Familiar Metric Management: Know What You Have To Do

In the December column we recounted that Leonardo da Vinci, aware of the invincibility of *feasibility*, recommended that the Canal du Midi across southern France not be attempted. The water supply to operate the locks was not then in sight. So Leonardo is remembered as a great painter, not forgotten as a failed canal builder. From his example we drew the lesson that works of software, like canals, should have a feasibility phase.

A century and a half later Pierre Paul Riquet, tax collector for the King of France for the region around Toulouse and the possessor, in consequence, of a mental map of the area, had the insight that the Montagne Noire north of the proposed canal route, not only had abundant rainfall, but also was at a higher altitude than the high point of the canal. That insight, in effect, concluded the feasibility phase.

Riquet embarked upon the second phase, *functional design*. His first step was to enlist the knowledge of Pierre Campmas, the *fontainier* of the mountainous area, the official responsible for water supplies. It would take us too far from modern-day software projects to review Riquet's many difficulties in planning and building the canal over the next 20 years. Let us just say that the initial cost estimate was nine million livres and the final cost was over 15 million livres. Sounds familiar, doesn't it?

It is the central business of the second phase of software development to reduce divergences of this kind to the levels that business practice can tolerate.

Metrics underlie software development

The most important metrics in the field of software development are measures of factors that, when projected into the future, enable a project organization to complete a software system within a time and budget at the desired reliability. It follows then that software people have to work with two elements. The first is the measurements that will quantify the factors. The second is the reality that the factors describe.

The key measurements. In this series of columns we have been identifying the minimum set of measurements needed to manage a software project: development time, effort, and size (or functionality) of the eventual software. From these three metrics of past projects we derive, via the software equation, the process productivity that will be applicable to the next project. Also, from these metrics, another equation projects the defect rate.

Now, considering the next project, we have the task of estimating development time, effort, and defect rate. We already know the process productivity metric from past projects. We need to establish at this point in the estimating operation the size of the proposed software product. It might be measured in source lines of code, function points, subsystems, modules, or even objects. All are, fundamentally, an expression of size. In whatever terms we measure this "size," we have to know first what it is that we are going to develop. It is that which we are to "size."

What it is

What we have to produce is described at this stage by the functional design. Some call it the architecture. We

have to carry it far enough to establish two points:

- What is to be done in sufficient detail to make an overall estimate of *size.*
- How it is to be done with sufficient clarity to identify potential *risks.*

Size. In this issue of ITMS our emphasis is on "what is to be done." We treated the sizing process itself in ITMS, December 1995, as well as in Chapter 4 of our book, *Measures for Excellence: Reliable Software On Time, Within Budget.* Sizing should be done by highly experienced people, but the precision with which they can size depends on the level of maturity of the design. At the end of the feasibility phase, for example, the feasibility team has carried the design only far enough to eliminate the critical risks. At this point the estimators can make only a "ballpark" estimate of time, effort, and reliability. By the end of the functional-design phase, the estimators have to make a more precise estimate, one with which their organization can live through the main-build phase.

Risks. In the feasibility phase we identified and avoided in one way or another risks that would make the project infeasible. Now in the functional-design phase we look for risks significant enough to make planning the construction phase chancy. By the end of this phase, we want to have a plan that we can execute within estimates of time, effort, and reliability. To reach this planning level, we need to work through these risks far enough to know how we are going to do these parts of the system, how much time and effort this doing will take, and what level of reliability we can expect.

Working through this level of risk often means that we have to firm up the requirements for that part of the system, carry out a tentative design, implement it in code, test that code, and show it to the stakeholders concerned with that area. This design and code is not usually what we will eventually deliver. Rather, it is a bare-bones code that demonstrates that the functionality in question works. It lacks user protection, error checking, and other niceties customarily found in the completed commercial product. This early prototype code may be only 10 to 15 percent of the final code. It does establish that we can do the risky function and that it will take about so much time and effort to do so.

Requirements change. It may turn out, however, that we cannot overcome a serious risk, at least not within a reasonable time-and-effort scope. In such cases, we may, in concert with the client, water down the requirement that is causing the risk. Furthermore, in the course of exploring the risks and setting up a functional design, the potential users and the design team almost inevitably will unearth problems with the proposed system that will lead to requirements changes. In fact, solidifying the requirements is one of the underlying purposes of the functional-design phase.

Why we neglect functional design

A functional-design phase seems to make sense. How do you know how much time and effort the main build will take if you don't figure out, at least approximately, what you have to do? Yet many big projects seem to slide over this phase. Capers Jones reported that 25 percent of projects in the lower end of the large-project range (100K to 500K SLOC) are cancelled. At the upper end of the range (greater than 1,000K SLOC), 65 percent are cancelled. [fn]Capers Jones, *Assessment and Control of Software Risks,* Prentice Hall, 1993.[/fn] Among the root causes he identified are:

- "Project was poorly planned."
- "Project could not achieve technical goals, such as critical performance or throughput targets."
- "Discovered during testing that it probably would not work well enough to be released."

“Poorly planned” implies, among other possibilities, that developers had to plan the project before they had a satisfactory functional design. The latter two root causes suggest that the developers did not have the opportunity to identify and reduce important risks before they were pushed into the main build.

Jones’ statistics take on flesh and blood in the experience of Robert Mittelstaedt, member of four boards of directors, who told Computerworld (Dec. 15, 1997): “I am amazed at the number of times I am called in to look at a system that has been in development for five years and is millions of dollars in the hole and still isn’t going to work, and they are getting ready to shut it down.” Why, then, is neglect of this phase widespread? The answer seems to be threefold.

- *Time*. This phase takes time up front. In our observation, the minimum time length of the functional-design phase is about one third of the main-build time. Clients, customers, users, and even many managers get nervous when all they see are a few guys and gals sitting around, supposedly thinking. Seeing some code coming out calms their nerves and in the customary waterfall development process that doesn’t happen until the main build gets under way. Hence the rush to get to the main build.
- *Effort and cost.* This phase also takes effort up front. Effort for large projects is about onefifth of the main-build effort. Who is going to pick up this cost? The answer varies with the nature of the application. In the case of a contractor preparing a bid, he stands the cost. It is his overhead and he wants to minimize it. In the case of an in-house software organization, the cost may be part of its own overhead structure or it may be financed by the in-house client. In either case the cost bearer wants to minimize it. In the case of a shrink-wrap organization, it is part of the ongoing cost of development and the organization may be under less pressure to minimize it.
- *Lack of appreciation.* Stakeholders fail to appreciate that getting risks out of the way up front and getting a firm basis for planning and cost estimation lead to a smoother main build, not to say few or no project cancellations.

Man of action

What would Pierre Paul Riquet do if he were in software today? We believe he would:

- Appreciate the merits of phased development.
- Modify financial arrangements to support the cost of it. For example, a client might support functional design with a time-and-materials purchase order.
- Produce some prototype code during functional design, demonstrating that a difficult part of the architecture can function or that a risk can be overcome. (And, not so incidentally, soothe participants’ ulcers. Riquet had the long-famed Sun King—Louis XIV—to soothe.)