Chapter 4. Four Variables

> *We will control four variables in our projects—cost, time, quality, and scope. Of these, scope provides us the most valuable form of control.*

Here is a model of software development from the perspective of a system of control variables. In this model, there are four variables in software development:

- Cost
- Time
- Quality
- Scope

The way the software development game is played in this model is that external forces (customers, managers) get to pick the values of any three of the variables. The development team gets to pick the resultant value of the fourth variable.

Some managers and customers believe they can pick the value of all four variables. "You are *going* to get all these requirements done by the first of next month with exactly this team. And quality is job one here, so it will be up to our usual standards." When this happens, quality always goes out the window (this is generally up to the usual standards, though), since nobody does good work under too much stress. Also likely to go out of control is time. You get crappy software late.

The solution is to make the four variables visible. If everyone—programmers, customers, and managers—can see all four variables, they can consciously choose which variables to control. If they don't like the result implied for the fourth variable, they can change the inputs, or they can pick a different three variables to control.

Interactions Between the Variables

Cost—More money can grease the skids a little, but too much money too soon creates more problems than it solves. On the other hand, give a project too little money and it won't be able to solve the customer's business problem.

Time—More time to deliver can improve quality and increase scope. Since feedback from systems in production is vastly higher quality than any other kind of feedback, giving a project too much time will hurt it. Give a project too little time and quality suffers, with scope, time, and cost not far behind.

Quality—Quality is terrible as a control variable. You can make very short-term gains (days or weeks) by deliberately sacrificing quality, but the cost—human, business, and technical—is enormous.

Scope—Less scope makes it possible to deliver better quality (as long as the customer's business problem is still solved). It also lets you deliver sooner or cheaper.

There is not a simple relationship between the four variables. For example, you can't just get software faster by spending more money. As the saying goes, "Nine women cannot make a baby in one month." (And contrary to what I've heard from some managers, eighteen women still can't make a baby in one month.)

In many ways, cost is the most constrained variable. You can't just spend your way to quality, or scope, or short release cycles. In fact, at the beginning of a project, you can't spend much at all. The investment has to start small and grow over time. After a while, you can productively spend more and more money.

I had one client who said, "We have promised to deliver all of this functionality. To do that, we have to have 40 programmers."

I said, "You can't have 40 programmers on the first day. You have to start with one team. Then grow to two. Then four. In two years you can have 40 programmers, but not today."

They said, "You don't understand. We have to have 40 programmers." I said, "You can't have 40 programmers." They said, "We have to."

They didn't. I mean, they did. They hired the 40 programmers. Things didn't go well. The programmers left; they hired 40 more. Four years later they are just beginning to deliver value to the business, one small subproject at a time, and they nearly got canceled first.

All the constraints on cost can drive managers crazy. Especially if they are focused on an annual budgeting process, they are so used to driving everything from cost that they will make big mistakes ignoring the constraints on how much control cost gives you.

The other problem with cost is that higher costs often feed tangential goals, like status or prestige. "Of course, I have a project with 150 people (sniff, sniff)." This can lead to projects that fail because the manager wanted to look impressive. After all, how much status is there in staffing the same project with 10 programmers and delivering in half the time?

On the other hand, cost is deeply related to the other variables. Within the range of investment that can sensibly be made, by spending more money you can increase the scope, or you can move more deliberately and increase quality, or you can (to some extent) reduce time to market.

Spending money can also reduce friction—faster machines, more technical specialists, better offices.

The constraints on controlling projects by controlling time generally come from outside—the year 2000 being the most recent example. The end of the year; before the quarter starts; when the old system is scheduled to be shut off; a big trade show—these are some examples of external time constraints. So, the time variable is often out of the hands of the project manager and in the hands of the customer.

Quality is another strange variable. Often, by insisting on better quality you can get projects done sooner, or you can get more done in a given amount of time. This happened to me when I started writing unit tests (as described in Chapter 2, A Development Episode, page 7). As soon as I had my tests, I had so much more confidence in my code that I wrote faster, without stress. I could clean up my system more easily, which made further development easier. I've also seen this happen with teams. As soon as they start testing, or as soon as they agree on coding standards, they start going faster.

There is a strange relationship between internal and external quality. External quality is quality as measured by the customer. Internal quality is quality as measured by the programmers. Temporarily sacrificing internal quality to reduce time to market in hopes that external quality won't suffer too much is a tempting short-term play. And you can often get away with making a mess for a matter of weeks or months. Eventually, though, internal quality problems will catch up with you and make your software prohibitively expensive to maintain, or unable to reach a competitive level of external quality.

On the other hand, from time to time you can get done sooner by relaxing quality constraints. Once, I was working on a system to plug replace a legacy COBOL system. Our quality constraint was that we precisely reproduce the answers produced by the old system. As we got closer and closer to our release date, it became apparent that we could reproduce all the errors in the old system, but only by shipping much later. We went to the customers, showed them that our answers were more correct, and offered them the option of shipping on time if they wanted to believe our answers instead.

There is a human effect from quality. Everybody wants to do a good job, and they work much better if they feel they are doing good work. If you deliberately downgrade quality, your team might go faster at first, but soon the demoralization of producing crap will overwhelm any gains you temporarily made from not testing, or not reviewing, or not sticking to standards.

Focus on Scope

Lots of people know about cost, quality, and time as control variables, but don't acknowledge the fourth. For software development, scope is the most important variable to be aware of. Neither the programmers nor the business people have more than a vague idea about what is valuable about the software under development. One of the most powerful decisions in project management is eliminating scope. If you actively manage scope, you can provide managers and customers with control over cost, quality, and time.

One of the great things about scope is that it is a variable that varies a lot. For decades, programmers have been whining, "The customers can't tell us what they

want. When we give them what they say they want, they don't like it." This is an absolute truth of software development. The requirements are never clear at first. Customers can never tell you exactly what they want.

The development of a piece of software changes its own requirements. As soon as the customers see the first release, they learn what they want in the second release...or what they really wanted in the first. And it's valuable learning, because it couldn't have possibly taken place based on speculation. It is learning that can only come from experience. But customers can't get there alone. They need people who can program, not as guides, but as companions.

What if we see the "softness" of requirements as an opportunity, not a problem? Then we can choose to see scope as the easiest of the four variables to control. Because it is so soft, we can shape it—a little this way, a little that way. If time gets tight toward a release date, there is always something that can be deferred to the next release. By not trying to do too much, we preserve our ability to produce the required quality on time.

If we created a discipline of development based on this model, we would fix the date, quality, and cost of a piece of software. We would look at the scope implied by the first three variables. Then, as development progressed, we would continually adjust the scope to match conditions as we found them.

This would have to be a process that tolerated change easily, because the project would change direction often. You wouldn't want to spend a lot on software that turned out not to be used. You wouldn't want to build a road you never drove on because you took another turn. Also, you would have to have a process that kept the cost of changes reasonable for the life of the system.

If you dropped important functionality at the end of every release cycle, the customer would soon get upset. To avoid this, XP uses two strategies:

1. You get lots of practice making estimates and feeding back the actual results. Better estimates reduce the probability that you will have to drop functionality.
2. You implement the customer's most important requirements first, so if further functionality has to be dropped it is less important than the functionality that is already running in the system.