In the other, we huddle in our cubicles and actually do it. Unfortunately, these worlds are fundamentally disconnected and sometimes even unrecognizable to each other. This leads to what I call the *methodology gap*. The wider the gap, the less relevant software engineering ideas are to practice.

METHODOLOGY GAP

Everyone who writes articles, makes presentations, or specifies any sort of policy intends their work to be useful. But usefulness is situational. To say that a hammer is useful is to say that we know of important situations in which some worthwhile benefit would be gained by using a hammer. A useful hammer is necessarily a component of a *good solution to a problem that matters*. Otherwise a hammer, although it may be an aesthetically interesting sculpture, isn't useful.

To create any useful tool, therefore, we must anticipate both how it could be used *and* how it would be worth the effort and cost to use it. We must then have the skill to design the tool and the ability to deliver it to a client in such a way that they see its usefulness and are inclined to use it. Software engineering methods—by which I mean any strategic or tactical ideas intended to guide the practice of developing software—are tools, too. A methodology gap exists whenever a stated method doesn't correspond to effective practice.

A methodology gap can manifest in several ways. It could be that a method *can't* be practiced, *shouldn't* be practiced, or simply *isn't* practiced as designed. The gap is not necessarily a bad thing—we can productively strive for ideals that will never be perfectly achieved. But when such a gap goes unrecognized, or calcifies into a tradition of self-deception, it can suck the marrow out of your efforts to control or predict your software development process.

The causes of a methodology gap are usually subtle and complex. Methodologists or management may not understand the realities faced by practitioners. Practitioners may not understand or respect the methods. Or they may forget them accidentally or apply them recklessly. The methods may be poorly designed or poorly communicated.

The gap can also perpetuate itself. When practitioners know that publicly accepted methods aren't really followed, what incentive do they have to embrace any new idea that comes along? Process hypocrisy leads to process cynicism and vice versa. A vicious cycle.

SOFTWARE PROCESS IS DYNAMIC

Slippery language encourages the confusion of method with practice. It's hard to communicate precisely on this subject. The dictionary definitions of "method" (a means or manner of performing or effecting something), "practice" (the act or process of doing something), and "process" (a series of actions, changes, or functions bringing about a result) are easily confused. To me, *method* is an idea that guides practice, *practice* is a practitioner applying a method to cause an event, and *process* is the resulting pattern of events. I take a dynamic, systems view of projects; I believe that process emerges as multiple practitioners apply multiple methods in response to their moment-by-moment understanding of the state of the process.

The view deepens further when I consider that any software development practice worth talking about involves significant problem-solving. And that any problem-solving activity is potentially an open-ended project, dependent on the skill and motivation of the practitioner. Competent software people don't simply "follow" methods, the way one might follow a mechanical formula for long division. People breathe life into methods.

HEROES MAKE THE SYSTEM GO

The ability and propensity of people to both invent processes on the fly and adapt processes to the circumstances at hand is indispensable. That is why heroism—at least heroic ingenuity and occasionally heroic effort—is essential to software engineering. Heroes are people who take initiative to solve open-ended problems. Our methods should be designed to empower heroes and help them work together: How to be a responsible hero and how to manage heroes should be part of the official body of knowledge that defines our craft.

We who read this magazine and also work on real projects are caught between two fundamentally disconnected worlds.

Why must we rely on heroes? Why not make methodology more complete and sophisticated so we can succeed without heroes? Because that's impossible, by definition. Software development without human problem-solving is not called development, it's called a compiler.

Useful tools, however, can improve the reliability and capability of heroes. Which brings us to defined methodologies.

Methodology documents are notoriously incomplete and oversimplified. This

is not necessarily a problem, of course, if the people who design methods share enough of a reality with those who will interpret and apply them. Show an electrical engineer an electronics schematic and he can build a stereo. Show the same schematic to an untrained person and he won't know what to do with it. At ST Labs, I ran into this problem when I first defined test processes for our staff. At the time, almost all of our testers based their process on these simple principles: understand the product, try its major functions, report anything that looks wrong. This strategy led to a wide variation of processes, some terrible and some good enough, depending on the skill of each tester and the particulars of the product.

When I tried to define our methods, I found myself struggling with ever longer descriptions, descriptions that fragmented into families of subdescriptions as I tried to convey basic information as well as variations in technique. I eventually realized that defining high-level instructions for general testing was not going to be useful until we trained everyone about what test techniques were in the first place. Good method documentation goes hand in hand with a staff skilled in interpreting those descriptions so that they solve the right problems at the right time. Methodology and the people who practice it are one system.

Skill in ergonomic design and method documentation is rare. Most methodologists settle for simple linear descriptions of software processes that aren't linear at all. They write processes as if they were writing programs for robots. They write for rule followers rather than problem-solvers. They smooth out all the political incorrectness of actual software development and create documents that are hard to use. Then they drop them onto an organization like propaganda leaflets from a spy plane, trusting management to sort everything out.

I don't want to be too hard on the methodologists, though. I've designed a lot of bad methods myself. I have a drawer full of them. It isn't so easy to tell when methods are poorly designed or documented. A lot of methods can work, in principle, if we only devote enough energy to them. People have been known to quit smoking cold turkey, after all. It could easily be argued that this method of quitting smoking is sound; it simply requires the quitter to exert enough willpower. It's certainly easy for me to argue this, since I've never smoked. Perhaps better management is what we need?

THE MYTH OF MANAGEMENT POWER

It's commonly assumed that methodology is so strongly related to practice that, given adequate management, methods and practice are practically the same thing. Set your methodology and, if management does its job, practice will follow. The complexity of software process ideas and realities are thus compressed into a neat morality play, as we solemnly scold that "process improvement begins at the top."

Management is the big bear rug under which we sweep our most unsettling problems. The truth is, management is impotent to directly influence software-development practices. Managers have little true power to control intellectual workers. As chief engineer at ST Labs, I theoretically have the authority to dictate to any tester at ST Labs what is and is not an acceptable testing practice. Big deal. There's no way I can know, except in little bits and snippets, what anyone is really doing. If I do encounter someone who is clearly and presently doing a poor job of testing, I have to walk on eggshells about it or risk crushing morale. For one thing, I can't know all the circumstances surrounding the situation, for another, anything I say or do will be, at best, a temporary fix. It takes sustained training and mentoring to effect real performance improvement.

> The complexity of software process ideas and realities cannot be compressed into a neat morality play.

It's a fantasy to think that top management has the power, by simple force of will, to assure that good methods are followed. Using direct authority to force or intimidate intellectual workers only creates the kind of resistance that absolutely guarantees a methodology gap. Management is powerful, but it takes more talent and training than most managers have to coax an organization to improve quickly.

Software development is a problem-solving enterprise of fantastic complexity, engaged under variable conditions with limited time and resources, for the purpose of creating a good enough product. Methodology development is a similar enterprise, but one whose methods are even less well understood than those of software development. There are few if any professional organizations for methodology developers, few resources, and the art is primitive. Methodology development requires a rare bundle of skills—from technical writing to politics to general systems analysis.

If software development practices and methodologies are to come together, their realities must be reconciled. That's where we come in, we the readers of magazines like this. To get a handle on those realities, we need to be software project naturalists first and moralists second. We must observe how systems do work, before deciding how they should work. We must acknowledge the limitations of our observations and the many overlapping and contradicting values that must be accommodated. We must learn how to communicate methods in such a way that practitioners are empowered, not penned in. There are a few pioneers who have made headway on this path: Gerald Weinberg, Bob Glass, Tom DeMarco, Tim Lister, Ed Yourdon, and Fred Brooks, to name a few. I can provide a full bibliography, if you're interested.

As a software testing methodologist, it's my job to question emperors and tip sacred cows. The number one occupational hazard for testers is tunnel vision, and I can't help but feel that this industry is in a bit of a tunnel. I'll try my best, with this department, to find some daylight. So long as our true practices are shrouded by a false view of our methods, we will be frustrated in our efforts to close the gap between our current experience and that grander success we keep reading about. ❖