

IT METRICS STRATEGIES

Helping Management Measure Software and Processes and their Business Value



Sizing Up Your Promises and Expectations

by Michael Mah

In previous issues of *ITMS*, I tackled the subject of managing Internet-speed deadlines (see *ITMS*, May and June 2000). Specifically, what do you do when IT projects are given a deadline first, before you know the full scope of the requirements? Also, how do IT applications development and maintenance projects fundamentally behave — what are the “laws of nature”? How can that knowledge help us estimate projects better?

Well, since then, I have continued to take audience surveys about current practices. I have conducted these queries at several recent speaking engagements.

I ask, “How many of you are given the deadline first? How many of you have at least one-half, one-third, or one-quarter of the requirements well established when you’re tasked with starting the build phase of your project?” In nearly all the cases, the answers are frighteningly the same: a fixed deadline, for what seems like an impossibly short schedule, and an undefined scope.

The Problem with Overpromises

The problem with this scenario is that project teams invariably slide into a dynamic where the deadline is firmly solidified, but the promised scope becomes wildly excessive, resulting in disaster. Without reliable

Continued on page 2.

executive summary

This issue continues to offer practical advice on how to use metrics to make better decisions.

First, I continue to bang away at the topic of managing risk in my article “Sizing Up Your Promises and Expectations.” When we examine IT project overrun statistics, we often see that a significant percentage of projects that manage to make both cost and schedule do so by the only method known to mankind when the deadline looms like a freight train — they cut function to the core. I take a look at how people get into that sordid mess in the first place. It begins with the initial overpromise, either through tacit allowance of scope growth or by blind over commitment on what the team takes on at the onset. In this space, we have to talk about the issue of “sizing” software IT projects.

Next, I’m pleased to introduce a new author to *ITMS*: Arlene Minkiewicz, chief scientist at Price Systems LLC. Her article, “Practical Software Measures Offer Solutions to Process Challenges,” talks about the priorities of successful measurement programs. She articulates how important it is to understand the problems that you’re trying to solve and to let that guide you in deciding what to do first. This is about aligning your measurement program to get the most out of your efforts.

Then, Stan Rifkin of Master Systems gives great advice in “How to Select Software Project Macro-Estimation Tools.” He offers his personal checklist that can help you get to the heart of the matter when looking for automated models to help with that all-important “scenario analysis” during decision time on critical IT projects.

All in all, this trio of topics is designed to help you get down to measuring what matters most and to use metrics in a meaningful way.

Michael C. Mah, Editor

september 2000 vol. VI, no. 9

Sizing Up Your Promises and Expectations.	1
Practical Software Measures Offer Solutions to Process Challenges.	1
How to Select Software Project Macro-Estimation Tools.	13



Practical Software Measures Offer Solutions to Process Challenges

by Arlene Minkiewicz

A good measurement program is one of the cornerstones of any successful process-improvement program. In fact, measurement is essential if you want to be able to quantitatively identify improvements. But knowing this doesn’t help when you find yourself responsible for the implementation of a measurement program in your organization. There are just a few questions you want answered. Where do you start? What are the right things to measure? Why me?

Successful measurement programs generally don’t try to accomplish too much at one time. It is important to understand the problems you are trying to solve with your measurement program and select the measures that best meet your goals. It is best not to try to solve all of

Continued on page 9.

Continued from page 1.

software-estimation capabilities, organizations badly overpromise. It seems to be in our very nature. This seems to happen with both inhouse development and in many situations where IT is outsourced.

Even if the initial promise is realistic, there's another problem — scope growth, or requirements creep. Along these lines, another question in my audience survey is, "How many of you know when to say 'no' as the requirements and scope grow on your projects? For example, when do you know that the number of work requests has crossed the line into an 'impossible zone' for a project with a fixed deadline?" Very few hands go up.

I contend that as long as people struggle with these issues, we will see an elongation of project schedules. To staff up on a project with a given deadline, overpromise on what will be done, suffer the inevitable slips, panic about the end date, and then pull functionality out at the last minute (to the "core") to make the schedule, destroys process productivity. That path costs more, takes longer, and produces more defects. In some cases, it's not uncommon for a 10-month schedule to become 15 months, and for the effort and costs to double as staff is ramped up during the chaos of the final months.

Our Competitiveness in the Marketplace Demands a New Core Competency

What is an IT manager to do? The situation of an IT applications backlog combined with the need to accelerate schedules creates a doubly painful situation where the reflexive response is to overpromise.

For many organizations, this dynamic is so pervasive that it is by far the number one killer of process productivity, resulting in projects being late and over budget, with poor quality. There comes a point for some

where the effects are so extreme that to focus attention on other issues affecting IT productivity is to pursue an abstraction. Other issues affecting productivity become almost irrelevant!

In fact, I can recall one project that my team benchmarked several years ago. The project size was more than 4 million lines of code. Productivity was phenomenal and reflective of the technology that was employed by the team. Ironically, this actually contributed to the problem of scope growth and churn. Why? Because management felt that it had a free ticket to do so because of how efficiently the code was able to be produced.

In the end, the project failed because the requirements grew wildly out of control. Consortium members fought over scope and lost sight of common interests, pulling the project in five different directions. The project crashed and burned, and more than US \$100 million was written off in losses. Management never deployed the system.

The size of the project was grossly underestimated at first and then ballooned into a monster. Ed Yourdon summed it up well: "If you underestimate the size of your project, common sense says that it doesn't matter which methodology you use, what tools you buy, or even what programmers you assign to the job."

That demands a new core competency for many IT organizations: managing scope; managing your promises; your commitments; the growth of the promise; the churn; the size of the amount of software that you commit to deliver. The rapid change in the marketplace demands that you handle these dynamics well. If you don't, you can be assured of bad outcomes. If you become great at handling this, you'll have a critical edge, and your productivity and creativity will show.

Editorial Office: Clocktower Business Park, 75 South Church Street, Suite 600, Pittsfield, MA 01201. Tel: +1 413 499 0988; Fax: +1 413 447 7322; E-mail: michaelm@qsm.com.

Circulation Office: *IT Metrics Strategies*® is published 12 times a year by Cutter Information Corp., 37 Broadway, Suite 1, Arlington, MA 02474-5552, USA. For information, contact Megan Niels, Tel: +1 781 641 5118 or, within North America, +1 800 964 5118, Fax: +1 781 648 1950 or, within North America, +1 800 888 1816, E-mail: info@cutter.com, Web site: www.cutter.com/consortium/.

Editor: Michael Mah. Publisher: Karen Fine Coburn. Group Publisher: Anne Mullaney, Tel: +1 781 641 5101, E-mail: amullaney@cutter.com. Production Editor: Lori Goldstein, +1 781 641 5112. Subscriptions: \$485 per year; \$545 outside North America. ©2000 by Cutter Information Corp. ISSN 1080-8647. All rights reserved. Unauthorized reproduction in any form, including photocopying, faxing, and image scanning, is against the law. Reprints, bulk purchases, past issues, and multiple subscription and site license rates are available on request.

Appoint a Chief Memory Officer

In a recent article on creating what he calls “dynamic stability,” Professor Eric Abrahamson of Columbia University’s School of Business urges organizations (not just IT) to appoint a chief memory officer. He states, “Only by remembering the past ... can we avoid making the same old mistakes — and take advantage of valuable opportunities. By contrast, companies that forget the past are condemned to relive it.”

Using the recent success of Apple Computer as an example, he goes on to say, “Apple has successfully resurrected the past with the iMac ... recreated the culture that challenges employees to build something ‘insanely great.’ ... There is nothing essentially new in this mix — it’s just that the company had forgotten its past as it ran through four CEOs in as many years. The return of Steve Jobs made the renaissance possible. He not only provides charismatic leadership, he also serves as the organization’s memory. Every company needs memory keepers *with the clout to make themselves heard* [italics added]. They help an organization undertake change without engendering unnecessary chaos, cynicism, and burnout.”

You will need a chief memory officer to successfully manage scope growth and requirements churn within IT. And he or she can be successful at solving this issue with a little knowledge of a few size metrics.

You’ll Need to Go into Your Own Project History

To be successful, you will have to go into your own history. On a few of your recent projects, maybe not a lot, it will be vital to catalog what your teams were able to pull off with regard to project size, given the amount of time and effort that was expended. How many people were working for how long, and what did they produce? Start with the Carnegie Mellon Software Engineering Institute’s (SEI) Core Metrics. (See my article, “Meditations on Which Metrics Matter,” in the February 2000 issue of *ITMS* for a discussion of these metrics.)

You won’t necessarily have to dig deep into the bowels of your cost-accounting systems to get this information. On benchmark engagements, I usually conduct an interview

with a project team to sketch out a staffing profile of the number of people over the different phases of the project. It’s usually quite an accurate representation of the effort that was involved. An example of this staffing sketch is shown in Figure 1.

Once you draw this out, you can eyeball the elapsed time for each major phase. For example, the time for the Functional Design (FD) and the Main Build (MB) phases shown in Figure 1 are 7 months and 11 months, respectively. (In the figure, FS stands for Feasibility Study.)

The effort is captured by adding up the full-time equivalent (FTE) head count over the period for each phase. So for the Functional Design and Main Build phases shown in Figure 1, the effort was 39 person-months and 101 person-months respectively.

Next comes a quantification of the “output” of the team.

In some cases, where no coding is involved, the work products can be defined in “non-software” units of output. It might be something like the number of master processes and subprocesses deployed for an enterprise resource planning application, for example. Time and effort would be accounted for in the same manner as when software development is involved. Units for time are elapsed weeks or months, with effort being in person-months.

In IT projects where applications development and maintenance is involved, the products might be the number of new, modified, and reused software modules; the number of simple, moderate, and complex programs; the volume of customer project requests; the number of work requests; how many use cases; the number of end-user requirements; the quantity of business processes; the number of function points; Java scripts; C++ Objects; Powerbuilder objects; or COBOL code.

For the project described earlier, the team was able to quantify the programs and their respective sizes for each release by counting the code in each program. The amount of modified functionality was summed with the amount of new functionality. The combined total represented the amount that the team

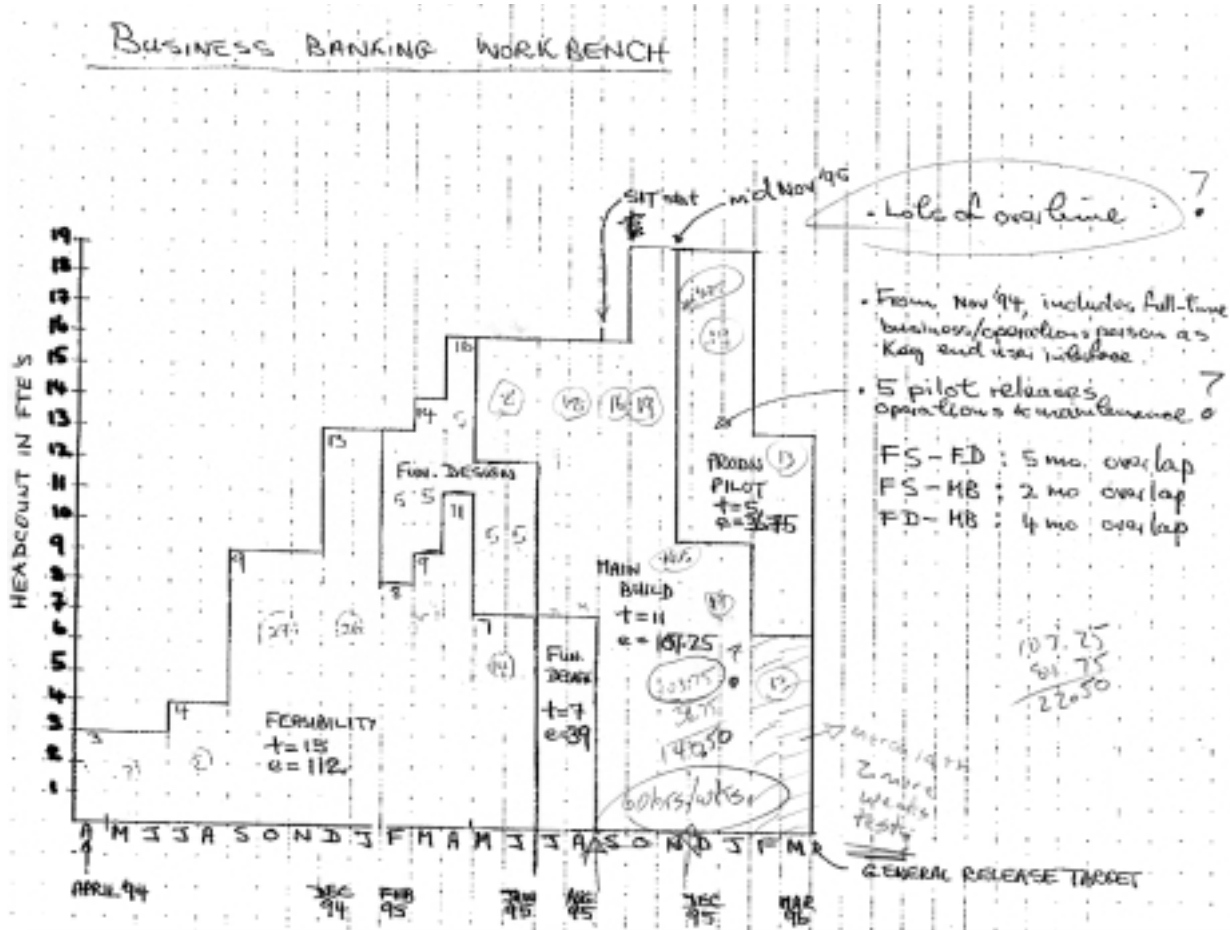


Figure 1 — Staffing sketch: gathering time and effort metrics.

worked on during each release. An example of this is shown in Figure 2.

At this point it is useful to note that some categorization is in order. You'll need to account for different "buckets of functionality." One such "bucket" is new software built from scratch. Another is existing software that is changed or modified. A third is existing software that is reused "as-is" without any changes. A fourth variant on this category are software applications that are purchased off the shelf (packages). This last category can be considered "reused" software, but that which is acquired as opposed to software that an organization had previously designed, coded, and tested itself. Keeping this level of detail is important to understand the component makeup of your releases to understand their productivity attributes.

An example of conceptualizing this breakdown is shown in Figure 3 (on page 6).

Lines of Code and Function Points: Why Both Metrics Have Been Tainted

Two common metrics for sizing the volume of software in IT projects are source lines of code (SLOC) and function points. In my opinion, the debate over which measure is superior has left both tainted by overzealous industry "experts" in the process. The net result is that many organizations waste time in energy-sapping debates. Organizations that get trapped in this situation often experience a semi-paralysis, with people in their metrics programs arguing more and doing less.

The Carnegie Mellon SEI advocates the use of SLOC and, specifically, logical source statements (LSS) that strip out comments and blanks. The benefit of SLOC or LSS is that their counts are repeatable and can be automated. You won't have a situation where an individual's perceptions or interpretations of counting rules would result

ALRETS - Release 1

Program Name	Lines of code	Lines of code in base pgm.
SLN Parameter		
L4RNEDRP	644	613
L4RNUPD1	1201	1056
L4RPD40	687	648
L4RPL10	1156	1628
L4RPR10	1323	1848
L4RPU10	1909	1258
L8RPS1BC	2193	1987
L8RPS2BC	1395	1099
L8RPWTRC	812	659
Arrangement Parameter		
L4RPL12	1819	1628
L4RPR12	2104	1848
L4RPR12G	807	546
L4RPU12C	1608	1258
L8RPN1BC	2095	1987
L8RPN2BC	1171	1099
Translation Parameter		
L4RPL14	1178	1628
L4RPR14	1432	1848
L4RPU14	1858	1258
L8RPT1BC	2238	1987
New lines of code:	0	
Modified lines of code:	27,488	Base lines of code: 25,883
Total	27,488	
minus 10% for comments:	24,739	23,294

Handwritten notes:
 - "Low 40% 50% 100% Tested"
 - "20% modified"
 - "R1+R2"
 - "11.2"
 - "3,468 - 1,463 = 2,005"
 - "1605 - 8126"
 - "1987 19636 890 26762"
 - "12.1" (circled)

ALRETS - Release 2

Program Name	Lines of code	Lines of code in base pgm.
L7RACINT	1714	N/A
L7RBDAYS	2082	2082
L7RBHIER	254	275
L7RBKOUT	1903	N/A
L7REDIT	2802	N/A
L7RFCTCK	1101	N/A
L7RFCTL	1904	1711
L7RHIST	1579	N/A
L7RINIT	287	N/A
L7RLOAD	284	N/A
L7RMVF	707	N/A
L7RMVFC	2057	N/A
L7RRETS	829	707
L7RRETS	1622	2057
L7RSHIFT	747	N/A
L7RUPDTE	2187	N/A
L7RUPDTI	2115	N/A
L7RUPDTU	518	N/A
L7RXRPTS	1994	1994
New lines of code:	17,911	
Modified lines of code:	8,685	Base lines of code: 8,826
Total	26,596	
minus 10% for comments	23,936	7,834

Handwritten notes:
 - "12.1" (circled)
 - "17917 678W on 26 U"
 - "17917 6354 ch mod 21374"
 - "16 120 new 315 first"
 - "1599 chg"

Figure 2 — Project size for three releases (sample).

in different counts as might be the case with function points.

There are other organizations (perhaps with a financial or philosophical interest in function points) that argue strongly in their favor. In some cases, function points are a good fit if we're talking about an IBM COBOL

mainframe application with a Create Read Update Delete paradigm against an underlying database.

However, there are arguments made that, for modern-day Web or client-server development, function points are a round peg in a square hole. Some system architectures

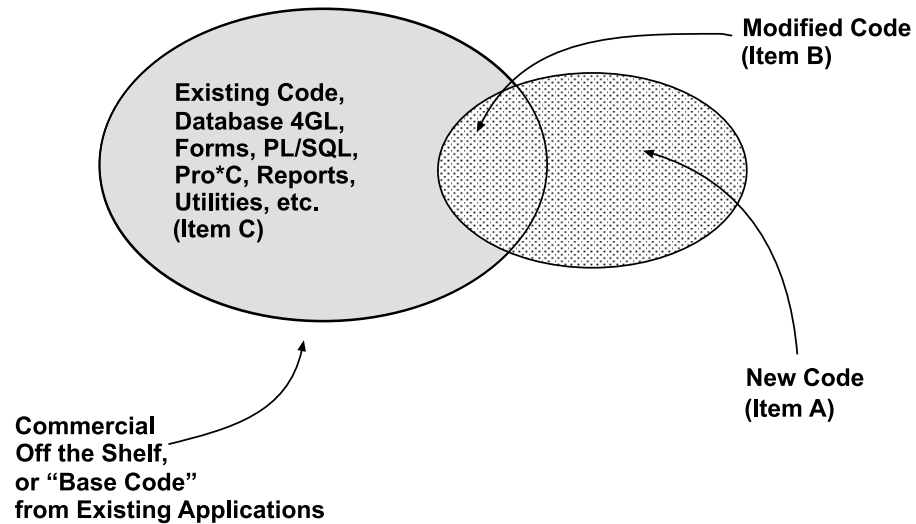


Figure 3 — Diagrammatic view of new code, modified code, reused code (items A, B, C).

simply might not be compatible with the function-point view of the world. Moreover, function points are labor-intensive to count; you don't have the equivalent of source counters and configuration management library reports to give you size statistics.

So, in some cases the code is entirely relevant. After all, what happens after the detailed design phase for applications development and maintenance projects? People code. (They do not "function point.") When there's a bug, the bug is in the code. You can't get away from code if you are in the business of creating software that runs on a computer. Arguing that code metrics are irrelevant is like saying transistors are irrelevant to microprocessors or memory chips.

The truth is that *both* are useful. And while neither is a panacea to fully address the notion of output in terms of functional size, they work, and that can be good enough.

This discussion is bound to stir up debate. Nevertheless, what is important is that you do *something*. You must quantify the amount of the "building blocks" of a software application. If you can, count both code and function points. What that will do is enable you to understand the relationship between these entities, like fluid ounces in a gallon of water. Gallons, like function points, are a measure of quantity at a certain level of abstraction. It takes 128 fluid ounces to a gallon just like it might take 90 or so lines of code in COBOL to make a function point in your environment.

Typical proportions between the amount of code that it takes to create a function point are listed in Table 1. As you can see, more powerful programming languages generally require less code to produce a function point. This table is a useful start to understanding the scaling relationships between the two. In fact, if you count both code and function points in your applications, the table will enable you to cross-check the proportions that you're seeing in your environment.

But this is not perfect. As described earlier, certain categories of applications like Web-based applications or algorithm-intensive programs do not fit the underlying Create Read Update Delete architecture that defines function point counting. In those cases, function point counts will be inappropriate, so the scaling relationships will be moot. You may need to count both the code and perhaps other higher levels of abstraction, like objects, classes, modules, or simple, moderate, and complex programs.

Remember that no one size measure is perfect. But be careful not to throw the baby out with the bath water. Be cautious of broad, sweeping generalizations. I recently heard of a CIO for a major corporation who threatened to not sign a major outsourcing deal if it had function points written into the business agreement. Five years ago, the then-CIO of the same corporation swore against code metrics. Both CIOs had reasons for their views. But in some ways they've been duped. A unit of measure is

not evil in and of itself. It might simply be a case of misuse or misappropriation.

Your History Will Help You Better Manage the Future

Once you have a sense of what your teams were capable of producing in the past, it will enable you to have meaningful data to know what you can realistically promise in the future. "But what we're doing now is unprecedented!" you argue. That may be true. So what? Knowing your past will still enable you to have a starting point for what you can do in the future, even if some assumptions and extrapolations will have to be made. Your past projects were just as "unprecedented" back when they started. A little risk analysis never hurt anyone — just don't extrapolate your neck so far that your head gets cut off.

The point is, not having any knowledge of your recent IT capacity will most definitely leave you blind. And if you've suffered from stepping on land mines in the past, you'll need some self-knowledge about what you did before so you avoid making the same mistakes.

Furthermore, organizations that document their recent output will be well positioned to create the productivity baselines I outlined in previous *ITMS* articles. These describe how to benchmark your own productivity without needing a consultant. The basic idea is to individually plot the schedule, effort, and defect performance of your projects as a function of size and draw a line through the data to identify a trend. You'll be able to visually see the individual performance across small, medium, and large projects. (See the March and April 2000 issues of *ITMS*.)

Moreover, underlying patterns will emerge to reveal findings within your data for various categories of applications. Nuances (those critical differences) will be uncovered that help identify factors that influenced team performance. This is like striking gold. If you know the negative impacts of key factors that hurt you on recent projects, they can't hurt you anymore unless you let them. Conversely, when you uncover those magical elements that resulted in good outcomes, you'll discover how to create the same

Table 1 — Function Point and Code Scaling Relationships by Language

Language	Function Point Language Factor
Excel, QUATTRO PRO, Mosaic, Screen painters, spreadsheet languages	5
Database Query, PACBASE, SQL, PowerBuilder, TI-IEF, Program Generators, TELON, Notes, Nexpert, Java, Fusion, Delphi	15
ADS/Online, MUMPS, NETRON/CAP, SMALLTALK, Oracle Forms, ReportWriter, Easytrieve, SAP	20
APL, Objective C, Visual Basic	30
MS C++ V7, CLIPPER, FOXPRO, object-oriented default, statistical default, decision support default	35
Database Languages, AI Shells, FOCUS, ORACLE, RAMIS II, SYBASE	40
Informix, simulation default, C++	45
English Based Language, MAPPER, Natural, RPG III	55
Ada, QuickBasic, RPG II	60
BASIC, PROLOG, Object Assembly	65
PL/1, Ada 83, problem-oriented default, TALON, PLM	70
REX II, 3rd Generation Default	80
PASCAL, ANSI COBOL 85, Tandem (TAL)	90
ALGOL 68, CHILL, FORTRAN, UNIX shell scripts, JOVIAL	105
C Default	130
Macro Assembler, JCL	220
Basic Assembly, SPS	320
Machine Language	640

conditions and repeat your successes. Don't repeat failure.

The key insight here will be knowing the limits of what was accomplished in terms of work output from your own past. Specifically, when you understand the maximum amount of output that was produced for a given amount of time and effort, you'll know the upper limit of what you can promise in the future.

Herein lies the line in the sand. Knowing your organization's limits is the first step to not overpromising for your next project. If

you don't know these limits, you'll be like the people who "don't know when to say no," when scope growth dominates the dynamics on your IT project and causes them to suffer.

Separate Output Size from Complexity

When you embark on your project, you'll have to remember to not get hung up on prejudicing your data collection by entangling output with complexity. Most people resist getting started in this area by using arguments that show their paralysis around data collection. Specifically, people say things like, "But if I count the modules in this application, they're more complex than counting modules in that other application and they took longer to build. So a module is not a module is not a module. Let's not bother, since it would be a waste of time!"

Don't get stuck in this tarpit.

Separate size from complexity. More complex modules take more time and require more work-effort to build. Catalog them into simple, moderate, and complex modules. Keep the amount of time and effort that it took to build them as a separate issue. You'll find that a project with a certain composition or "pie distribution" of predominantly simple modules most likely took less time to finish and expended less effort. It was easier. Projects with more complex modules took longer and were harder to pull off, so they required more work-effort.

Good for you! This will help you know what the outcomes might likely be for future projects with varying degrees of difficulty. When you have a deadline for an IT project with lots of tough stuff, you can probably bang out more simple modules but fewer ugly, tough ones. This will help you draw the appropriate lines in the sand and not overpromise and get in big trouble with scope growth.

Summing It Up

As professionals, many of us in IT have to be acutely aware of our own emotional makeup that predisposes us to the slippages and overruns that characterize IT projects, whether we are managers, developers, or end users.

Many struggle with what I call the trap of the productivity paradigm. That, in order to feel competent and valuable, we have to produce more and more in less and less time. The problem with this is that in today's technology-driven economy, it is often never enough.

The backlogs simply don't go away. Moreover, we take on ever more complex projects and raise the high bar yet another notch. Also, our hero culture of rewarding individuals for their contributions often looks at software reuse as plagiarism, so we tend to create a lot of things from scratch. What is considered "goodwill" between developers and managers results in a lot of unofficial "under the table" work that is promised because we value the feelings from the internal relationships that are fostered by a "can-do" attitude.

The danger here is that we really don't know when to say no. And in our efforts to satisfy, we simply sign up to do too much. Perfectionistic tendencies to please the boss or the customer set up unrealistic expectations, and what actually happens in the end is that people are dissatisfied and burned out, feeling unappreciated in the process for all the personal sacrifices that have been made.

Getting the management numbers on what people have been capable of delivering is your first step at interrupting this cycle of lost productivity.

After acquiring this knowledge, some will continue risky behaviors in spite of what the data tells them. They may choose to overpromise again on the next big project. But this guarantees mediocrity — the very thing many of us "people pleasers" are trying to avoid. You do not have to choose this path.

If you consciously manage your promises and the expectations of what will be delivered, you can find a way out of this dilemma. This will involve some reflection on your part to proactively decide what you are willing to commit to on your next project. The remarkable payoff in managing this well is that teams can suddenly find themselves making decisions that reduce the chaos, manage requirements churn well, contain ballooning project scope, and actually improve productivity in the process.

References

Abrahamson, Eric. "Change Without Pain," *Harvard Business Review*. July-August 2000.

Mah, Michael. "Internet-Speed Deadline Management, Negotiating the Three-Headed Dragon," *IT Metrics Strategies*, Cutter Information Corp., May 2000.

Mah, Michael. "Software Estimation Tricks of the Trade: Secrets They Never Told Me,"

IT Metrics Strategies. Cutter Information Corp., June 2000.

Carleton, Anita, Robert Park, and Wolfhart Goethert. "The SEI Core Measures: Background Information and Recommendations for Use and Implementation." *The Journal of the Quality Assurance Institute*, July 1994.

Practical Software Measures

Continued from page 1.

your process problems at once, but rather develop an incremental plan that addresses one or two at a time. Measurement programs are often met with suspicion and resistance. It is best to start small, show some success, then build on that success rather than introducing an elaborate far-reaching program that is intimidating and more likely to fail.

The question still lingers: where do you start? I think the best place to start is with a thorough evaluation of your measurement needs. This should include:

- Identifying the areas where improvement is needed
- Prioritizing these needs against organizational improvement goals
- Identifying what measurements are best suited to meet those needs

The following sections should help with the last part of your task. Some of the more common problems that require measurement are outlined below along with some types of measurements, and suggested metrics, that will lead to solutions.

Requirements Management

If a big part of your organization's process challenges appear to revolve around the fact that requirements are poorly stated, poorly understood, and constantly changing, then maybe this is an area where you want to start your process-improvement efforts. In order to be able to manage any project successfully, you need to understand what you are building and what is not going to be built.

You need to be able to communicate this not only to the developers who implement the software but also to the customer who pays for it. If you are not doing this well now, you are discovering late in the project that the requirements have been incorrectly implemented or that the requirements that were correctly implemented do not actually meet the needs of your customer. The later in the project you discover this, the more it costs.

Measurement alone can't solve this problem, but it certainly will help. Only with measurement can you begin to get an idea of the extent of the problem and locate the point in your process where it starts. You may have a *qualitative* understanding of this but it is important to *quantify* it.

A good way to do this, if your organization keeps the requirements document up to date as requirements change, is to count and record at every milestone the number of pages or the number of "shalls" in your requirements document. If you find that this number changes substantially after the "final" requirements review, you have identified an important goal for your process-improvement program — reducing the number of changes to requirements after requirements have been finalized. If your organization is not doing a good job of keeping the requirements document up to date as requirements are added or changed, you should count the number of change requests that track to changes in requirements or to the addition of new requirements. This count should also be accumulated at each milestone.

This type of measure will help your organization to understand where in your process the requirements are changing and make it possible to quantify improvements as requirements are better understood up front and more controls are put on the change process.

Once you have assessed where you are, it is then time to make some changes. At a minimum these changes should include:

- A defined process for documenting, reviewing, and accepting software requirements
- A defined process for the review and documentation of changes or additions to the requirements once a requirements document has been approved

Both processes must include all affected groups in the organization. Include in your process some additional measurements that will make it easier to evaluate the process. In addition to tracking the number of changes *against* each project milestone, it is also important to start to track the number of changes and the status of each requirement *at* specific milestones.

This will enable project managers to identify “problem” requirements early so they can address breakdowns in the requirements process and take appropriate action. The measures for status of ongoing requirements should include the percentage complete for the current activity. Percent complete can be measured using percentage of estimated code complete, the percentage of estimated functionality designed, the percentage of test cases verified, or, if your organization is proficient at effort estimation, an earned value calculation of budgeted versus actual effort for each activity.

Cost Overruns

A very important part of good project management is the ability to estimate the cost or effort that is required to implement a system once you have established the requirements for that system. A good project manager is able to estimate the resources required to implement a particular software solution from the initial requirements document. Once this has been done, the project

manager has a solid foundation for managing and controlling the project based on this original estimate (updated sensibly as requirements change).

The crucial measurements for effective cost and effort estimation include:

- Estimated size
- Actual size
- Estimated amount of reuse
- Actual amount of reuse
- Quantification of the functional complexity of the application
- Estimated effort
- Actual effort

This is not to imply that these are the only measures important for managing project costs, but these are good ones to start with. There is enough information in these measures to get project managers to the point where their project performance is repeatable. As cost-estimating processes mature within an organization, there are many other measures that will raise the estimating process from one that works on a project level to one that extends value to the entire organization.

It is important to record both estimated and actual values because it is in the comparison of these two that much can be learned. By comparing actual results to estimates, and understanding why they differ, project managers can improve their ability to estimate future projects. The software size values are important because it is upon these that the effort estimates are based and against which the actual effort can be compared in order to determine productivity or unit cost for the software development organization.

Similarly, keeping track of how much of the code is reused allows for the separation of reused activities and new development that is necessary to understand how this project relates to others. The quantification of application functionality is just an indication of the complexity of the code that must be developed. In using historical data to predict future efforts, it is important to understand how the new project is similar to and differs

from historical projects with respect to code complexity.

The size of a software product is one of the hardest things to quantify. There are multitudes of metrics that can be used for software size. Certainly the most popular is the source lines of code (SLOC) value. This is a measure of the number of physical (real lines in the source code file) or logical lines (lines as defined by the programming language being used) in the finished product.

There are many ways to estimate SLOC from requirements, some available through commercial software-estimating products, others based on individual organizational histories. There are many problems associated with using SLOC as a size measure, particularly if you are using newer technologies such as object-oriented analysis and design, but the beauty in the SLOC measure is that the process of collecting actual values is simple and can be automated.

There are other size measures that have gained popularity recently, including function points (a measure of the functionality delivered to meet end-user business needs), number of objects, number of classes, number of shells in the requirements document, number of use cases, weighted methods per class, and many others. The selection of a size metric should be made carefully, and care should be taken to ensure that the rules for estimating and measuring actual size are well documented and adhered to because consistency is the most important factor when using past experiences to predict future successes. Measures of the amount of reuse should be made using the same metric used for new code.

Effort should be estimated in person-months or person-hours for each of the software development activities in your software life-cycle model. During the actual development, effort should be tracked against these same activities in person-months or person-hours as well. It is important that you have documented rules for what specific tasks fall into each of the activities and that these procedures are understood and followed by all the parties required to record their time. The more thoroughly everyone understands the rules for recording actual effort, the more

accurately future estimates will reflect actual organizational performance.

Finally, the measure for application functionality could be as simple as a low, medium, or high designation, or it could be a value between 1 and 10 determined using an algorithm that uses a weighting system to assign low values to simple functionality (like data I/O and simple math) and high values to complex functionality (like real time and interactive GUIs). Sometimes the introduction of a commercial software-estimating tool will help an organization develop standards for quantifying measures like functional complexity.

Schedule Overruns

Schedule overruns often go hand in hand with cost overruns. If you have yet to take control of the software cost-estimating processes, it is quite likely that the schedule estimation process is lacking maturity as well. However both are not equally important to all organizations. Some organizations will meet schedule regardless of the impact on cost while others will sacrifice schedule commitments to keep costs down. The processes and measurements that need to be in place to solve both cost and schedule overruns are fairly close. To get your schedule-estimating efforts off to a good start, the following quantities should be measured:

- Estimated size
- Actual size
- Estimated amount of reuse
- Actual amount of reuse
- Estimated duration
- Actual duration
- Quantification of the functional complexity of the application

Once again, these are not all the measures that will make you a good schedule estimator. This is the place to start to get a handle on how good you are at estimating schedules right now and to provide some data upon which to base future estimates.

As far as what should be measured, for size reuse and functional complexity, the information given for cost estimation applies. The duration should be estimated in calendar days for each development activity, keeping in mind that an important part of your process document is that which indicates which specific tasks are included in which activities. Actual duration should be tracked against the same activities.

Defect Prevention

Maybe the driving measurement goal for your organization is to assess and improve the quality of the products that you produce. It should be fairly clear how measurement can help you to *assess* product quality. What might be less clear is how you can use measurement to *improve* the quality of your products. The measures that will help you assess and improve product quality include:

- Actual size of each major component of the software system
- Number of defects per size unit in each component in the finished product
- Number of defects per size unit per severity category in each component
- Number of defects discovered at each project milestone
- Actual amount of time spent on peer reviews, code walkthroughs, etc.

Product quality assessment can be accomplished by maintaining a count of the number of defects discovered in the released version of the product and reporting this against the product size. If you keep track of size in SLOC, then your defects should be tracked in defects/SLOC or defects/KSLOC (1,000 lines of code). This count should be tracked for each major component in the system. In addition to counting the defects, it is also important to track severity of these defects. One way to do this is to rate defects on a scale from 1 to 4 with the following definitions:

1. **Cosmetic problem** — e.g., misspelling on UI or poor alignment of reports.
2. **Minor annoyance** — user is inconvenienced when accessing functionality,

but the problem can be easily worked around.

3. **Major annoyance** — access to functionality is limited or requires complicated work-around.
4. **Severe** — program crashes or major functionality is unavailable or works incorrectly.

Tracking severity is important to assessing product quality because low severity defects are often released intentionally to meet schedule demands. Understanding where on the spectrum your defects currently fall will also help you target your improvements to the areas where they are needed the most.

Once you have made an assessment of the quality of released products, it is time to start improving your quality and testing processes. One way to figure out where these improvements are most needed is to start figuring out where in the process defects are being injected and where they are being found. Not only do you need to record whether a defect was found in requirements documents, design documents, during unit test, component test, or system test, you also need to know during what activity this defect was found.

If you are finding that problems in requirements are not identified until the coding has started, you need to start to improve your requirements-review process. If coding problems are not found until component testing, you need to implement a process that focuses on peer reviews and/or walkthroughs. The goal is to find defects during the phase where they are being injected. A defect becomes progressively more expensive to correct as you move away from the phase where it occurred.

Another important measure to keep is the amount of time you spend reviewing your work. If you start tracking the amount of time spent reviewing requirements, design, code, etc., you will notice that as this number increases, the number of defects found at each phase will decrease. If this does not start to happen, then your measurement program has identified another problem area: your reviews are not effective, and it may be time to invest in some training in this area.

Conclusion

In software development there are many real-world problems with solutions that are facilitated by measuring the right things at the right time in the process. Process-improvement programs need to include measurement not only as the means for identifying the areas for improvement but also as the means to determine progress against improvement goals. If you are responsible for a measurement program or process-improvement program, the first thing you need to do is determine what problems need solving the most, and then start measuring.

The problems presented here are by far, not inclusive of all the challenges you are taking on with your improvement program. Chances are really good if you are just starting out with measurement and process improvement, that some of these problems are on your list. And the measurements and metrics presented here will not offer completely comprehensive solutions to these problems — they are presented as good starting points.

Once you get comfortable with the role of measurement in your processes, you will begin

to identify other measurements that will help these processes and your organization mature.

About the Author

Arlene Minkiewicz is the chief scientist at Price Systems LLC. In this role, she is responsible for the research and analysis necessary to keep the Price Estimating Suite responsive to current cost trends. She has more than 15 years of experience with Price, designing and implementing cost models. Prior to her current assignment, Ms. Minkiewicz functioned as the lead of the Product Enhancement team with responsibility for the maintenance and enhancement of all the Price products. She speaks frequently on software measurement and estimating and has published articles in *Software Development* magazine and the *British Software Review*. Ms. Minkiewicz has a BS in electrical engineering from Lehigh University and an MS in computer science from Drexel University. She is a member of the International Society of Parametric Analysts and International Function Point Users Group.



How to Select Software Project Macro-Estimation Tools

by Stan Rifkin

Over the years, I have developed a checklist for selecting automated tools to perform macro software project estimates. Mine is similar, though not identical, to Robert Park's *Checklists and Criteria for Evaluating the Cost and Schedule Estimating Capabilities of Software Organizations* (SEI-95-SR-005, January 1995).

There is a zeroth item, one before selecting and evaluating an estimation tool: establish your project commitment process and then the part within that that focuses on estimation. Park, formerly at the Software Engineering Institute (SEI), has written a number of SEI technical and special reports on the subject of estimation processes, and I recommend the reader to them. The reports are available for free on SEI's Web site (www.sei.cmu.edu/publications/) and are cited at the end of this article.

Park's Fig. 5-1 (shown here in Figure 4) in *Software Cost and Schedule Estimating: A Process Improvement Initiative* (SEI-94-SR-03, May 1994) is one of the most-cited examples of an overall software macro-estimating process.

Basically, for the estimation process to support a commitment process, the following has to happen:

- Commitments have to be based on the work to be performed; therefore, there must be agreement on this.
- Estimates have to be based on (a) the work to be performed and (b) historical records of performance.
- Commitments must not exceed the capability to perform or else there is no reason to estimate.

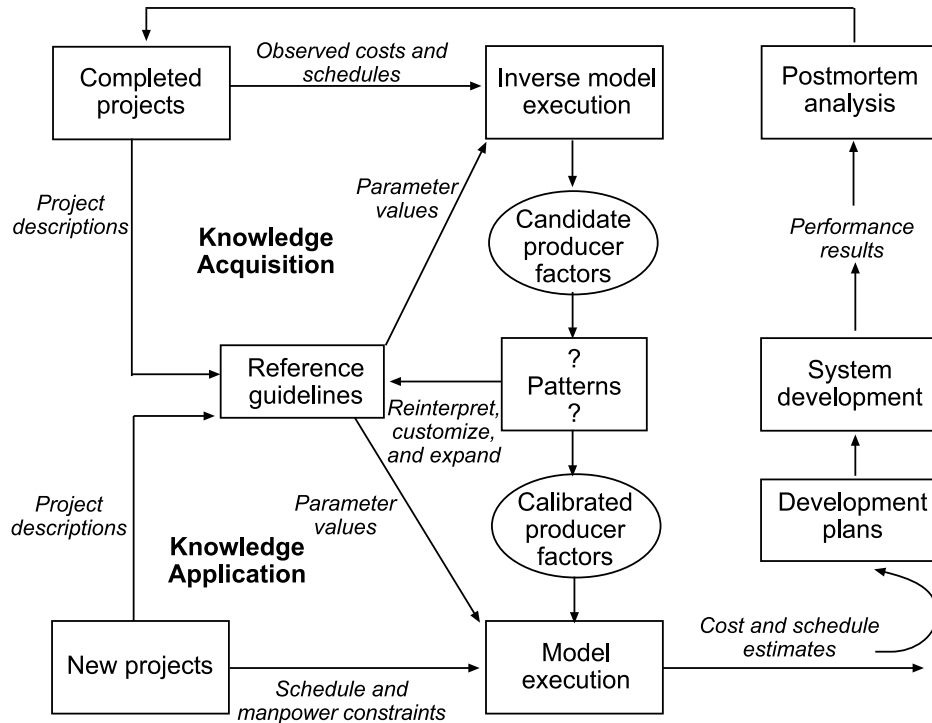


Figure 4 — Graphic template: parametric estimating. (Source: Software Engineering Institute)

Incidentally, these form the basis of the Capability Maturity Model (CMM), particularly the lower levels of software process maturity. CMM is about meeting commitments, and one of the key observations is that it is easier to meet rationally justified commitments via estimates based on reason and actual historical performance than to spend time creating stories about why we are late, over budget, etc.

With a commitment process and estimation subprocess in hand, and the admonition to select a tool that is consistent with both, here is my checklist of criteria, in priority order. The items at the top of the list dominate, so failing those top criteria can disqualify a candidate tool from further consideration.

Checklist for Tool Selection

The following items are included on my checklist of issues I look at when selecting tools.

1. The underlying algorithm is published in the public domain.

I am worried about surprises, so I want to read about the formulas used, assumptions made, constraints, etc. I am particularly

looking to see if $duration = effort/resources$. It is built into every project management software package (such as Microsoft Project), and it states that the time it takes to complete a project is the person-days required divided by the number of people. So, if I have a 10-person-day project and 2 people, it would take 5 days. If I had 10 people, it would take 1 day.

If this formula is in the tool, I would have to reject it, as it is well known that the relationship among duration, effort, and resources is not linear. It may be true in building construction that doubling the number of carpenters cuts their duration in half, but doubling the number of programmers probably stops the software project!

Tools such as SLIM, Price S, and all generations of COCOMO publish their formulas, so at least I am not going to be surprised!

2. It accurately estimates completed projects.

We use a few typical completed projects to see if the candidate tool estimates them accurately in retrospect. Clearly, if the tool cannot accurately estimate our type of

work, we do not want to use it on real, future projects.

The process of estimating completed projects, what Park in the diagram calls “inverse model execution,” leads right to the next criterion.

3. I don’t want to subjectively “guess” at the values of variables.

If the actual project performance does not equal the estimate then, naturally, I want to know where I went wrong. Some tools use scales to assess items such as team experience, by selecting a number between, say, 1 and 10. There may be text associated with each value, such as “new team,” “worked together for many years,” etc. I cannot measure team experience, I can only guess at its value. When actual does not equal estimate, I do not know which subjective estimates are off and by how much, so I eschew them. I only want to work with items that I can objectively measure during the conduct of a project. I would be especially disturbed by long lists of such subjective questions, as I would have no hope of tying my responses to actual project performance that I cannot measure.

4. The assumptions of the tool mirror my realities.

Estimates are application-area specific (see Chapter 1 in *Measures for Excellence: Reliable Software on Time, Within Budget* by Lawrence Putnam and Ware Myers, Prentice Hall, 1992), so we need to be sure that the candidate tool has been particularized for my application type (e.g., avionics, process control, business). It also has to be applicable to the size, programming language, lifecycle, and degree of centralization and formalization present in my projects. Often these aspects are not stated in estimation tools, so they are disqualified from consideration. One nameless tool uses a lifecycle that I do not, so I know that the estimates will not mirror my reality.

5. The tool will not generate an impossible schedule.

This was a common response to COCOMO and its progeny. Because it was a formula, it could compute a duration that was impossible in empirical terms. That is, it could compute a duration that had never before

been accomplished by any team anywhere. Clearly there is a region or range of project values (duration, effort, rate of adding people, quality, and number of features) such that it is impossible to accomplish a project with that combination. (See *Measures for Excellence*, Putnam and Myers, p. 95.) I want to know that set of values so that I am sure to steer clear. It is an added bonus if the tool tells me explicitly what the impossible region is so that I don’t have to spend time guessing, like playing Minesweeper!

6. The tool takes into account the effects of schedule compression.

We all know the quip about nine women making a baby in a month (this is a variant of $duration = effort/resources$). In software projects we are commonly asked to produce the minimum duration estimate. In other words, what is the effort required for the stated requirements such that the project will finish in the shortest time? Candidate tools that simply use a formula to compute the duration (such as the old COCOMO) might not take into account the nonlinear effect of schedule compression, that reducing the duration by 10% can increase the effort by 40%. Trying to reduce schedule by 20% is even worse. Michael Mah referred to this in previous issues of *ITMS* as the 200/20/6x rule, whereby doubling the effort (200%) results in only about a 20% schedule compression, but with a severe reliability penalty — a 6x rise in defects.

One can see, too, that the formula $duration = effort/resources$ has the effect of not modeling compression, which is yet one more reason to avoid tools that use it. One of the reasons I like tools that model compression is that we are commonly given the delivery date during the commitment process, so I need to be able to ask and answer whether I can stand (that is, manage) the compression in the dictated schedule.

7. I want a range, not a point, estimate and the probability of achieving it.

We all know that estimates are most useful if they give us a range and probability, such as “There is a 70% chance of slight rain, and a 50% chance of significant rain.” In order to intelligently make software project decisions we need the same thing. We need to know the probability or risk for each feasible band

of estimates. This way we can adjust our input parameters in order to compute not only an answer about duration, effort, quality, and features, but also a risk level that the organization can tolerate. Without the range and risk level, it's all or nothing — not a very rational, or realistic, approach. How do we know on a point estimate what is the probability of achieving it?

Conclusion

Whether or not you use my list of criteria, be sure to list your own before you go shopping for a macro-estimation tool. Tool vendors have a technical name for those who seek tools without stated, written criteria: loose wallets!

Acknowledgements

This article is based on a presentation as moderator of the *January Maryland Society for Software Quality Roundtable 2000*, "Help! for estimation."

References

The following SEI references are often available for free download on the SEI Web site: www.sei.cmu.edu/publications/.

McAndrews, Donald. *Establishing a Software Measurement Process*. SEI-93-TR-16, July 1993.

Park, Robert. *A Manager's Checklist for Validating Software Cost and Schedule Estimates*. SEI-95-SR-004, January 1995.

Park, Robert. *Checklists and Criteria for Evaluating the Cost and Schedule Estimating Capabilities of Software Organizations*. SEI-95-SR-005, January 1995.

Park, Robert. *Goal-Driven Software Measurement — A Guidebook*. SEI-96-HBK-002, August 1996.

Park, Robert et al. *Software Cost and Schedule Estimating: A Process Improvement Initiative*. SEI-94-SR-03, May 1994.

Web Sites

www.incose.org/tools/tooltax/costest_tools.html. Provides an interesting, though slightly out of date, list of software cost estimating tools.

www.methods-tools.com/tools/frames_projmgmt.html. Offers a more up-to-date list, containing many other tools.

About the Author

Stan Rifkin is a principal with Master Systems Inc., a firm that offers advisory services related to software improvement. He worked at the Software Engineering Institute on software process improvement, the American Association for the Advancement of Science as CIO, and the National Headquarters of the American Red Cross as the director of systems development. Mr. Rifkin's previous articles have appeared in March and May 2000 issues of *ITMS*. Mr. Rifkin can be reached at sr@Master-Systems.com.

Please start my subscription to *IT Metrics Strategies*® for one year at \$485, or US \$545 outside North America. Phone Megan Nields at +1 781 641 5118, or fax +1 781 648 1950, or e-mail info@cutter.com.

Please renew my subscription.

Name _____

Title _____

Organization _____

Dept. _____

Address _____

City _____ State/Province _____

Zip/Postal Code _____ Country _____

Tel. _____ Fax _____

E-mail _____

Payment or purchase order enclosed

Please bill my organization

Charge my Mastercard, Visa, American Express, Diners Club, or Carte Blanche

220*5ITS

Card no. _____

Expiration Date _____

Signature _____

Web site: www.cutter.com/itms/

Cutter Information Corp.

Suite 1, 37 Broadway

Arlington, MA 02474-5552 USA