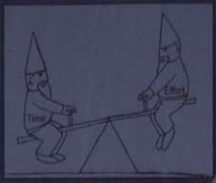

FIVE CORE METRICS

The Intelligence Behind Successful Software Management

Lawrence H. Putnam *and* Ware Myers
authors of MEASURES FOR EXCELLENCE

DH

FIVE CORE METRICS

The Intelligence Behind Successful Software Management

Also Available from Dorset House Publishing

Adaptive Software Development:
A Collaborative Approach to Managing Complex Systems
by James H. Highsmith III foreword by Ken Orr
ISBN: 0-932633-40-4 Copyright ©2000 392 pages, softcover

The Deadline: A Novel About Project Management
by Tom DeMarco
ISBN: 0-932633-39-0 Copyright ©1997 320 pages, softcover

Dr. Peeling's Principles of Management:
Practical Advice for the Front-Line Manager
by Nic Peeling
ISBN: 0-932633-54-4 Copyright ©2003 288 pages, softcover

Measuring and Managing Performance in Organizations
by Robert D. Austin foreword by Tom DeMarco and Timothy Lister
ISBN: 0-932633-36-6 Copyright ©1996 240 pages, softcover

Peopleware: Productive Projects and Teams, 2nd ed.
by Tom DeMarco and Timothy Lister
ISBN: 0-932633-43-9 Copyright ©1999 264 pages, softcover

Project Retrospectives: A Handbook for Team Reviews
by Norman L. Kerth foreword by Gerald M. Weinberg
ISBN: 0-932633-44-7 Copyright ©2001 288 pages, softcover

The Psychology of Computer Programming: Silver Anniversary Edition
by Gerald M. Weinberg
ISBN: 0-932633-42-0 Copyright ©1998 360 pages, softcover

Quality Software Management, Vol. 2: First-Order Measurement
by Gerald M. Weinberg
ISBN: 0-932633-24-2 Copyright ©1993 360 pages, hardcover

For More Information

✔ Contact us for prices, shipping options, availability, and more.

✔ Sign up for *DHQ: The Dorset House Quarterly* in print or PDF.

✔ Send e-mail to subscribe to *e-DHQ,* our e-mail newsletter.

✔ Visit Dorsethouse.com for excerpts, reviews, downloads, and more.

DORSET HOUSE PUBLISHING
An Independent Publisher of Books on
Systems and Software Development and Management. Since 1984.

1-800-DH-BOOKS 1-800-342-6657
212-620-4053 fax: 212-727-1044
info@dorsethouse.com www.dorsethouse.com

FIVE CORE METRICS

The Intelligence Behind Successful Software Management

Lawrence H. Putnam *and* Ware Myers

Dorset House Publishing
3143 Broadway, Suite 2B
New York, New York 10027

Library of Congress Cataloging-in-Publication Data

Putnam, Lawrence H.
Five core metrics : the intelligence behind successful software management / Lawrence H. Putnam, Ware Myers.
p. cm.
Includes bibliographical references and index.
ISBN 0-932633-55-2 (soft cover)
1. Computer software--Development--Management. 2. Computer software--Quality control. I. Myers, Ware. II. Title.
QA76.76.D47P8675 2003
005.1'068--dc21

2003046157

Cover Design: David W. McClintock
Cover Illustrations, from Figures 7-03 and 7-04: Elisabeth Thayer

 Published by Dorset House Publishing Co., Inc., 3143 Broadway, Suite 2B, New York NY 10027

Distributed in the English language in Singapore, the Philippines, and Southeast Asia by Alkem Company (S) Pte. Ltd., Singapore; in the English language in India, Bangladesh, Sri Lanka, Nepal, and Mauritius by Prism Books Pvt., Ltd., Bangalore, India; and in the English language in Japan by Toppan Co., Ltd., Tokyo, Japan.

Printed in the United States of America

Library of Congress Catalog Number: 2003046157

ISBN 13: 978-0-932633-55-2 12 11 10 9 8 7 6 5 4

Digital release by Pearson Education, Inc., June, 2013

Acknowledgments

We want to thank our colleagues who have been most helpful in providing their insights and opinions concerning the material in this book. They have been generous in sharing examples from their experience and a rich lore of stories from their consulting work in a wide variety of different companies, industries, and environments. Our thanks go to Doug Putnam, Larry Putnam Jr., Lauren Thayer, Kate Armel, Michael Mah, Ira Grossman, Bill Sweet, Stan Rifkin, Jim Greene, Anthony Hemens, and Hans Vonk.

Many of the figures in this book were specially prepared by people at Quantitative Software Management or were inspired by figures presented in QSM training materials. These resources were greatly appreciated. In particular, Doug Putnam revised and adapted many of the figures; his help was invaluable. Elisabeth Thayer drew the gnomes that appear on the cover and are used in figures to illustrate some of the key ideas. Again, we are thankful for all these efforts on our behalf.

Many people think that the role of editor and publisher is merely to check the spelling and to correct the grammar. Not so! The work done at Dorset House by Nuno Andrade, Vincent Au, Wendy Eakin, and David McClintock, in suggesting many meaningful changes to the text, has improved the style, consistency, integration of the ideas, and readability. We are deeply indebted to them for their diligent work in tending to these matters.

Finally, we thank our consulting and tools clients with whom we've worked over the past decade or so. These professionals have demonstrated the efficacy and value of good metrics programs leading to useful estimates up front, to control of projects under way, and to quantified benefits of process improvement initiatives. They have succeeded in making these ideas work.

Contents

FIVE CORE METRICS

The Intelligence Behind Successful Software Management

This page intentionally left blank

Introduction

Processes, methodologies, and methods for developing software have become the center of much activity in recent years. Today's practices for developing software are much more effective than those historically employed. However, these practices are complex, and they rely heavily on software tools. Neither processes nor tools are easy to adopt. Mastering them takes *time* over a period of years, and the measurement of improvement year by year keeps the ever-changing goal in management's sights.

Moreover, a process improvement effort does not rest solely on the enthusiasm of a solitary developer here or there. It encompasses the entire software organization and even its clients, in-house or out. A program of this magnitude depends on the understanding and support of the management structure—not only at the beginning, when enthusiasm runs high, but also over the long haul.

But what can sustain such interest, if not wild enthusiasm, over time without end? Let us look to the history of business for guidance. Double-entry bookkeeping did not arrive with the first Homo sapiens, tens of thousands of years ago. It was invented rather recently—in fourteenth-century Venice. Bookkeeping is said by some to be the epitome of dullness. Yet for six centuries, the principal output of the accounting process—the profit metric—has maintained the interest

and excited the daily effort of businesspeople. It is a way of keeping score in the great game of business.

Software development is part of this great game. At some point, profit tells the participants whether the software game is going well. Unfortunately, profit comes rather late in the game. The project may have crashed and burned long before the bean counters' profit-and-loss statement reveals that the company is wallowing in loss.

Therefore, software development organizations need a more immediate measure of how they are doing. This measure is especially needed to motivate senior management to persist in its support of process improvement. We need this measure to enable project managers and the developers themselves to persist in their efforts to control software development. We need it to enlighten clients and users.

Some approaches use periodic assessment to motivate improvement, such as the Capability Maturity Model (CMM) of the Software Engineering Institute (SEI) and specification 9001 of the International Standards Organization (ISO). The attention they have gained supports our belief that many managers feel the need, not only for the guidance that these models provide, but also for the assurance that assessment offers.

Unfortunately, assessments are of necessity periodic, years apart. When performed, an assessment does indeed give a jolt to the pursuit of process improvement, but in the long intervals between assessments, motivation falters. Further, assessments are imprecise. They depend upon the judgment of the assessors, who may be swayed by the artifices of those assessed.

Metrics, in contrast, are continuous, that is, weekly or monthly—as often as measures are made. They are reliable, since they rest upon counting definable elements of software production. They meet the need for immediate control at the project level. They also meet the need to measure process improvement. Being accurate and frequent, they quiet management's nerves—or excite management to action—every week or month.

Many observers proclaim passionately that people are important. Measures don't solve problems, they say, people do. Metrics just get in the way, they add. We agree that people are important—leadership is better than dictatorship, and people still have to solve the problems encountered in software development. Metrics, when poorly chosen, inaccurately collected, and unwisely applied, do upset people and impede problem solving. But measures that are well chosen, accurately collected, and wisely applied do intensify the motivation that improves the process within which people solve problems.

Software development, to be effective, needs an appropriate process. A process, to endure successfully in a world of limited resources and time, has to be measured, evaluated on those metrics, and often redirected as a result. Its relationship to scarce resources is gauged through metrics. The common complaint of overstimulated developers—"Just get out of our way"—fails to grasp these fundamentals.

Our intent in this book is to show that just a few key metrics—five, in fact—are enough to meet these needs. These five metrics provide the equivalent of the historic profit metric that energizes business, but they provide it much sooner—during development, instead of after release. Consequently, the metrics have more of a chance at critical points to motivate senior management and senior stakeholders to continue their support. Simultaneously, the metrics encourage acquisition managers, project managers, and the developers themselves to pursue the often frustrating path of process improvement.

Before describing the structure of this book, we pause to describe the experiences from whence this research sprang.

The Evolution of the Metrics

The ideas expressed in this book are the outcome of Larry Putnam's research and work in software development, first in the Army and then with twenty-five years of operating the consulting firm Quantitative Software Management, Inc. (QSM) of McLean, Virginia. Larry's final years in the Army were particularly productive in developing these concepts:

> My Army career roughly coincided with the early decades of the computer age. In a series of assignments, between periods with the troops, I became familiar with the ability of the pioneer computers to perform huge, tedious computations. It was this experience that prepared me to apply computer power, later on, to the metrics of software development.
>
> Early on (between 1959 and 1961), in one of the physics courses I took at the Naval Postgraduate School, I had to do some tedious calculations to a precision of twelve decimal places. The only tool available to us for that kind of work was the desktop mechanical calculator, and I had to hire time on it in town. Tedious experiences like that made computers very appealing when I first encountered them.

Computing Nuclear Weapons Effects

My first encounter with nuclear weapons effects was at the Army's Special Weapons Development Division in the Combat Development Command at Fort Bliss, Texas, where I was stationed between 1961 and 1964. The computer we used was a Bendix G15, which was the size of a refrigerator and had about as much power as a programmable calculator has today. One of my jobs was supervising the preparation of the Army's nuclear weapons selection tables, which commanders used to pick the right weapon for a particular tactical operation. As the weapons developers upgraded the weapons, the Army had to recalculate these tables, each of them several hundred pages in length.

At first we programmed the G15 in Assembly language. Later, I had an opportunity to program the machine in a higher-order language, ALGOL. These were very small engineering programs of perhaps fifty-odd lines of code.

A few years later (in 1966), I needed to do some blast calculations to support the course on nuclear weapons effects I was teaching at the Defense Atomic Support Agency in Albuquerque, New Mexico. Next door, Sandia Laboratories had just received a Univac 1108, the largest scientific computer then available. They had not yet fully loaded it, so they offered time to users on the base, providing the applicants did their own programming. They offered a FORTRAN course for this purpose, and I took it.

As you may expect, when the night operators fed my first deck of IBM punch cards into this giant computer, it immediately kicked me off. The FORTRAN course had failed to teach the procedures for job control cards! After about ten tries, I got my program past that hurdle, only to run into syntax errors in my own program. It turned out to be a lengthy period before I got my program to compile, run, and generate the data I wanted. In the back of my head, however, I lodged a firsthand appreciation for the perils of big-time computer programming. Also lodged there, fortunately, was some ability to apply mathematics and statistics to difficult problems. That would come in handy later.

Computer Budgets in the Pentagon

In 1972, after I'd completed a tour in Vietnam and two years commanding troops at Fort Knox, Kentucky, the time came for duty in Washington. The personnel people, in their inscrutable way, divined that I was eminently qualified to take charge of the Army's automatic data processing budget. I was going to deal with the budgetary process for the Army's procurement of computers and funding of software development programs.

The Army was spending close to $100 million per year developing software to automate its business functions—payroll, inventory management, real property management of bases around the world, mobilization plans, force deployment plans, command and control structure. It was virtually everything that had anything to do with business and logistics. The hardware on which these programs were to run cost another couple hundred million dollars.

As I began this tour, I knew little about software beyond the FORTRAN, ALGOL, and Basic programs I had written. Most of the initial work on these Army business functions had been coded in Assembly language. The Army Computer Systems Command was redoing this Assembly code in higher-order languages, principally COBOL. We were in the midst of completing 50 to 70 systems in the range of 50,000 to 400,000 lines of COBOL when I began to hear those ominous words, "overruns," and "slippages."

I really became aware of the Army's problems with software the first time I went to the budget table across from the people in the Office of the Secretary of Defense. We were looking at the next fiscal year and the five years that followed. To take an example, when the Standard Installation Division Personnel System first became operational the year before, its project organization had 118 people. For the next year, we had projected the count to fall to 116, then to 90 for each of the next five years. Those were numbers that had come up from the field.

"What are these ninety people going to do?" the budget analyst from the Office of the Secretary of Defense asked, reasonably enough. "Isn't the system finished?"

Well, there was a big silence in the room. I was new. I didn't know the answer. I looked to my right, then to my left,

at the long-term civilian employees who had come into the Pentagon with the first computers. They were the acknowledged experts, but they were strangely quiet. Finally, the lame answer dribbled out: "maintenance." Nobody on the Army side of the table could satisfactorily explain what that meant.

"Look, this is a ten-million-dollar item," the budget analyst finally said. "Unless I can get an answer, I'll have to delete it. It's getting late, so let's adjourn for today and reconvene at nine A.M."

We scurried off and called the Army Computer Systems Command, which was responsible for the Personnel System. We waited by the phone into the evening. Finally, the response came, again some lame comment about maintenance, but we knew it would not make sense to the budget people. And it didn't. By 9:15 the next morning, we had lost $10 million.

After the budget meeting, as I walked down the halls of the Pentagon with my boss, a major general from the Corps of Engineers, he mused, "You know, Larry, this business of trying to plan the resources and schedule for software projects is very mystifying. It wasn't like that in the Corps of Engineers. Even early on in a project—the big dams, the waterways—we always had some feel for the physical resources we would need: how many dump trucks, the number of cubic yards of concrete, the power shovels, the people. From numbers like these, we could make a crude estimate of schedule and costs."

We reached the elevators, and he fell silent until we got off on our floor.

Then he continued: "Any time I try to get similar answers on software, I get a dialogue on the architecture of the computer itself, or a little explanation of bits and bytes, or some other irrelevancy. Never anything about how long the work is going to take, how many people it is going to require, what it will cost, or how good it will be at delivery. That's the kind of information we need at our level, here in the Pentagon. That's what we need to come to grips with this business of planning and managing software development."

At this point, he turned and went into his office. He didn't seem to expect any immediate answer from me, but his comments set me off on a line of thinking that has lasted to this day. How do software systems projects behave? Can we

model this behavior with a few core parameters? Is there a way to get the answers that senior managers want?

A couple of weeks after this budget disaster, I stumbled across a small paperback in the Pentagon bookstore. It had a chapter on managing R&D projects, by Peter Norden of IBM.[1] Norden showed a series of curves, such as the one in Figure I-1, that depicted the buildup, peak, and tail-off of the staffing levels required to move a project through research and development and into production. He pointed out that some of these projects were for software, some were hardware-related, and some were composites of both.

What struck me about the function, which Norden identified as a Rayleigh curve, was that it had just two parameters. One was the area under the curve, which was proportional to the effort applied. (In the case of software projects, effort is proportional to cost.) The other was the time parameter, which related to the schedule.

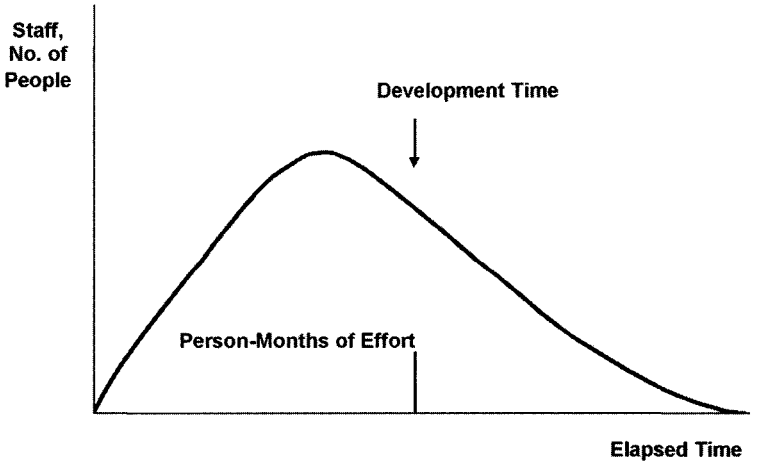

Figure I-1: *A Norden-Rayleigh curve showing the number of staff and the amount of effort required by development projects over time.*

I found that I could easily adapt these Rayleigh curves to the budgetary data I had on the Army software projects. We had the number of person-years applied to each project in our budget, for each fiscal year of each project. So, I quickly plot-

[1]Peter V. Norden, "Useful Tools for Project Management," *Operations Research in Research and Development,* ed. B.V. Dean (New York: John Wiley & Sons, 1963).

ted all the software systems that we had in inventory and under development. From the plots, I could establish the parameters of the Rayleigh curves. Then, from these parameters, I could project the curves out to the end of the budgeting cycle. Within a month, I had about fifty large Army development projects under this kind of control. I was able to do credible budget forecasts for those projects—at least for five years into the future.

The next time the budget hearings came around, a year later, we were in the downsizing phase at the end of the Vietnam War. Budget cuts were endemic, and we were asked to cut the application of effort on a number of existing systems. The turnaround time was short; we had twenty-four hours to report on the impact such cuts would have.

Now that I had the Rayleigh curve and an understanding of that methodology, I was able, using a pocket calculator programmed with the Norden-Rayleigh function, to quickly make estimates of what would happen if we reduced the projections for several of the projects. It was easy to show that the aggregate of these cuts would wipe out the Army's capability to start any new software projects for the following three years.

We did not lose any money at that budget meeting.

Applying the Rayleigh Concept to New Projects

Naturally, the next important question was,

> How do I use the equations that stand behind the Rayleigh curve to generate an estimate for a new project?

More questions followed:

> It's nice to pick up those that are already under way, but is there some way I can find the time and effort for a new project? Is there a way to build a budgeting and staffing profile for getting the work done?

I looked into that. Right away the notion arose that somehow we had to relate the size of the Rayleigh curve—the time and effort it represented—with the amount of function the project was to create. To measure the functionality, we had to ask, How do the people building software for the Army—its in-

house developers, like the Army Computer Systems Command and its contractors—think about the functionality they are creating?

I found out that they thought about the lines of code they had to write. They talked a lot about the number of files they were creating, the number of reports they were generating, and the number of screens they had to bring up. I saw that those types of entities were clearly related to the amount of functionality a project had to create. I would have to relate these functional entities to the schedule and effort needed to get the job done.

Here, my experience with analyzing nuclear effects data came into play. I knew that I had to

- *measure* a number of samples of the activity
- find the *pattern* in these measurements[2]

The next year-and-a-half to two years, I spent about a third to half of my daily Army schedule analyzing data. The first set of data was a group of about fifteen to twenty systems from the Army Computer Systems Command. I attempted some mathematical curve-fitting, relating the size of those systems in lines of code, files, reports, or screens to the known development schedules and the associated person-months of effort.

The first approach was to use a simple regression analysis of functionality, expressed in lines of code, as the independent variable; person-months of effort served as the dependent variable. I used the same approach with schedule.

Next, I did some multiple regression analysis in which I related effort to combinations of lines of code, files, reports, and screens. The statistical parameters that came out showed that these relationships might be useful for predictions, but they were not extraordinarily good fits. Certainly, more work and investigation were needed before any conclusions could be drawn.

By this time (1975 to 1976), I had been in contact with other investigators in this area. Judy Clapp from the Mitre Corporation had done some studies on ten to fifteen scientific and engineering systems that were being built for the Electronics Systems Division of the Air Force Systems Command at Hanscom Air Force Base. C.E. Walston and C.P. Felix at IBM Federal Systems Division had published a paper in the

[2]Lawrence H. Putnam and Ware Myers, *Industrial Strength Software: Effective Management Using Measurement* (Los Alamitos, Calif.: IEEE Computer Society, 1997), p. 5.

IBM Systems Journal,[3] having amassed a database of about seventy-five projects that gave a good feel for a range of different parameters related to software. All of this information was very useful toward establishing relationships between lines of code, pages of documentation, time, effort, and staffing.

In trying to do this analytical work, I had to go back about twenty years, to my academic training at the Naval Postgraduate School, to refresh my memory on statistical analyses, working with data, and drawing logical inferences and conclusions. I had to do a lot of relearning to polish skills that had become very rusty from years of neglect.

The Rayleigh Concept Leads to the Software Equation

One promising experiment employed multiple regression analysis to relate the size of the systems in lines of code to the schedule and the person-months of effort applied. I did these curve-fits first with the Army data, then with the Electronics Systems Division data, followed by the IBM data. I was lucky in that I got some very nice fits in about twenty of the Army data systems.

Concurrently, I did some theoretical work on integrating Rayleigh curves. I tried to establish the parameters of integration from a little bit of the historic data from the Army and IBM. I found good consistency in generating the key parameters for the Rayleigh equation:

- the work effort (area under the curve)
- the schedule parameter (distance along the horizontal axis)

These different, independent approaches to getting a parameter-estimating equation were leading me in the same direction and producing similar results. What ultimately fell out is what I now call the *software equation*. It related the amount of function that had to be created to the time and effort required. It originally looked like this:

Quantity of Function = Constant x Effort x Schedule

---

[3]C.E. Walston and C.P. Felix, "A Method of Programming Measurement and Estimation," *IBM Systems Journal*, Vol. 16, No. 1 (1977), pp. 54–73.

In the process of fitting a curve to the data representing a number of systems, I made three significant observations:

1. A *constant* was generated.
2. Effort and time were both raised to *powers.*
3. *Effort and time* were present in one equation.

Constant. The fact that a constant was generated in the process of curve-fitting is not in itself significant. A constant is always generated in this process. When you generate a line or a curve from data on a number of instances, you will find that you need a constant to balance the ensuing equation. The point of importance here is that this constant is not invented to balance the equation. It is an outcome of the historic data. Consequently, it has some kind of relationship to the data from which it originated.

Parenthetically, we should note that this constant is not the single, unchanging number its name implies. It is actually a parameter. Although in any given instance, such as a particular project, it is a single number, it may be different for each project, depending on the associated facts.

I thought a lot about what this relationship might represent, what the physical meaning of this parameter might be. Somehow, it seemed to be related to the efficiency of the development organization or the level of technology the organization was applying to its development practices. That is, where expert opinion believed an organization to be more advanced, a larger parameter was at work. That is, organizations that expert opinion believed to be more advanced fell heir to a larger parameter.

The first name I used to describe this empirically determined parameter was Technology Factor. I used that term in the first papers I published with the IEEE Computer Society, between 1976 and 1977. I have continued to use that parameter to represent the efficiency of the software development organization. Over the years, I have renamed it several times:

- Technology Factor
- Technology Constant
- Productivity Constant
- Process Productivity Parameter

The latter is probably the most descriptive term for the real relationship of this mathematical parameter to the software development process.

Powers. In the process of curve-fitting, I found that exponents were associated with both the time and effort parameters. The presence of these exponents means that the software equation is nonlinear. This, in turn, signifies that the software development process is not simple or linear; it is complex, or nonlinear. Well, in view of the difficulty we have always had with it, that complexity is not a great surprise!

Effort and time together. The curve-fitting brought effort and schedule together in the same equation. In other words, they influence each other. They are not independent entities. This conjunction leads to some possibilities in project planning, such as the "time-effort trade-off" that we will explore at length in Chapter 11. The additional fact that both effort and time carry exponents has further implications as well.

This work was the genesis of my software equation, which we still use today. Though originally derived by statistical analysis, the data I analyzed was itself real. Since then, the QSM software equation has been applied to tens of thousands of projects that were also real. The result: It has proven to be a very good measurement and estimating tool for more than twenty-five years.

The QSM software equation is a macro model that links the amount of function to be created to the management parameters of schedule and the effort required for production. The empirically determined constant, or process productivity parameter, represents the productive capability of the organization doing the work. This equation brings together four of the management-oriented measures that the major general from the Engineer Corps yearned for, now some thirty years ago.

The presence of the process productivity parameter suggests that the software equation provides a very good way to tune an estimating process. If you know the size, time, and effort of your completed projects, you can calculate your process productivity parameter. Then you can use this calculated parameter in making an estimate for a new project. So long as the environment, tools, methods, practices, and skills of the people have not changed dramatically from one project to the next, this process of playing back historic data can serve as a very useful, simple, straightforward calibration tool.

Second Key Relationship: Manpower Buildup Equation

The other relationship that emerged in these studies was the direct one between time and effort. Clearly, these two were parameters in our Rayleigh equation. But was there anything we could learn from the basic data as to the relationship between the two? Again, more curve-fitting. There was a distinct relationship:

Effort is proportional to schedule cubed.

This relationship had been noted and discovered earlier by Felix and Walston and several other investigators who had done research in software cost estimating.[4]

This finding was especially important because it gave me the basis for making estimates. I had two equations and two unknowns (time and effort). The software equation involved size, time, effort, and the process productivity parameter. The second equation linked effort with development time. With two equations, one could solve for the two unknowns.

The second equation, of course, also required some parametric determinations. We found a parameter family that seemed to relate to the staffing style of the organization. Organizations that use large teams of people tend to build up staff rapidly. Rapid buildup produces a high value for the ratio of effort divided by development time cubed.

Organizations that work in small teams generally take longer to complete development. Small teams are typical of engineering companies that tend to solve a sequential set of problems one step at a time. For such companies, I saw that the relationship of effort divided by development time cubed produced a much smaller number for the buildup parameter. This observation told us that organizations adopted different staffing styles. This parameter was actually a measure of the manpower acceleration being applied to a software project.

It became evident in studying these different types of organizations that large-team organizations tend to finish their projects a little bit faster than small-team organizations. The latter took a little bit longer to complete their work, all other things being equal. This finding suggested some sort of trade-off between team size and how long it takes to get the work done.

---

[4]Ibid.

The Rayleigh Curve As a Process Control Vehicle

The other significant idea that I started working on during my Army years was the notion of using the Rayleigh curve as a process control vehicle for projects while they are under way. The curve projected the planned staffing and effort. As the project progressed, actual staffing and effort could be compared to the plan. Managers could observe deviations from the plan and initiate prompt corrective action.

An extension of the control idea was to take real data as it was happening on a project. If this data differed from the original plan, something had to change. The purpose was to calculate the actual process productivity parameter being achieved. Using it, a new Rayleigh curve could be projected to a new completion date. The new curve would let you dynamically predict effort, cost, and schedule to completion.

I did some early curve-fitting investigations but ran into some problems and snags that for a time prevented this idea from being fully realized. Nevertheless, there was enough work and enough positive results to suggest that the process control application should be pursued.

It had become evident, however, that not many people were interested in dynamic control. Most organizations were having so much trouble coming up with the initial forecast that the idea of learning how to control an ongoing project was not high on their priority list. Trying to reach good solutions in dynamic measurement and control was premature. (However, I did eventually work out the project control aspects, as we report in Chapter 13.)

And now, the book itself, reflecting the experience of another quarter of a century, is divided into four parts. Part I, What Software Stakeholders Want, begins on a positive note with the view that some software organizations are doing very well. They are doing well because they have integrated key metrics into their development process. These practices give them predictability.

Part II covers the five core metrics needed for effective control: schedule time, effort, functionality (as expressed in size), reliability, and productivity. Because these five metrics are related to each other, developers can use known values to predict the others. This provides a base for estimating. Then, they can control the progress of their

projects against those predictions. Further, with these measures, they can track improvements in the way they develop software.

Part III is given over to the application of metrics at the project level. This set of chapters applies the metric concepts to the estimation and control of a single project. Estimation and control are important because software development does not in reality begin with programmers sitting down to write code. There are such preliminary maneuvers as finding out what to do. That "what" is the essential basis of the time, effort, and other resources needed to eventually write code that implements the "what" successfully.

Part IV extends the employment of metrics to control that resides above the project level. The general idea is that the acquirers of software development capability can employ these metrics to guide their relationship with development organizations. Higher levels of management can use them to manage a portfolio of projects and to guide their employment of reusable components. Organizations can gauge improvement by tracking these metrics.

This page intentionally left blank

Chapter 3
Integrate Metrics with Software Development

People are important, and people have to solve the intricate problems encountered in software development. It is equally true that collaborative leadership is better than command management, especially in knowledge work, for people do not solve problems on command—at least not well. It is also true that a software development process, clumsily applied, can get in the way of what people are trying to do. Moreover, we might as well admit—for it is true—that when metrics are poorly chosen, inaccurately collected, and unwisely applied, people get upset. When people are upset, they solve problems poorly. To put it in a nutshell, people solve problems; metrics provide the schedule time and staff allowance within which people solve problems.

Thus, when pertinent metrics are applied effectively, people become more productive. Metrics make the process of software development more reliable and efficient. The challenge is to work metrics into that process. This chapter begins the exploration of the typical phases of development and the ways metrics can be integrated into the process.

Metrics Meter Limited Resources

As we pointed out in the previous chapter, software development, like everything else on this third rock from the sun, operates on a planet of limited resources that measures time by rotations of its globe. Every

rotation carries costs because the people doing the work like to eat every day. That is, we mean to say, neither staff power nor schedule time are free goods.

The reliability of the product isn't free either. People work over a period of time to produce software. That software may be of high reliability—with few defects—because the people had sufficient time to avoid or correct the defects. However, software released with many defects merely transfers that working time, usually multiplied one hundred-fold, to the users, the help-line crew, and the maintenance staff.

Staff, schedule time, and the number of defects represent three of the five core metrics. The fourth core metric is the amount of function contained in a software product. It is commonly measured in lines of source code or in function points. The fifth core metric is the productivity level—or process productivity—of the project. But when do we measure the five core metrics: size, productivity, time, effort, and reliability?

In the beginning, the software project is just a gleam in some dreamer's eye, and there is nothing to measure. At the end, we have a product and can count the lines of code, but at that point, the count is of little value. Measurements would have come in handy earlier, when we were trying to estimate the time and effort. It is evident also that we need metrics at many in-between points. That is, between the start and the end of the software development process. We need to have metrics integrated into that process. The work has to be measured and evaluated on those metrics, and often redirected as a result.

Thus, development operates through process. Process, in turn, operates through resources. The key resources, time and effort, are scarce. In consequence, they have to be metered out to projects. That metering is properly the province of a measurement system, or metrics for short.

That is the nub of the metric side of the argument. Unfortunately, developers in the trenches often see the situation quite differently. They see a rather amorphous project, full of problems that may take a long time to penetrate. They see a schedule imposed by upper management or clients with little grasp of the problems still to be unearthed. They see a staff of inadequate size with junior members inexperienced in the problems to come. To them, that too-short schedule and inexperienced staff are the "metrics" imposed on the project. And, of course, in that situation, these "metrics" are what pass as the outcome of management's stab at the time and effort metrics.

That word "stab" is the key to a happier metric future. Suppose that instead of "stabbing" more or less blindly at the metrics, management were able to reach these metrics as the outcome of an intelligent methodology. That is, the time and effort metrics were well suited to the needs of the developers. The productivity level at which the staff can function was well established. Under this scenario, developers could be happy with the metrics offered with the project.

It is evident that developers are unhappy, not with metrics per se, but with lousy metrics. In a world of finite resources, metrics have to be an integral part of an effective software process. But they don't have to be lousy.

What Is the Process?

An early representation of the software development process was the waterfall model, summarized in the first column of Table 3-1. Many understood the model to prescribe a one-way path through the workflows of software development: requirements capture, analysis, design, and so on, as shown in the table. In truth, that understanding was based on the first diagram of workflows in Winston Royce's landmark paper, "Managing the Development of Large Software Systems." If readers of the paper had turned the page, they would have seen that the second diagram shows feedback arrows from each workflow to the preceding one. "As each step progresses and the design is further detailed, there is an iteration with the preceding and succeeding steps," Royce wrote.[1] Really, a quite modern attitude! Despite Royce's iterative view, many software people still consider his model to be a sequential, single-pass flow of work—not an iterative process.

The U.S. Department of Defense (DoD) divided the software problem into four phases: feasibility study, high-level design, main build, and operation and maintenance. These phases were convenient for managing contracted parts of the development. With little hard knowledge to go on, the first two phases could be handled as level-of-effort contracts. By the time the high-level design was completed, the knowledge was on hand for a fixed-price contract.

Developed in 1999, the Unified Process brought together the previous processes or methodologies of Ivar Jacobson, Grady Booch, and James Rumbaugh, each already a well-established methodologist. Although divided into four phases like the Department of Defense's version, the phase names are quite different. The four phases are named not in terms of their content, but in terms of their position in

[1]Winston W. Royce, "Managing the Development of Large Software Systems," *Proceedings, IEEE WESCON* (August 1970), p. 2.

the development sequence. Within the Elaboration phase, for example, the Unified Process locates a number of activities involved in getting ready to build the system in the next phase.

Table 3-1: *The waterfall model of software development, originating in the 1960's, listed what we now call workflow activities. It was followed by the four phases of the Department of Defense process. The Unified Process, pulled together in 1999, redefines the four phases and assigns them new names to make clear that they are something different. In the final column, a survey author, attempting to come up with a generic set of names, confuses workflows and phases.*

	Waterfall[2]	Dept. of Defense[3]	Unified Process[4]	Generic[5]
I	Requirements	Feasibility Study	Inception	Initiation
II	Analysis	High-Level Design	Elaboration	Conceptual Requirements
III	Design	Main Build	Construction	Analysis
IV	Coding	Operation, Maintenance	Transition	Design
V	Testing			Construction
VI	Operations			Deployment

The generic set, shown in Table 3-1, presents a combination of sequence names, such as Initiation, and workflow names, such as Analysis, that is confusing. The confusion arises from our long immersion in the waterfall model. If you are going to go through the workflow activities (those listed in the first column of the table) only once—in waterfall fashion—process and workflows are the same. However, in the more modern sense of the spiral model or iterative development, which assume repeated passes through workflows instead of a single pass, there is a distinction between process and workflows. For example, if a team decides that it needs to develop a prototype in the first phase, it has to go through an abbreviated version of requirements capture, analysis, design, and coding to get to the prototype. In other words, part or all of the workflows may occur in any phase. There is a

---

[2]Royce, loc. cit., pp. 1–9.

[3]Lawrence H. Putnam and Ware Myers, *Measures for Excellence: Reliable Software on Time, Within Budget* (Englewood Cliffs, N.J.: Prentice Hall, 1992).

[4]Ivar Jacobson, Grady Booch, and James Rumbaugh, *The Unified Software Development Process* (Reading, Mass.: Addison-Wesley, 1999).

[5]Ellen Gottesdiener, "OO Methodologies: Process & Product Patterns," *Component Strategies* (November 1998), pp. 34–44.

distinction between phase and workflow, and the Unified Process grasps this difference.

In the following sections, though, we describe each phase of the software development process in terms of the Unified Process model, and we note that the DoD model, or some version of it, is widely employed.

Phase One: Inception

It would be nice to know, before spending a few million dollars, that the proposed system is technically feasible and that there is a business case for it. In the very beginning, when all you have is a high-level executive waving his arm broadly and saying, "Wouldn't it be super to have a single system that integrates all our operations?" you can't estimate time and effort. You don't have enough information to make even a rough estimate of the size of the necessary software.

You first have to take three preliminary steps: You have to break the grand vision down into something concrete enough to study; you have to define what is within the system and what is external to it; you have to figure out if your organization can produce this system that you have roughly defined.

Is the system technically feasible at the current state of the art? This question seems to arise in areas such as the Department of Defense, the Federal Aviation Administration, and other organizations where well-meaning but not technically based leaders dream dreams that reach beyond what is immediately possible. This question also arises in new areas, sometimes called "green fields," in which we have not previously done a project. In more mundane fields, feasibility of many applications is more certain. It is enough to realize that the project is a follow-on or that the developers have worked in this application area before. That experience helps us feel certain that the work is at least feasible.

Even if a proposal is technically feasible, though, it may not be appropriate for our particular organization at this time. We may lack certain skills needed by the project. Our organization may not have access to the needed funds. The product may not be consonant with our company's core competence. We explore these issues and others in greater depth in Chapter 9. In the meantime, Figure 3-1 illustrates the position of Phase One at the beginning of preliminary work on the project.

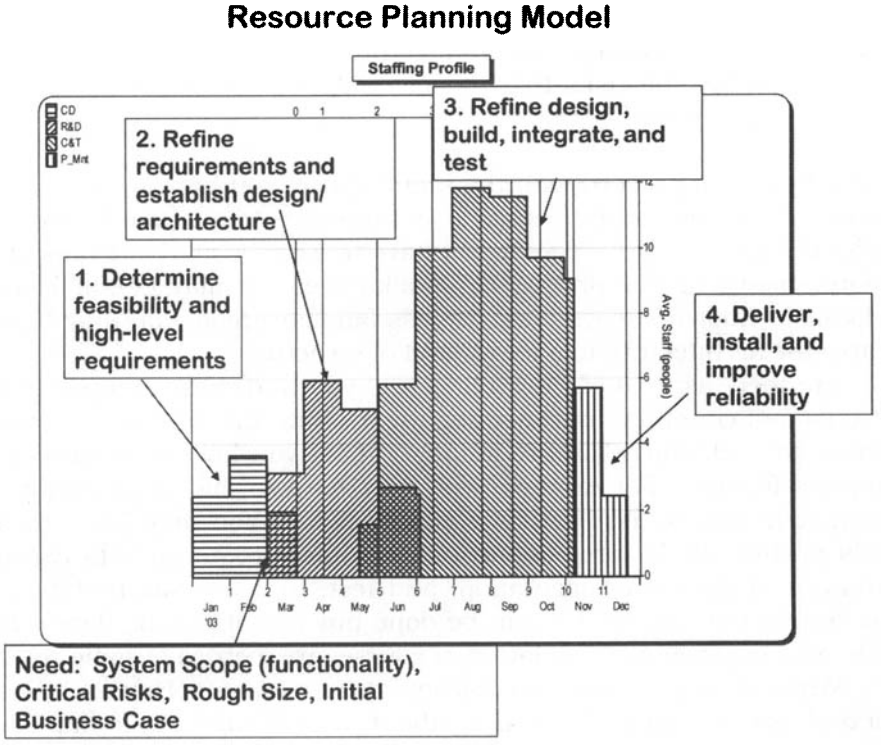

Figure 3-1: *The Phase One (Inception or Feasibility) team does enough work to justify the move into Phase Two.*

Several of the key decisions made in the first phase should rest on a background of reliable metrics. Even the rough estimate of system size that is possible in this phase depends upon having readily available knowledge of the size of past systems. Coming up with the very rough estimate of construction time, effort, and cost involves having some knowledge of how to turn the size estimate into this very rough estimate.

Phase Two: Elaboration

In order to bid on a business basis, you need to extend further your knowledge of the project. You need to know with considerable precision what you are going to do. That is the purpose of the Elaboration phase. This phase has three goals:

- Extend the requirements to the point of blocking out most of the architecture.
- Block out most of the architecture.
- Identify the risks that, if not mitigated, could result in excess costs.

Before you can proceed with the architecture, you have to have a good grasp of the key requirements. It is these requirements on which you base the architecture. You do not have to capture all the details of the requirements in this phase. In familiar areas, it may be sufficient to block out known subsystems. In less familiar areas, you may have to carry the architecture to a lower level of structure.

Are there aspects of the architecture you understand so poorly that you cannot count on implementing them within the time and effort estimates toward which you are aiming? If so, you have to explore these aspects further. For example, you may have to find an algorithm, or even code one, to assure that it can be done. You may have to carry this portion of the project through complete requirements capture, analysis, design, implementation, and test, to a working prototype, to be certain not only that it can be done but that it can be done within business constraints. The latter, of course, are metrics of some sort.

When does your estimate become good enough? During this second phase, as you find out more about what it will take to implement the system, your size estimate becomes more precise. Your corresponding time and effort estimates become more accurate. You carry the architecture, design, and risk reduction to the point at which your estimate meets two criteria. First, it falls within the plus and minus limits normally associated with practice in your field of business. Second, it is within some probability of successful completion. Note that both of these criteria are based on metrics!

Figure 3-2 summarizes what the Phase Two team needs to accomplish.

Just where, during the progress through Phase Two, the size estimate becomes good enough to base a bid on, is not a settled issue. Under pressure from management, customers, and the market, this phase is often ended prematurely before enough is known to support a valid bid. Of course, for organizations with little in the way of a background in metrics, a lack of knowledge of the system being bid makes little difference. They were just guessing anyway. For the organization using metric methods, the size estimate midway through this phase still has too large a margin of error. Its estimates of time and effort at this point will have a comparably large range of uncertainty. Bidding such numbers leads to all the troubles with which we are familiar in

the third phase, Construction—time and effort overruns, poor quality, even cancelled projects.

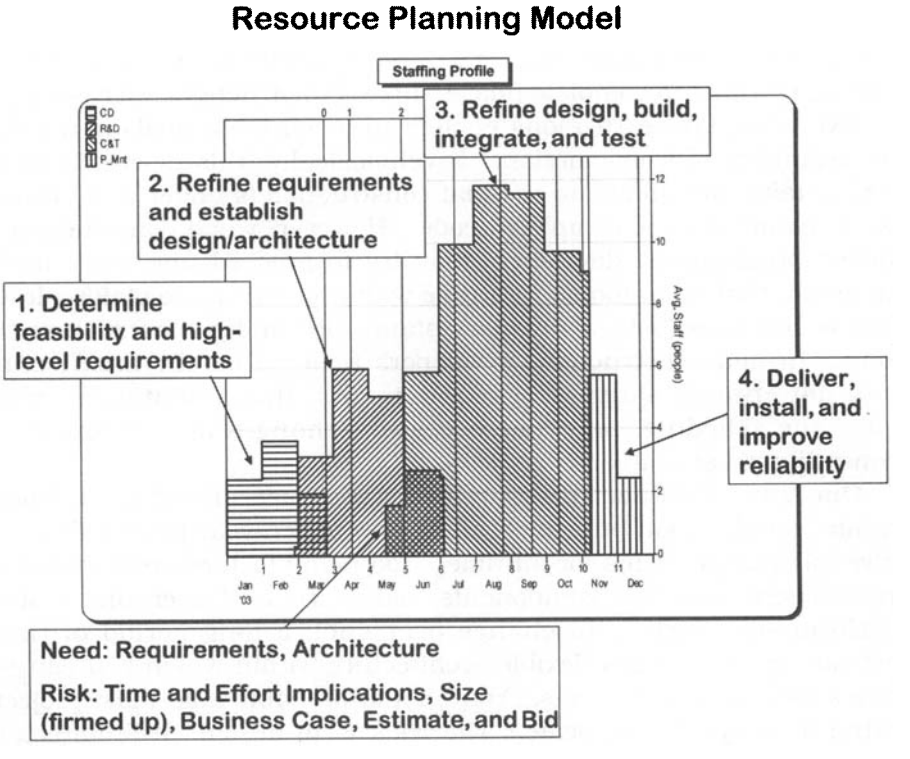

Figure 3-2: *The Phase Two, Elaboration, team carries the exploration of the project to the point of supporting a bid.*

As long as bids are based more on executive intuition than on metrics, it makes little difference where you end the second phase. For organizations that are using better estimating techniques and learning how to make more precise estimates, it is more important to carry the second phase to the point of having an accurate size estimate.

Well, that sounds reasonable, you may be thinking. Then, on further thought, you ask, "Who is going to finance this up-front work?" Business units sometimes like to think they are passing costs to some other business unit. Between companies, the one requesting a bid likes to think it is passing the cost of what it ought to absorb to the software contractor. Similarly, within a company, a department needing software sometimes tries to pass the cost of planning that software

to the software department. If these little "successes" result in inadequate attention to Phase Two, it is the using company or department that eventually suffers from the poor software that results. Thus, whoever finances Phase Two, the using organization needs to make sure that the financing reaches the goal of providing the schedule and effort funding required to complete Phase Three, Construction, successfully.

Extending Phase Two long enough to reach these goals may delay the beginning of Construction. Psychologically, this deferral can be hard to take. We all like to see that construction is under way. People like to count lines of completed code. However, when construction is started prematurely, developers may try to proceed under an inadequate bid, that is, in too short a time with an inadequate staffing level. That is bad from both a business standpoint and a technical standpoint. In mid-construction, developers realize that the architecture does not entirely satisfy the requirements, that unmitigated risks upset the schedule, and that necessary changes are costing more money than is available.

One way to reduce these problems is to keep the project length within bounds. Don't try to set up a five- or ten-year project that will solve all your problems for all time. Too many factors—requirements, environment, reusable components, hardware, and operating system platform—are certain to change over such a long period of time. Instead, try to set up a flexible architecture within which you can set up a series of short projects. You should probably hold most projects within a two-year time scale. The trick is to design an architecture that is extendable into subsequent releases and generations of the product.

In this second phase, Elaboration, a measurement system is even more critical than in the first phase. It is this phase that, by definition, concludes with a business bid. A bid is the supreme metric that governs the project through the Construction Phase and the Transition Phase. But a bid does not spring into existence from nothing, like lightning from the brow of Zeus. Rather, it is built up from lesser metrics, like size, productivity level, and reliability.

Phase Three: Construction

As we mentioned at the beginning of this chapter, *people solve problems;* metrics don't. However, metrics can help people solve problems. There have to be *enough* people over time, and that is a metric known as effort, measured in staff hours or staff months. There has to be a sufficient period of time and that is another metric, schedule time,

measured in weeks, months, or years. Moreover, users of software generally want the people to solve the problems adequately—we might hang the sobriquet "quality" on that desire. There are many aspects of quality, some not readily reducible to a number. Reliability is quantifiable, however, and reflects to some extent the other aspects of quality. In the form of defect rate (or its reciprocal, mean time to defect), reliability is the third key metric.

Therefore, the time and effort implicit in the business bid provide the problem-solving people with the effort and time they need in the third phase to construct software to an acceptable level of reliability and quality. Ergo, metrics are critically important to the success of the third phase.

In addition, metrics play a second role in the Construction Phase: control. That is an ominous word—nobody likes to be controlled. We would all like to be free spirits! But remember, we live in a world of limited resources, and *control* is what keeps us safely within those limits.

Given the initial estimated metrics for effort (staff-months), schedule, and defects at the end of the second phase, we can project the rate at which effort will be expended and defects will be discovered during the Construction Phase. We can also project the rate at which *function*, such as lines of code, will be completed. In addition to projecting the average rate of occurrence of these metrics, we can project bands above and below the average.

We have now established the basis for "statistical control" of the key variables in the Construction Phase. Statistical control means that we count up the actual number of staff members, the actual lines of code produced, and the actual number of defects found during each week and see if these actuals fall within the statistical-control band. If they do, the project is proceeding according to plan. If an actual falls outside the control band, it is out of control. Work is not going according to plan. Something is probably wrong. It is a signal to look for the problem.

Under statistical control, we discover the problems as they happen, every week. They are fresh. The people who were involved are still around. The traditional alternative is to discover such problems in system test. At that point, there is little project schedule left. Some of the people are gone, while others no longer remember very clearly what they did—perhaps months ago—that now turns out to be wrong.

The Construction Phase is where most of the work of software development is accomplished. It is where most of the money is spent. Metrics are important in this phase for two reasons: to provide ade-

quate resources and time for the task, and to confirm that the resources are being utilized according to plan.

The alternative is to fly by the seat of your pants. That does not work in the air. In a cloud, the seat of your pants is deceptive. Instrument-flight instructors preach, "Depend on your instruments, not your pants." We preach a similar message: "Depend on your metrics, not your hopes and fears."

Phase Four: Transition

The Construction Phase ended with system test conducted in-house, resulting in a system state known as initial operational capability. However, the system has not yet operated in a user or customer environment. That is the province of the Transition Phase. This phase oversees the movement of the product into the user environment, which may differ slightly from the in-house environment.

In a broad sense, there are two types of products. The first, the shrink-wrapped product, goes to many customers. As a start, the vendor may conduct a beta test (usually toward the end of the Construction Phase) with a representative selection of those users. The second type of product goes to one customer, often for installation at one site, sometimes at several sites. The customer usually conducts an acceptance test in its own environment.

In general terms, there will be two types of feedback from the user environment:

- Defects will turn up, either due to the new environment or because they were undetected during the in-house tests. The software organization has to correct them. In some cases, it may be possible to hold them for the next release.
- The software may fail to meet some of the needs the users now identify. Again, the software organization may be able to modify the system to accommodate these needs, at least those that it can readily add within the current structure. In other cases, the modification will be added to the list for consideration in the next release.

The amount of effort and schedule time needed by the Transition Phase depends upon the extent to which users encounter new needs and defects. If requirements were carefully captured in the early phases, if users were consulted during analysis and design, if users

tried out operating prototypes, if they viewed operating increments early in the Construction Phase, and if this early feedback resulted in modification of the initial requirements to accommodate needs discovered along the way, then few new needs should turn up in the Transition Phase. At best, however, a software organization will find it difficult to estimate the amount of modification work that will feed back from users. Keeping a record of what happened on previous projects can serve as a guide.

If a project has followed good inspection and testing practices, the system will reach initial operational capability with fewer remaining defects. However, there are metric methods for estimating the number of defects that do remain at this point. Then, as the users operate the system in their own environment, they gradually unearth these defects. They or the system builder correct the defects and revise the estimate of the number still remaining. Using this methodology, it is possible to make a rough estimate of the amount of rework to be expected from defect discovery.

On the whole, however, the core metrics—size, effort, time, productivity, and defect rate—are less applicable in the Transition Phase than in the earlier phases. The best guide is the experience of navigating previous projects through this phase. On that basis, the software organization can allocate a certain amount of time and effort. It can then make modifications and correct defects within that budget while trying to defer more time-consuming modifications to the next release or to the maintenance budget.

Index