XP, Iterative Development, and the Testing Community

I was disappointed with a recent attack on Extreme Programming in StickyMinds, "XP: That Dog Won't Hunt." What concerns me is the broader slam of iterative software development. The author said, "XP's fundamental premise that the software will be done when it's done and will cost what it costs is the antithesis of sound business practice. . . . If XP proponents don't successfully address these problems, XP will suffer the same fate as the long list of other iterative practices that preceded it."

I am constantly surprised by testers who denigrate the iterative approaches. They characterize the iterative alternatives as undisciplined, unsound, and irresponsible. I think that in the process, they make the wrong enemies, lose the respect of the wrong people, and cut themselves off from opportunities to address serious, chronic problems in software development.

Under the waterfall (the primary non-iterative software development process) , a "complete" set of "requirements" is defined in advance (not necessarily correctly), then the system is designed (not necessarily well), then coded, and then tested. There is some allowance for redefinition and redesign, but so much that comes later depends so heavily on what has come before that the process is brittle. Changes become exponentially more expensive as the project goes forward because so much depends on what has been done (and tediously documented) already. The exponential increase in cost of fixing bugs, for example, is well documented.

Reluctance to fix bugs is an inherent problem of the waterfall, which places testers (change recommenders) late in the project, when every change they recommend will be overpriced (because late changes are exponentially more expensive than earlier changes.) In such a process, *of course* our change requests will be bounced.

Another characteristic waterfall problem is the conflict between testers and project managers. A project manager trades four key factors against each other:

- time to delivery
- cost,
- reliability and
- feature set (scope).

Consider these tradeoffs in the waterfall. What happens if the project is behind schedule when it reaches the testing phase? Well, there's not much point cutting features because they've already been specified, designed, documented, and coded. Most of the project budget has been spent. So we trade time (ship now but buggy) against testing (ship late). Testers fight project managers for more time. I guess some people in the field like these wars. We hear enough about them at conferences--people get together to complain about those nasty developers year after year.

In contrast, the well-planned iterative project starts from the expectation that people (developers, users, executives) are not very good at figuring out how they will actually use the product, estimating costs, prioritizing features, or anticipating the problems they will actually encounter

during development. It is designed to help us manage the risks associated with errors and omissions in our assumptions and estimates.

We start with estimates. We build consensus around a broad vision for the product (rather than pretending to build consensus around the details), list features of interest and guesstimate their implementation times/costs, work with key stakeholders to prioritize the features, and then get moving on the implementation. Each time we add a feature, we also test it and fix it, stabilizing the product at the desired level of quality. Then we design, code, test and fix the next feature, and the next. The product is incomplete, but from a point very early in development, the product is usable and testable and tested. From that point forward, people can use the product and give experience-based feedback that reveals erroneous designs, not-previously identified requirements, and bugs. The feedback drives changes in the priorities and details of what is implemented next.

Iteration gives management control over the schedule, cost and feature set. They get increasingly accurate estimates because we regularly update our estimates of time/cost remaining to completion. They can stop new development at almost any time and (given a few final days of testing and fixing) have a product that works. I have personally seen this used to salvage a project that had gone out of schedule control. (They shipped a product that worked, with fewer features.) I have personally used this to salvage a project that lost critical staff and could not be completed as planned. (We shipped the product early. It worked, but with few features. We then planned for a Version 2, as a separately budgeted project, to add all the features we had lost from this one.)

Note the tradeoff here. Features versus time. Not reliability (and testing) versus time. What we release will work, in contrast with the waterfall, where we often release products with tons of features, only some of which work.

The politics of iteration are also different. The conflict is no longer between testers and project managers. Instead, the people who cope with late tradeoffs are senior managers, marketing managers, development managers, and technical writers. The managers get to balance (and fund) the tradeoff between more time and more features. In XP, the managers have been able to prioritize the features throughout the project, so these tradeoffs should not be as bitter.

The nature of the tester's role changes in these projects. We are no longer the high profile victims, we are no longer the lonely advocates of quality, we are merely (!) competent service providers, collaborating with a group that wants to achieve and is working to achieve high quality.

I think these are positive changes. I think we should consider them carefully before dismissing iteration in favor of processes that are as badly broken as the waterfall.

About the Author. Cem Kaner is Professor of Computer Sciences at the Florida Institute of Technology, where he teaches courses on testing, software measurement, empirical research methods, and computer law and ethics. He is senior author of three books, *Lessons Learned in Software Testing, Bad Software,* and *Testing Computer Software.* He's also an attorney (a former prosecutor) whose idea of a good time is holding companies accountable for releasing defective software.