Good Enough Software

Most IT organizations assume that their users would like them to develop software instantly, at no cost, and with no defects. But that's just not possible in today's world; in more and more application domains, we've been forced to accept the fact that the reengineering slogan of "faster, cheaper, better" really means "fast enough, cheap enough, good enough." In the past few months, the concept of "good enough" software has been getting a lot of discussion: the uproar over the Pentium bug suggests that it was deemed not good enough, while the surprisingly large numbers of defects that are publicly acknowledged in popular shrink-wrapped software products -- e.g., word processors, spreadsheets, tax calculation programs, and PC operating systems from a variety of the best-known software companies -- suggests that those products *are* good enough.

All of this is beginning to challenge some of our basic assumptions about software development, and I believe that it will lead to some fundamental changes in the way we manage software; thus, it's going to bring about the same kind of radical reengineering in IT organizations that we have imposed upon other parts of our companies. Some purists -- especially the long-time professionals of the computer industry -- may express horror at the "flight from quality." But I think it will lead to more rational software development projects, and a more rational way of negotiating the criteria for successful projects with our customers and managers.

In the past, we often negotiated critical project success parameters *once*, at the beginning of the project, and then attempted to optimize a few other parameters that the customer was often unaware of. For example, the functionality, schedule, budget, and staff resources were typically negotiated in terms of political constraints: we were told that our task was to deliver a software system with a certain amount of functionality (which was often ambiguous, misunderstood, and poorly documented) within a certain schedule and budget (which was based on hysterically optimistic estimates, or simply imposed by managerial fiat), and with a relatively fixed staff of developers. Within those constraints, the developers often attempted to optimize such features as maintainability, portability, reliabiity, and efficiency. Thus, the battle cry for many projects was: "We'll deliver high-quality, bug-free software on time, within budget!"

For an important class of software projects, that battle cry is still relevant: obviously, nobody wants to fly on an airplane whose guidance control software has as many bugs as our PC word processor. Nobody wants their telephone system or their bank's ATM system to crash as often as their desktop operating system. But for another class of software projects -- which is arguably far larger today than the class of "critical" software systems -- rapid delivery of the software to the customer is sometimes more important than the number of defects it contains. In other situations, "feature richness" may be the dominant factor; in still others, cost might the only thing the user cares about.

Much of the shift that we're now experiencing is associated with the fact that information technology is now a consumer commodity: unit costs are low, and everyone can have it. In the past, most of us worked on customized, proprietary, one-of-a-kind systems with schedules measured in years, and budgets measured in millions of dollars. Today, some of us are still employed by organizations who want customized systems - but the schedules and budgets have shrunk considerably. And the customers will often point out to us that they can achieve the almost the same results by lashing together a jury-rigged combination of Microsoft Word, Lotus Notes, and Borland Quattro, which they can obtain from a mail-order discount catalog. The shrink-wrap software may be clumsy and limited in its functionality for the user's application, but it's cheap -- and it can be put into service tomorrow morning.

There's also an issue of inertia that we have to cope with, especially in the consumer-level desktop marketplace. Suppose, for example, that you have *no* word processor on your desktop PC and the time has come to acquire one. You have a choice between products A, B, and C. Product A costs $500 and it comes with a money-back guarantee; product B costs $100 and comes with a long legal disclaimer at the front of the user manual that basically says "caveat emptor." Meanwhile, product C costs only $50 and has twice as many features as A and B, and its developers are so confident of its quality that they're bragging about their double-your-money-back guarantee -- the only problem is that it's vaporware, and despite all of the glowing reviews in the trade magazines, it won't actually be available for six months. Assuming that you need a word processor *now,* presumably you would make a rational choice between A and B, based on your assessment of the importance of cost versus defects.

But suppose that you already have a word processor -- perhaps product B -- installed on your computer, and you've been using it for the past year. Some of its features are slightly annoying, but it's adequate for accomplishing your mundane, daily word processing tasks. And suppose that it has become evident to you that the quality isn't all that great; it crashes once a day, and you've become accustomed to saving your documents every 15 minutes to avoid losing work in progress. Now vendor C finally delivers its product to the marketplace, and it really *does* cost only $50, and it really *does* have a level of quality ten times higher than your existing product. Would you bother switching? Maybe -- but maybe not. What if product C required you to convert all of your existing word processing documents to a different format? What if it required you to switch to a different operating system? You might well conclude that product B was "good enough" -- and the project manager for product B might well have outsmarted the project manager for product C, even though C was pursuing a set of goals that we would all admire as software professionals.

The point is that the software project manager has to be aware that *each* of the project parameters -- cost, schedule, staffing, functionality, and quality -- is potentially critical today, and it is up to the customer (which may be the end-user for an inhouse system, or the marketing department for a software product company) to decide what the proper balance is. It's also crucial to remember that the balance between the parameters is a dynamic one, and may need to be readjusted on a daily basis. After all, the business environment is likely to change in a dramatic, unpredictable way -- and this can easily change the customer's perception of the importance of schedule, cost, etc.

As a mental abstraction, any intelligent customer -- especially one who has survived in today's tumultuous business times -- is aware of the need to make trade-offs and balance priorities. But customers are often naive about the details; for example, it may not occur to them that defects (aka "bugs") are an aspect of the software that we have to consciously plan for, and for which we have to establish trade-offs in terms of other parameters. And, of course, they may not want to make cold-blooded, rational, calculated decisions about those tradeoffs: it's understandable (even though immensely frustrating) for a customer to demand a software system in half the time, and half the cost, with half the staff and twice the functionality, and half as many defects as the developers believe is technologically possible.

What does this mean for the IT project manager? If we can assume for the moment that we're dealing with rational customers, and that a rational negotiation can determine the criteria for project success, then it is incumbent upon the manager to be as forthright and detailed as possible about *all* of the relevant success criteria. Thus, instead of just assuming that zero-defect quality is required, the project manager should be able to say something like, "Our standard approach for developing the software you've described will require X number of people, and Y units of time, with a cost of Z dollars; we'll deliver P units of functionality, with a defect level of Q bugs per function point."

Chances are that the proposed combination of X, Y, Z, P, and Q will *not* be acceptable to the customer; the likely response might be "You can't have Y units of time; we need to have the software in half that much time." A less rational response might be, "We want twice as much functionality as you proposed, but you can only have half as many people, half the time, and half the cost." This is possible, assuming (unrealistically, most likely) that the customer completely relaxes the constraint on the number of defects; after all, I can deliver an infinite amount of software, with an infinite amount of functionality, in zero time ... if it doesn't have to work at all. The least rational response of all, from our customers, is one that contrains *all* of the project success parameters to some level that is demonstrably unachievable.

It *is* perfectly rational for our customers to challenge our proposal for X, Y, Z, P, and Q -- particularly if we can get them to focus their attention on one parameter at a time. If the user wants the software in 0.5 units of time, then it's incumbent on us to provide a counter-proposal that shows the impact that such a change will have upon one or more of the other parameters. Some twenty years ago Fred Brooks reminded us, in his textbook *The Mythical Man-Month* (the second edition of which has just been published), that time and staff-resources are not interchangeable in a linear relationship; if we reduce the project schedule in half, it will *more* than double the required staff. Or we can cut the schedule in half, keep the staff constant, and increase the cost in a non-linear fashion (e.g., by having the constant-level staff work extraordinary levels of overtime).

The mathematics of the relationships between X, Y, Z, P, and Q are something we don't know enough about at our present level of software engineering. Larry Putnam and Ware Myers have explored this in their book, *Measures for Excellence* (Prentice Hall, 1992), but much more work is necessary. Similarly, some of the commercial estimating packages allow for some exploration of the tradeoffs between these parameters when establishing the initial project estimates and plans -- but they rarely allow for dynamic renegotations once the project has commenced. Renegotiation may not be all that important on a project that only takes three months; but in a project that extends over a year or two, it's almost inevitable in today's turbulent business environment.

While the precise nature of the mathematical relationships has yet to be developed in detail, we have enough information today -- especially from the work of such metrics experts as Larry Putnam, Howard Rubin, and Capers Jones -- to provide a reasonable basis for a rational discussion of the issues with our customers. The biggest difficulty, I believe, will be one of politics and management. Getting our customers to engage in such a rational negotiation will probably require some extensive education; but getting our project managers to negotiate in this fashion will be equally difficult.

It will be indeed difficult to say to our customers, "I'm going to deliver a system to you in six months that will have 5,000 bugs in it -- and you're going to be *very* happy!" But that is likely to be the world many of us are likely to live in for the next several years.