Continued from page 1.

it's that this organization developed IT applications that were only about making money. If the company didn't make enough money, at least no one would die from a software defect. (However, if a trading application experienced a major outage, you could practically count the enormous amount of lost revenue by the hour.)

When commercial organizations espouse this philosophy (or lack of one) on quality, other issues might come into play. For example, this could have easily been the philosophy of a company called Ashton-Tate over a decade ago with a product called dBase IV. Remember Ashton-Tate and dBase IV? Time to market was so important that defects suffered severely. While nobody died from the number of defects when the product shipped (although I could be wrong), there was an entity that did die. The company. Word got out about dBase IV being unreliable, and the company collapsed.

Is Quality Really Job One?

This brings us to a subject for IT metrics that is altogether critical — the use of metrics to manage defects and the quality of a system at the time it is placed into service. In fact, about 15 years ago, this is exactly how I wound up involved in the metrics field.

In that past life, I was the group leader for integrated test planning at a major *Fortune* corporation, which was tasked with designing and building a major subsystem under a multiyear, multibillion-dollar project. This project was the Trident II nuclear submarine.

On that ship, defect-free software was crucial. The navigation subsystem that I worked on employed some of the most brilliant scientists I've ever met, designing what was undoubtedly one of the most complex applications in the world. This application basically had to "fly a ship" underwater, without windows, using a combination of inertial navigation, sonar, satellite communication, and gravity mapping tied together with six massive processors running parallel in real time. The stakes were high. For example, if a sub lost its way or even bumped into something in the dark, then a lot of sailors could die. (When I heard about the loss of the Russian submarine Kursk in August, it was particularly painful. Whatever the cause of that disaster, it's clear that quality does matter.)

At that time the goal was to chart the defect rate curves for the software to know whether the project would meet the deadline and operate reliably on that date. To accomplish this, it was vital to optimize the testing strategy across three test facilities to ensure that the system met all its performance and reliability requirements to meet its mission when the deadline came. Using defect tracking and forecasting, the teams executed a strategy that brought the project in successfully. Metrics played a crucial role.

On a side note, we've seen how Hollywood has featured nuclear submarines with movies like *The Hunt for Red October* and *Crimson Tide*. Through these films, many have learned that a single ballistic missile submarine is the third-most formidable "entity" in the world purely in terms of nuclear firepower (after the US and the former Soviet Union). It has to know where it is and where it is going. And it better not crash.

What Defect Graphs Typically Look Like

You may or may not be working on applications for nuclear submarines, but your work is surely crucial to the field that you're in. And along those lines, metrics on software defects can give you a wealth of information to help make better decisions. Metrics told us that dBase IV was an imminent train wreck at the time (we were brought in at the 11th hour). It also told me and my management

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

team that the Trident II submarine navigation subsystem project was "on course" and would operate reliably if we made the right choices along the way.

In last month's ***ITMS***, Jim Heires used the following quote from Walter Shewhart, a pioneer in statistical process control for telephone-manufacturing in the 1930s:

> Measurement is a sampling process designed to tell us something about the universe in which we live that will enable us to predict the future in terms of the past through the establishment of principles or natural laws.

Figure 1 is a graph showing a typical shape of a software defect curve, produced from monthly samples (the defects found and corrected per month) from the start of detailed design through deployment for an IT application.

The purpose of showing this graph in the context of Walter Shewhart's quote is to underscore the phrases *sampling process*, *natural laws*, and *predict the future through the past*.

We have learned some of these natural laws from defect data sampled at companies with IT metrics programs.

One of these laws is the characteristic shape of defects, which you can see rather easily (and even more so if you drew a smoothed curve using a rolling average). This curve reaches a certain peak and then tails down gradually in a pattern known as a Rayleigh curve, named after the British mathematician. This curve was also observed in the work of Peter Norden at IBM and then on software projects from software metrics research by Larry Putnam. These natural laws continue to reveal themselves today on IT projects, including Web development and e-commerce applications.

What This Means to You

The third phrase, *predict the future through the past*, will tell us what it means to you.

If your metrics program included as a minimum the ability to construct defect charts like Figure 1 using your own project metrics, then you could conceivably predict the point in time by which your IT project would be "good enough" for delivery (from a defect standpoint) by use of "trending." Trending is a fancy word for drawing a line through the data points and extrapolating out into the future. When the trend reaches a low enough point, say, 10 or fewer defects per month for example, then you've arrived (or will have arrived).

Basically, this concept uses defect metrics as criteria for time to market. If this is an IT

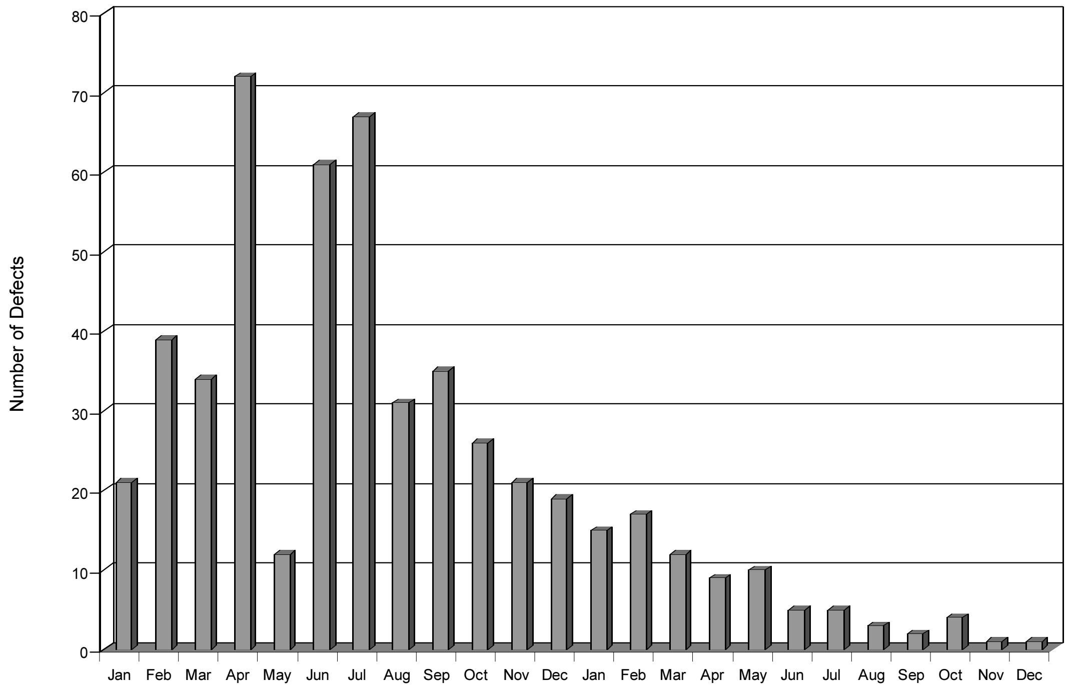

Figure 1 — Typical defect rate curve.

project for internal use such as billing, customer care, or networking, then it's obviously about time to production for an internal customer. But if it's for a commercial database application (or for a nuclear submarine for that matter), then it's for use by an external customer.

End-user experience of the system is, to a certain extent, a function of whether the end user is "feeling" the system at a "high defect rate" part of the curve (that's bad), or at the "low defect rate" part of the curve (that's good).

Ideally, what you want to do is create the necessary mechanisms to produce these curves. They in turn furnish the metrics information to help you anticipate the future, like a navigation aid. This involves a simple practice of assigning a metrics analyst or team of analysts to record the open and closed defects. Some companies also refer to these as trouble tickets, problem/action reports, or open/closed bugs.

You'll want to look for how fast that curve falls. If you have a top-notch inspection and test methodology, then you'll find and fix bugs at a healthy clip, and the curve will most likely drop fairly quickly, accelerating your time to market. If you are ineffective at stomping out the bugs, or if your product is infested with them, then that curve is going to stay high, and life is going to get ugly as the deadline approaches.

And if you decide to deliver the system prematurely, that will be your call. On this matter, I always remember the words of Tomoo Matsubara of Hitachi and the Cutter Consortium: "If we deliver late to the customer, we might hear about it one time, and then it goes away. However, if we deliver a system with poor quality, then we hear about it again, and again, and again."

How to Chart Your Own Defect Curves

You can chart your own defect curves using a defect-tracking application, a metrics tool, or even just Microsoft Excel. Early on, during detailed design, the source of your defect counts might be peer reviews or inspections (i.e., Fagan inspection methodology or similar), or the source may be from self-recorded bugs found during unit testing.

Chances are, however, that most of you out there don't do any defect metrics tracking. If you do, it might only be starting with the system test phase. Often times this seems to occur at about 70% into what is commonly known as the "main build" phase. In the chart shown in Figure 1, that might have been sometime in November. In that example, delivery of the system might have been around the second April, for a total elapsed time of 16 months for the main build. By that time the natural slope of the defect curve on this project was heading down, after an initial peak of the first April/May/June time frame.

You'll want to record both open and closed bugs. In that case you'll have two sets of parallel Rayleigh curves. They will have what is known as a phase lag. This lag is the elapsed time that you typically take to close out a bug or a trouble report from the time it is discovered. If you see that your phase lag is short, that's good. If your phase lag is long, that's bad.

You'll also want to establish a consistent definition of what is a defect. A handy one to start with might be: an error in specification, analysis, design, coding, or testing that leads to a fault in a program where the application's operation departs from the user requirements.

This is a fairly broad definition that includes not only defects from coding, but also defects from the design and specification phases. Some defects are also introduced in regression testing, or from the correction of defects, which sometimes produces new ones.

What's important is that your organization establishes a definition that works to capture at least these categories of defects so that an IT application is delivered that is reliable and meets the requirements of the customer.

Lastly, Not All Defects Are Created Equal

It's apparent that some bugs are worse than others. Typically I encourage testing organizations to classify defects in at least three categories of severity. For a basic tier set, many use Severity 1, Severity 2, and Severity 3 defects. These are in decreasing severity from critical to serious to minor. Critical might be considered showstoppers. Serious might be defined as blatantly wrong answers or outcomes. Minor might include cosmetic or tolerable defects. These three are often adequate in terms of resolution, but I've occasionally seen organizations use up to five classifications.

What you'll want to do is record defects by month and by severity class. These can be recorded in their own Excel rows or columns, with individual charts for each severity class, and for the total. You will see a family of curves like the one shown in Figure 1. There is great value in going down one level of detail using charts by severity class. You'll observe defect density both in terms of *when* they occur and in what category. This will reveal valuable information about the inherent nature of your applications and your design process.

In the latter category, you'll be able to embark on root cause analysis to focus energies on correcting aspects of your process that result in undesirable outcomes. These charts will also reveal time-based trends that enable you to deploy test teams to the right place, at the right time, in order to optimize your strategy and best meet time to market. The data is your friend.

Where We Go from Here

Once you've built the basic mechanisms for your "defect dashboard," you'll have valuable navigation information to guide your actions and decisions in an optimal manner. Life is too short and resources are too scarce for you to take a shotgun blast approach to quality. You'll need to think of a more efficient and surgical approach to make wise use of your precious time and staff.

As you begin to do this, you'll find that it is easy. Once the organization gets the hang of having this metrics data, it will be ready to take the next step — learning the "whys" behind the "whats". The patterns of the data are the "what". The "whys" are the causal relationships that are underlying the data to uncover aspects of your process.

For example, some may want to know the relationship between scope growth and defects rates. Or, what happens to the defect curves if a client asks for a requirements change in excess of 25%? How does that impact time to market? What happens with a staff churn of 20% or more on an IT project? What happens to defects with schedule compression?

These and other questions have answers that are the hidden gold for IT management with the wise interpretation of IT metrics. And, who knows, once you know these answers you too might be able to steer a submarine that doesn't have any windows.

Saving the World, One Project at a Time: Planning by the Numbers

Continued from page 1.

PBN is the concept of using historical project-level metrics to aid the project manager in discovering facts about software projects. Although not as simple as "painting by the numbers," the project manager (with support from a metrics specialist) can use this information for project planning, validation, and risk assessment. In many ways, I do believe that the metrics (or estimation) specialist performs the role of chief memory officer — someone who remembers the past, so the same mistakes can be avoided in the future. [1] As a software development manager, I became interested in software metrics for providing decision support information related to my projects. Software metrics provide many benefits for managing outsourcing contracts, process improvement, productivity benchmarking, and balanced scorecards; however, you are missing one of the greatest benefits if you are not using historical software metrics for making decisions and planning projects.

The analysis of project-level data has taught me some lessons over the years, which I will share. In the following sections I will outline the primary components of the PBN process.

1. Collect, slice, and dice internal historical project data.
2. Normalize the software project data for comparison.
3. Trend the data.

4. Analyze the data to discover facts.
5. Use the analysis results for planning software projects.

Collect, Slice, and Dice the Data

To begin with, data must be collected for internal historical projects. This can be done via a productivity benchmark or by less formal methods. [2,3] However, the data must include the Software Engineering Institute's Capability Maturity Model core measures of size, effort, schedule, and defects. [1] Data consistency is the key to having a good decision support database. To ensure consistency, document the rules in a data dictionary. Also, when building an internal historical project repository, the project data points should be categorized. Allow for enough stratification such that apples-to-apples project comparisons can be made if needed (project type, environment, user organization, development organization, language, etc.).

Next, the metrics specialist needs a "slicer/dicer" with the capability to extract and analyze the data statistically. This can be accomplished using software metrics tools such as SLIM-Metrics, Access, Excel, or other types of databases and statistical analysis tools. Specific project data can be compared to the entire database, specific subsets of data, or both. If you have access to industry data (QSM, International Function Point Users Group, etc.), benchmarking specific project data against industry data is beneficial as an additional decision point. However, the benefit of having internal historical project data cannot be overemphasized.

Normalize the Data

One of my metrics discoveries was the importance of normalizing project-size data for statistical trending. This is not to argue the merits of source lines of code (SLOC) versus function points (FP), since I use both measures. My objective is to describe how I have been using these measures, continuously improving the results as I refined my techniques by using discovery metrics. [7] There is no perfect software sizing measure; therefore, I use the best aspects of both SLOC and FP. In my opinion, this leads to better results, as long as you use consistent counting methods. I primarily use FPs for sizing project estimates. Because FPs reflect the user view of a software project, an FP count can be accurately determined based on the software requirements. As the project requirements change, the FP count can be updated to reflect changes in project size.

I first tried using FPs as the size component for statistically trending historical project data. However, I discovered that there is not a strong convergence of data, unless the project-size values are normalized based on programming languages and code mix. Therefore, I convert FPs to SLOC. This can be done using industry factors such as those provided by Software Productivity Research (Capers Jones) or as illustrated in the September 2000 issue of ***ITMS***, [1] but (preferably) you should use SLOC/FP factors that have been internally calibrated. I have also found that it is important to normalize project size, in relation to the effort required to produce the code (new, modified, reused, tested, etc.), in order to determine the productivity rate to use for a project estimate.

In the early 1980s, Robert Tausworthe of the Jet Propulsion Laboratory (JPL) of the California Institute of Technology determined a relationship between the effort to develop new code and the effort to modify code. Basically, the theory suggests that if the effort to develop a line of new code is taken as unity, then the effort to modify a line of existing code is some fractional value. The values that Tausworthe found on a number of JPL rehosting contracts are shown in Table 1. [5,6] I do not always use the full range of Tausworthe factors, simplifying it most of the time to using factors of 1.0 for new/conversion code and 0.24 for modified/deleted code (0.24 is the average of all of the effort ratios except new code). If the project involves a substantial amount of regression testing, as is the case with commercial-off-the-shelf (COTS) software customization and integration projects, then it is important to also apply the tested ratio of 0.12 to all the unmodified code.

I apply the Tausworthe factors to the SLOC/FP language conversion factors and not to the FPs. This allows a normalized effective source lines of code (ESLOC) comparison for trending a wider range of

Table 1 — Software Project Size Normalization Effort Ratios

This table shows ways in which existing code can be modified. For each type of modification, the ratio of the effort-to-modify to effort-to-code-as-new is given.
Note: The number of lines of code added, changed, and deleted are subsets of the number of lines reused. Therefore, their sum must be less than or equal to the number of reused lines. [5]

Type of Modification	Effort Ratio
New Code: Subject to entire development process.	1.0
Reused: The lines of code in modules that will be reused, but will be modified by additions, changes, and deletions.	0.27
Added: The lines of code to be added to reused modules.	0.53
Changed: The lines of code in the reused modules to be changed. This effort is typically less than the effort to add lines.	0.24
Deleted: The lines of code to be deleted (line by line) from reused modules.	0.15
Removed: The lines of code to be removed in modules or programs as whole entities. Testing must take place to check reused modules that interface with the removed modules.	0.11
Tested: The lines of code from the unmodified but reused modules that required no modifications but still exist and require testing with new and modified software.	0.12

projects (new development, enhancement, and COTS integration) as shown in the scatter charts later in this article. It also allows for additional analysis using FPs that have not been normalized. FPs provide a size measure of the functional software product size for cost-benefit analysis, whereas normalized ESLOC provide a measure of the physical software product size for estimation, risk analysis, and project data trending. I am sure that others have their own views on this matter based on their own experience, just as my views are based on my experience. Just make up your own mind as to what works best for you and what you can substantiate using your own discovery metrics.

Trend the Data

For longer than I care to admit, I used averages for productivity calculations related to software project estimates. Then, as I began benchmarking my estimates against project data trended for size, I noticed a great disparity related to the project productivity averages and the project size trends.

For example, as shown in Figure 2, I selected data for 19 projects from my repository, similar to the project I was estimating. (Note: the charts and values used in this article are for illustrative purposes only and are not related to any specific data set; however, they are based on actual findings.) The average productivity rate for these 19 projects, using the QSM Productivity Index (PI) scale, was 12.9. When I plotted my project estimate on a productivity graph trended for size, I noticed that the average productivity was 33% lower than it should have been for a project of that size. The three projects (A, B, and C) highlighted on Figure 2, of similar size, type, and the same development team, had an average PI of 19.1. Therefore, I used the trended PI of 19.1 for my estimate instead of 12.9. It was still an average, but it was a trended average.

There are many reasons why project size affects productivity. One reason is that there is a smaller percentage of effort allocated to overhead for larger projects than for smaller projects. Larger projects usually take longer, which is also a factor, due to the effort/schedule ratio. However, there is a limit that is reached where bigger projects are no longer better, and there is an optimal effort/schedule ratio, as we learn in the following section.

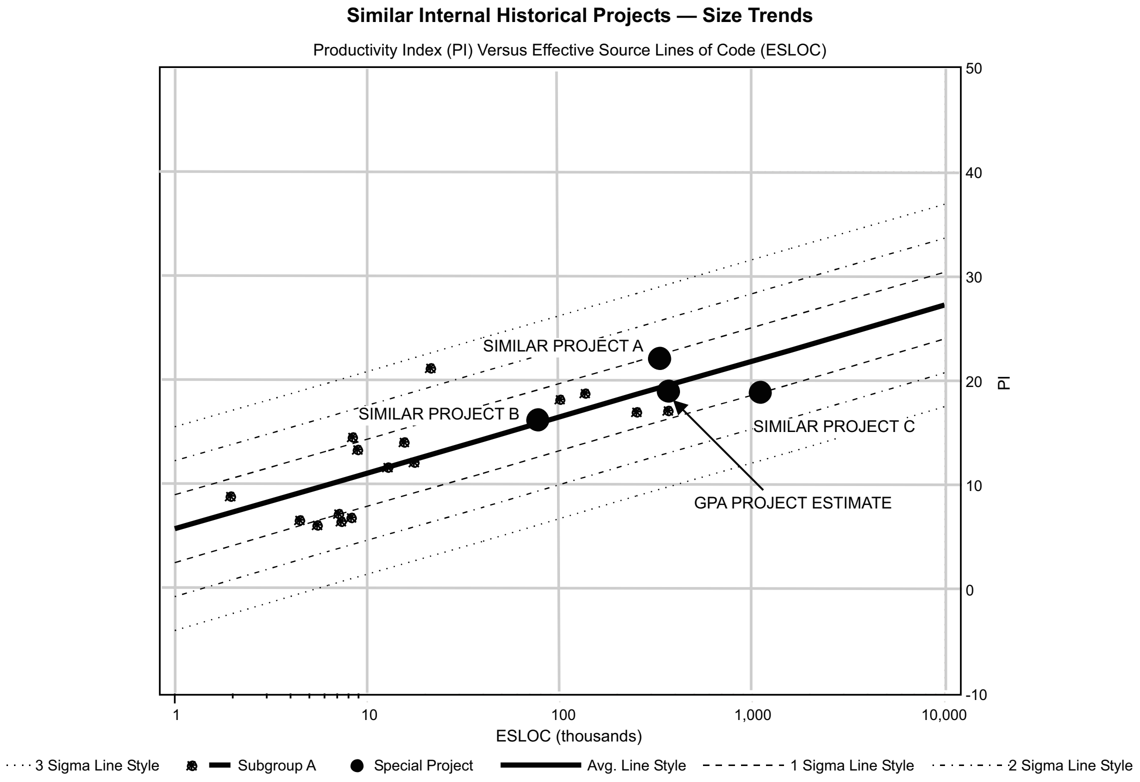

Figure 2 — Using trended data relative to project size.

Analyze to Discover Facts

Data analysis can be very enlightening and exciting as we discover new information. We can discover the answers to such questions as: How can we get more "bang for the buck" from our software projects? How do we increase business value? How do we staff our projects relative to schedule to optimize business value? With regard to the first two questions, one of the ways to do this is to increase productivity. Therefore, we may want to discover the effect that project size has on productivity. This can be analyzed and illustrated as shown in Figure 3.

The trend shown in Figure 3 illustrates the bell-shaped curve I discovered in relation to project size versus productivity. The data is on a log-log scale, which causes the trend lines to appear linear. However, you will notice that the data points form a curve represented by the dashed line that was added to Figure 3 to reflect this tendency. As the size increases, the productivity increases; however, it reaches a size where productivity starts declining. Specifically, prior to the 30K ESLOC mark, the data points consistently trended upward, then they leveled off. After the 150K ESLOC mark, the data points displayed an overall downward tendency. With this analysis, I discovered that the projects sized between 30K and 150K ESLOC provided the best "bang for the buck" and business value. This size range also has the highest convergence of data points within +/-1 standard deviation (SD).

Another fact we may want to look at is the optimal relationship between effort and schedule associated with the projects in the optimum size range. Using the Putnam Manpower Buildup Index (MBI) scale, where MBI = Total Effort/(Development Time)^3 [5, 6], Figure 4 illustrates how we can analyze the appropriate ratio of effort to schedule. The MBI is a measure of schedule compression where projects with lower MBIs are less compressed than projects with higher MBIs, and less compression equates to a lower cost and higher quality. In this case, using internal project data, the optimum MBI is in the 4-6 range.

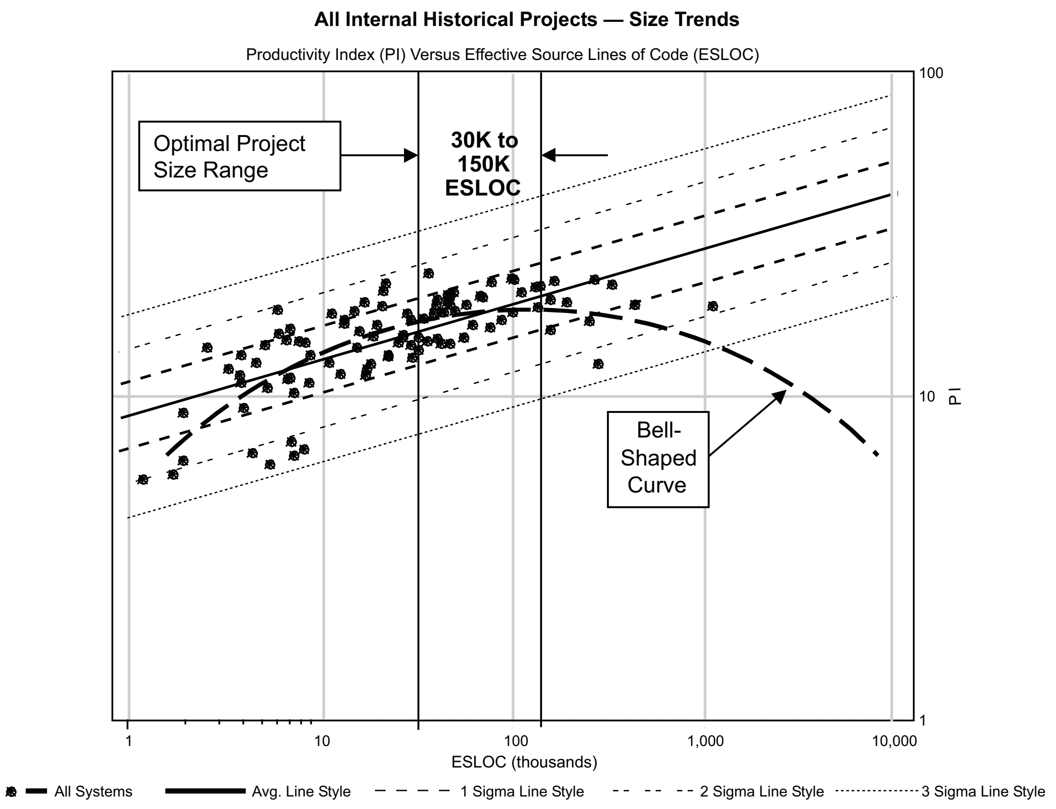

Figure 3 — The bell-shaped curve related to project size and productivity.

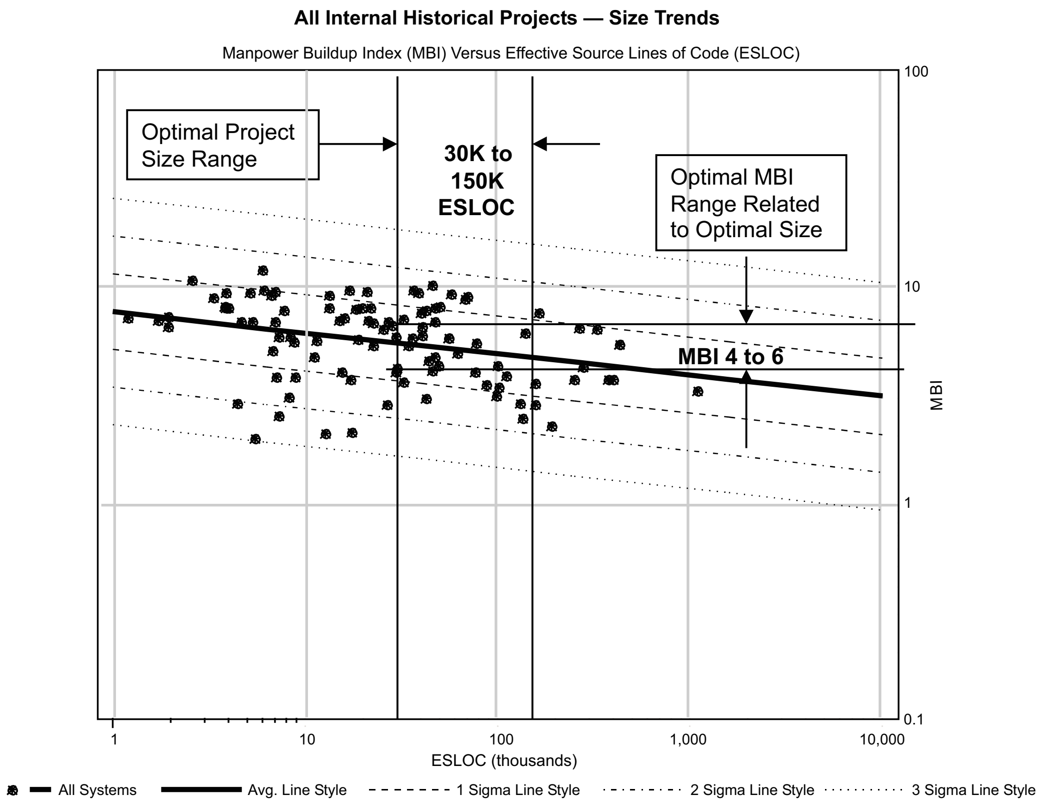

Figure 4 — Analyzing optimal effort/schedule ratio using Putnam MBI scale.

Use the Results for Planning

The primary benefit of PBN is for project planning. We can put to use the information we learned during data analysis for developing software release strategies, determining whether estimates are reasonable, and assessing risk. The PBN process supports the use of trended data for this purpose. When I prepare a project estimate for an internal project or validate a vendor estimate, I want to see how it compares to the historical project data trends. For example, I had a client tell me that my estimate seemed high for a 250K ESLOC project; however, when I plotted the data point on a trend chart, it was within the ballpark for a project of this size. In fact, the cost was right in the middle of the trend. Therefore, the client was much more inclined to accept my estimate. This concept could also apply to a vendor estimate — if the vendor cost was in the high range on the chart, it could be considered unreasonable. Estimate validations and risk assessments are done in much the same way. The following examples illustrate how the PBN concept is used for project planning. Example 1 illustrates planning software releases; Example 2 illustrates estimate validations and risk assessments.

Example 1: Project Planning — Strategy Analysis

Figures 5 and 6 compare monthly versus bimonthly release strategies for an internal client. These illustrations were used to encourage the client to consider alternatives that would provide greater business value. Table 2 documents the results of this analysis, which indicate that for the same yearly cost (effort) and a 35% longer schedule per release, the yearly code output can be increased by 267% and the quality by 75%, just by changing from monthly releases to bimonthly releases. The quality improvement is significant because it reduces the resources required for error correction between releases. Notice that the bimonthly release strategy uses a project size within the optimal size range illustrated in Figure 3, and that the MBI is in the optimal MBI range illustrated in Figure 4.

Example 2: Project Planning — Estimate Validation and Risk Assessment

This example is divided into two parts, illustrating how estimate validations and risk assessments were performed for the SRO and GPA projects (SRO and GPA are

Table 2 — Release Strategy Analysis (See Figures 5 and 6)

Release Strategy Analysis	
Assumptions — Monthly Releases (Figure 5)	▪ Monthly releases are based on an average of historical small releases — i.e., 5.2 months duration; 6,000 ESLOC; 6.7 MBI; 47 Staff Months (SM) of effort; 0.63 mean time to defect (MTTD); including the planning, analysis, and main build phases. ▪ Illustrates 12 monthly releases that complete during a calendar year.
Assumptions — Bimonthly Releases (Figure 6)	▪ Bimonthly releases are based on an average of historical larger releases — i.e., 7 months duration; 44,000 ESLOC; 5.4 MBI; 95 SM of effort; 1.1 MTTD; including the planning, analysis, and main build phases. ▪ This illustrates 6 bimonthly releases during a calendar year.
Conclusion	6 bimonthly releases during a calendar year could produce 267% more code than 12 monthly releases, same yearly effort/cost, 35% longer schedule per release, and 75% better quality: ▪ 12 monthly releases produce 72,000 ESLOC and require 564 SM of effort. ▪ 6 bimonthly releases produce 264,000 ESLOC and require 564 SM of effort. ▪ Individual bimonthly releases are 7 months duration each, versus 5.2 months each for monthly releases. ▪ MTTD (at deployment) for each bimonthly release is one defect every 1.1 days, versus 1 defect every 0.63 days (1.6 defects per day) for each monthly release.