Key Things We Have Learned

Lawrence H. Putnam
Ware Myers

A quarter of a century ago software development was in trouble, as some of it still is today. Projects were having trouble completing on time and within budget. Defects remaining at delivery were too high. Often managers had to cut back heavily on features and accept poor quality in order to deliver at all.

There was a way out of these difficulties then. There still is. That way grew out of the study of the core metrics on a number of projects. These fundamental metrics are:

1. Schedule time,
2. Effort (which in software development is proportional to cost),
3. Functionality (or amount of work to be done), commonly expressed in terms of size or source lines of code,
4. Defect rate, as an indicator of the quality being achieved,
5. Productivity of the development process.

A quarter of a century ago, Larry Putnam was working for an organization that had been keeping data on these metrics. He had some hard facts to work with. After several years analyzing this data, he had learned five key things:

1. There is a *minimum development time* below which a system cannot be successfully completed.
2. There is a useful *tradeoff* between time and effort.
3. There is functional *coupling* between size, schedule, effort, and reliability--change one, the others all change.
4. There is great *payoff* from improving the productivity of the development process.
5. There is *no silver bullet*--Fred Brooks is right.

Putnam has applied these five findings ever since. Not only did they come out of hard data, they have worked for decades now on thousands of projects in hundreds of companies. These five key findings are eminently practical. Get on board!

In more recent years he added one more finding to the list:

6. The way you *size* a project changes to accommodate new development methods.

Minimum development time still lives

This maxim is simple and founded on a simple reality. Twenty-five years ago we had examined the data on only hundreds of projects. None of them had been completed in less

than this minimum development time. Now we have data on thousands of projects. The dictum still holds. It holds no matter how many people are applied to the project!

Of course, projects differ in a number of respects: size of the proposed system, difficulty of the application type, and productivity level of the project organization, For each combination of these three factors, there is a different minimum development time. But we have learned the behavior from the data; that, in turn, produced an algorithm so that we can find the minimum development time in each case.

You can't overwhelm a project by adding people to it--beyond a certain maximum. That led us to the second key insight.

The effort-time tradeoff works

It is common sense that you can trade off effort and time--person-months against calendar time. If you assign more staff to a project--put in more person-months per calendar month, you can finish the project in fewer calendar months. Contrariwise, if you assign fewer staff to a project, you expect that it will take them more calendar months.

To a rough measure, that is true. However, there are two special factors at play in software development:

1. Your ability to reduce calendar time is strictly limited by the minimum development time, no matter how many people you assign to the project. So, one end of the tradeoff is fixed for a project of given characteristics.
2. Then, above the minimum development time, your ability to trade off calendar time and person-months is limited by the nature of the relationship between schedule time and effort.

The relationship was derived empirically from data on hundreds of projects and since confirmed by data from thousands of projects. That relationship is that the time you allow is a far heavier factor than the amount of effort you sanction. The practical effect of this lopsided relationship is that, if you allow more time, you can substantially reduce the amount of effort. You have to plan this time allowance in the planning period--adding time after you get in trouble is too late! Conversely, if you insist on allowing only a development time close to the minimum, you greatly increase the effort (and hence the cost) that you have to bear.

One effect of this relationship is that a relatively small staff turns out to be more effective—much more economical, better product--than the large staff that you are tempted to employ trying to hit the minimum development time.

Change one core metric, the others change

Let's haul out that old common sense again. What is the relationship in any kind of work? Well, it is something like this:

Our *Work* product *(*at some level of *Quality)*

is

the outcome of *Effort* applied over a *Time* interval at a *Productivity* level,

Let's try some thought experiments.

If we reduce the *Work* product, perhaps by dropping some features, but keep the same *Process Productivity,* then the *time and effort* will decrease.

Or, we could trade off *Effort* or *Time* in some combination. If we use a smaller team it takes a longer *time,* but we spend less *effort* and the *quality* is better. This is a very desirable way to proceed because a little *time* elongation permits a lot of *effort* and cost reduction

A common attempt in these days of Internet time is to arbitrarily reduce *Time*. Proponents of the new approach argue that they have to get the dot com site up and running before competitors garner all the customers. A good argument, certainly, but what happens? *Productivity* is fixed, at least in the short run. *Staff (*related to *effort)* may be pretty much fixed because the supply of capable people is limited. You either have to be satisfied with fewer features in the *Work* product or with lower *Quality*. Right now *quality* seems to what is being sacrificed.

There is one way out of this box and we take it up next.

Improving process productivity pays off

If you can improve the *Productivity* of your process, then you could reduce *Time* and *Effort* by a corresponding amount, and improve the *Quality* at the same time. As you can see, looking at the relationship set forth in the previous section, better *Productivity* means that you can reduce *Time* and *Effort*, or you can get more *Work* product at better *Quality*. Various combinations of these factors are also possible. Better process *productivity* is well worth undertaking, because it is the only way to achieve the *Better, Faster, and Cheaper* goals that have become a common watchword in recent times.

If it is such a good idea, you object, why isn't everybody doing it? Good point! There must be a Catch 22. Yes, sad to say, there are--several of them:

- Catch 1: Process productivity improves rather slowly on the time scale of project execution. Our record of core metrics on more than 5000 projects over a span of more than 20 years reveals that process productivity increases about eight percent per year for business systems, several percentage points less for engineering and real-time systems.
- Catch 2. Keeping productivity gains going up takes skillful management and sometimes that is lacking. It's good to have increasing productivity for the long-

run health of the organization, but it can't be counted on for much help in the short run of a single project.

- Catch 3. Keeping your process productivity improving is not a task for the "in again, out again" boys amongst us. It is task for the long pull.
- Catch 4. Sadly, process doesn't, by itself, solve difficult problems. They are solved by good people, who work intelligently, within adequate time spans, aided by effective tools and stay with it.

Whatever you choose to do to improve your development process, you have to keep everlastingly at it. Persistence in the face of the incessant change that besets our society is not easy to come by. There is a temptation to chase the latest miracle-promising fad. The metric, process productivity, that validates your continuing progress, helps you keep your eye on what is important. “A little plus every day” works.

Sizing changes with all new development methods

Planners used to characterize the functionality of new projects in terms of size, expressed in source lines of code, the traditional measure of size. They could compare the new project with a database of completed projects, also expressed in SLOC. Function Points came along and gave us a new way to think about how we quantified what we were creating.

With more components becoming available from vendors and from in-house reuse groups, size is more and more being made up of a number of different entities representing the function. Some percentage of the functionality is coming into the project ready-made as reusable components.

We might well expect that reusing components would require less time and effort than building them from scratch. How much less is open to question. At one extreme, for instance, where interfaces between components have been firmly established and the components themselves are known to be suitable, reliable, and not likely to change, their employment should take little time and effort. Planners could allow only a modicum of administrative time to acquire these components and another modicum of development time to integrate them into the system.

At the other extreme, interfaces between components are still loosely defined and not completed, component suitability remains to be established, reliability is suspect, and the vendor has a history of releasing upgrades every year or so. In these uncertain circumstances, the amount of time and effort required to reuse a component may be as great as that required to build and maintain a new component. It might even be greater.

Between the extremes, however, lies a great unknown area. Just where in this continuum is a project manager to place the time and effort it will take to acquire, integrate, and test a particular component? Components might take either essentially no time and effort or allowances of time and effort comparable to that of building anew. A fine kettle of fish!

The question arises: To what is the work to be accomplished on such projects proportional? There is still a project. Its time and effort still has to be estimated. The progress of the work still has to be controlled against that estimate.

The general answer is that the amount of work is proportional to something in your own past project experience. An example is a project consisting of an audit followed by a forecast that we performed for a client. The client called the project “system integration”. In an audit-forecast, the project is already at least one quarter of the way to completion. From the data on the portion completed we can compute the process productivity and use it in forecasting the time and effort to completion. To make this forecast, we need an estimate of the remaining work. Frequently, we use an estimate of its size, usually in SLOC, but other countable function entities are satisfactory as well.

However, in this case, we could not estimate the remaining work in this way because the system integrator was writing little or no application code. It was hooking together a specially tailored Microsoft Outlook, Lotus Notes, and some other commercial systems to make a messaging system. Our assignment was to determine when the current release and the next two releases would be ready to field.

What our client did have was a history of the number of requirements that had been levied for each past release and the number that had been specified for each of the three forthcoming releases. We found that the number of requirements on the past releases correlated well with schedule, defects, test cases, and a crude measure of effort on those releases. So we used number of requirements as the "size" metric; that is, the metric to represent the amount of functionality remaining to be accomplished.

The lessons of this experience are two:

- As new development practices come into use, the ways of assigning a metric to represent the functionality of the product may have to become a measure not previously used.
- Whatever this metric, it must also be available in an adequate selection of past projects. It is the metrics attached to past projects that provide the basis for calibration--and calibration is vital.

There is no silver bullet

Fred Brooks was right in 1985 when he predicted that software developers would find no single silver bullet in the next decade. In fact, 10 years later he followed up and decided that his prediction had, indeed, worked out. [i] Now we are half a decade further along and, while some bullets have charmed us, none has been silver.

There is no short cut to successful software development. Sure, it is important to develop software as rapidly as possible (but it cannot be developed more rapidly than is possible). Sure, it is important to minimize effort and the corresponding cost (but they cannot be

reduced by wishful thinking). The one variable that can shorten schedules and reduce effort is process productivity. But it is not a silver bullet. It is hard, intelligent work.

[i] Frederick P. Brooks, Jr. *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition*, Addison-Wesley, Reading, MA, 1995, 322 pp.