IT METRICS STRATEGIES

Helping Management Measure Software and Processes and their Business Value

Can Function Points Be Used to Size Real-Time, Scientific, Object-Oriented, Extreme Programmed, or Web-Based Software?

by Carol A. Dekkers and Shari Cartwright

It's been going on for years: the notion that every time a new technology, methodology, or approach comes on the software scene, we need new metrics to manage the software project. Although it is definitely true that new technologies bring differences in productivity and quality, there are some metrics that never go out of style — because the fundamental principles on which they are based have not changed. Fundamental measures such as the size of the logical user requirements, the work effort metrics, and defects can all be used to report the productivity or quality of the resultant software product, regardless of whether it was built using scripts, Java, C++, HTML, COBOL, Assembler, or any other language. It follows that the amount of work effort will vary widely depending on the project type, complexity, development language, work breakdown structure, and many other factors, but why would size of functional user requirements vary? Functional size is independent of all of these technical, quality, design, and other "nonfunctional" requirements; this is confirmed by the ISO/IEC 14143-1 (1998 Concepts of Functional Size Measurement standard). Requirements manifested in a Web-based application, for example, are still requirements, just as requirements implemented in a data warehouse application are requirements. As such, a

Continued on page 2.

Build In Quality

by Lawrence H. Putnam and Ware Myers

"Getting people to do better all the worthwhile things they ought to be doing anyway." That is the language with which Philip B. Crosby began his now famous book *Quality is Free*: *The Art of Making Quality Certain* (McGraw-Hill, 1979). "People" includes management, he went on to say, but it is up to the professionals in a field to instruct management about this portion of the management job. What can management do to make software quality more certain?

It can provide an adequate amount of schedule time for software development, sufficient staff hours, and the productivity to enhance that time and effort. It can provide a software development process in which the key actions that ensure quality are given early emphasis. You can see that providing "adequate" amounts of these ingredients is just another way of saying that quality rests on the ability to judiciously employ five metrics for time, effort, size, productivity, and quality.

Continued on page 6.

july 2001 vol. VII, no. 7

executive summary

Fast-changing times demand adaptability. With ever-tighter deadlines and the constant newness of technical challenges, it's often difficult to size projects. Even as times change, this fundamental problem remains. Many want to know whether sizing metrics that we learned in the past, such as function points, are adaptable to new and different environments.

Carol Dekkers and Shari Cartwright say yes. In the first article, the authors take on the issue of sizing real-time, scientific, object-oriented, Extreme Programming, and Web applications using function points. They express the notion that functional size measurement can be seen as independent of the changing software development landscape, and they address what function points were intended and not intended to do.

Next comes an article by Larry Putnam and Ware Myers on quality and reliability. They talk about "designing in" quality and related attributes such as expandability, integrity, and reusability at that critical front end — you can't "test those in" at the back end. We need to think more about the critical early phases when the blueprints are drawn and when the project is scoped. And for this to be done right, metrics (and project sizing) must play a critical role in these up-front planning phases.

In the third article, Jim Greene describes the challenges inherent in breaking a large project into smaller subsystems and integrating them into a larger product release. Here there are two challenges: chunking the release into the right-sized subsystem pieces and tying them together during a final integration. Monitoring the reliability patterns and defect rates during the project will alert you to any early warning signs and help you steer your course to a successful conclusion.

These articles are intended to illustrate the interdependence of size and quality issues. Collectively, we hope to convey the idea that improving dimensions such as size and requirements planning can positively influence other dimensions — quality and reliability.

Michael Mah, Editor

CUTTER CONSORTIUM

Continued from page 1.

software's functional size is not influenced by the technical or design considerations of today's software development.

Types of Software Requirements

Every new method of articulating software requirements and every new development method introduces its own unique set of terms, acronyms, and nuances. When looking at any type of software or development methodology, it's important to remember that there are three major types of software requirements:

1. The functional or logical user requirements (*what* the software must do)
2. The nonfunctional requirements (*how* the software must perform, including "ity" characteristics such as portability, reliability, etc.)
3. Technical and design constraints (*how* the software will be built)

The first two types of requirements are within the user domain and pertain to what the software will do and how it will function. The third type is specific to the development environment and involves choices in development methodology, work breakdown structure, development language, skills, geography, and so on. Function point analysis (FPA) addresses the first type of software requirements: the functional user requirements. If the value adjustment factor (VAF) is used to adjust the function point count, it provides some coverage of the nonfunctional requirements, but not all. (Nonfunctional requirements such as reliability, portability, and security, as outlined by the ISO standard suite ISO/IEC 9126 — Characteristics of Quality — are not covered by VAF.)

Functional Size Versus Work Effort Versus Productivity

What does this preamble really mean? It means that FPA, which is a structured method of sizing software based on its functional user requirements (ignoring VAF, which takes into account additional, potentially nonfunctional requirements), is applicable for sizing software regardless of how it is developed or implemented. The relationship between effort, functional size, and productivity can be summarized by the following equations:

Functional Size

= Logical (functional) size of software

= The independent variable (independent of implementation or design methods)

= Is a function of (*f*) the functional user requirements

Work Effort

= Project hours

= A dependent variable

= Is a function of (*f*) many constraints, including functional size, nonfunctional requirements (security, portability, etc.), technical and design considerations (such as development language, tools, methodology, and hardware and software configuration), and other project constraints (such as work breakdown structure)

Productivity

= Work effort/functional size

= A dependent ratio

= Is a function of (*f*) all of the same constraints as work effort (recall from mathematics that a dependent variable mixed with an independent variable yields a dependent result)

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: Tel: +1 781 641 9876 or, within North America, +1 800 492 1650, Fax: +1 781 648 1950 or, within North America, + 1 800 888 1816, E-mail: service@cutter.com, Web site: www.cutter.com/consortium/.
Editor: Michael Mah. Publisher: Karen Fine Coburn. Group Publisher: Kara Lovering, Tel: +1 781 641 5126, E-mail: klovering@cutter.com. Production Editor: Pamela Shalit, Tel: +1 781 641 5116. Subscriptions: $485 per year; $545 outside North America. ©2001 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.

Stated in another way, a software's functional size (for a given purpose) is independent of the type of project, type of system, difficulty to build, construction terminology (based on methodology), technical constraints, or construction methods (such as programming language). Functional size is similar (although far from a perfect analogy) to the square foot size of a floor plan in building construction. In software, our floor plan is the set of functional user requirements, and function points simply measure its size — nothing more, nothing less.

For example, let's consider the software requirements to provide the users with a facility in the software that enables them to view, add, modify, and delete data (it could be a screen, a Web page, etc.). These requirements do not change because the programming language or the operating system changes.

Some Common Challenges

Consider, for example, what happens when we are exploring the functional requirements for a new piece of software, and there are project "novelties" involved (i.e., new technology, tools or methodologies will be used on the project). The idea of using FPA to size the functional requirements as part of the cost or effort estimation process is often met with challenges or outright rejection by developers. Typically, the developers who reject function points as a valid sizing measure for these requirements fall into the following categories:

1. Developers who want to directly correlate function points with work effort and want a simple relationship, such as size x constant = work effort. Although the relationship is much more complex due to other factors (as shown previously) and the work effort equations may require change to accommodate nonfunctional or technical differences, these developers want to use the same work effort models, and they often want to increase the "size" variable to accommodate these differences, rather than revise the effort equation.

2. Developers who have not been trained adequately in the principles of FPA and anticipate that, because their software does not exhibit traditional "transaction-based" behavior, FPA will not adequately credit their application with a large enough count. (These developers usually unnecessarily reject FPA for anything other than straight transaction-based data processing because of the terminology presented in the method.)

3. Developers who need a micro-level or method-dependent sizing model to arrive at task-level estimates (such as how much time it will take to program this Web page or develop this Java script). Again, an analogy might be helpful: By remembering that function points are similar to the square footage of a room or building, it is easy to see how it can be difficult to accurately predict micro-task-level effort that depends on the technology. This is similar to trying to use square feet to estimate the amount of time to do the electrical work in an area of a building as a standalone task or to estimate the cost and effort to build a kitchen area independent of the rest of the construction. Almost all estimating models based on functional size are intended to provide overall software development estimates, not task-level and development method-specific estimates.

In any case, it is critical to obtain certified FP training (by a certified FP specialist instructor) from a vendor who understands the overall picture and can clearly explain the differences between functional size issues and the work effort issues. This will ease the confusion and illustrate how valuable the results of FPA can be when sizing the functional requirements of many types of software. Counting function points in new technology environments often is a simple matter of translating the terminology and shifting your thinking — once you know the principles of the method and can count one type of software, it is easy to apply FPA to any set of functional user requirements.

Function Points Applied to Web-Based Software

One of the most common questions of Web-based developers is "What counts? Do we count hyperlinks, screen refreshes, e-commerce, mouse clicks, navigation? The IFPUG [International Function Point Users Group] manual does not specifically state anything about any of these functions."

The answer is that every logical user requirement (elementary function) counts, whether it is implemented as Web-based software or another way. Here are some examples of the types of functions that are typically implemented in Web-based software (all of these are related to functional user requirements):

- E-commerce functions such as "calculate total of shopping cart items," "show discounted amount," etc. (external outputs[1])
- Display of retrieved user data back to a browser screen such as stored credit card information (external output or external query, depending on the logic involved in the process)
- A retrieval and display of data from another Web site within an intranet (external query)
- Navigation to an external Web site that includes sending of data to the other site (external query or external output)
- Security validation of a returning user (external query)
- Sign-up of a new user (external input)
- The logical entities to retain data, either within or external to the application (internal logical files or external interface files)
- A drop-down list that contains values read from a logical file (external query)

Function Points for Real-Time Software

A new set of issues arises with real-time applications because of the nature of real-time systems, which tend to have extensive background processing that may or may not directly affect the foreground processing. Many times, people confuse functionality with technical implementation.

Real-time applications can be (among other things):

- Interrupt-, poll-, or on-demand-driven
- Synchronous or asynchronous
- Electronic data interchange (EDI)

[1]Note that these examples are external outputs because their primary intent is to present data outside the boundary of the application, and at least one calculation is involved in the process. (External queries cannot include calculations.)

The way in which data comes into and goes out of the application is a technical implementation issue (the how of receiving and sending data). For example, interrupt-driven applications receive data that must be processed immediately; polling simply goes looking for the data at timed intervals. FPA is concerned with what data is coming into the application, not how the software has been implemented to get it there.

Regardless of how the incoming data gets into the application, it is handled as an external input as long as the data is stored or has some control aspect to it. Imagine an e-mail system that checks for incoming e-mails either constantly, timed, or when the user selects the send/receive button. These are all examples of receiving or storing data from an e-mail message, which would be typically counted as an external input. The interrupt that displays a window indicating that new mail has arrived and asks whether the user wants to read it now is an example of an interrupt notification to the user that would be counted (as an external query or an external output).

Synchronous versus asynchronous processing is, again, a technical issue. Data coming in or going out of the application is counted as either an external input or external output, regardless of whether the application has to wait for some response or can continue other processing until the response comes back. The key to determining the type of function (external input, output, or query) lies in the primary intent of the function. If the primary intent of the function is to receive and store data into the application boundary, or to control the processing, then the function is classified as an external input. If the primary intent is to present data to a user, then the function is classified as an external output or external query. See the IFPUG *Counting Practices Manual 4.1* (1999) for specific details on how to count and classify requirements into their function point functions.

EDI is a common messaging system between companies. Experienced counters know that any one particular EDI message may have several different meanings, so when new messages are received by the application or there are modifications to an existing message, the counter will examine each meaning for the message and determine which ones

were affected. For example, a "417 message" could be a shipping-billing message. Depending on the set of records in the message, the 417 could be a new bill, a modified bill, a canceled bill, a deleted bill, etc. This is a perfect example of where an inexperienced FP counter might misinterpret the requirements by saying "We are receiving one new message" and thinking that it represents one new external input. In reality, the single physical input stream (one new message) is providing multiple external input functions.

It is critical to remember that it's the logical functions that count in function points — not the physical implementation, regardless of the type of software or hardware that is used.

Scientific, Object-Oriented, and Extreme Programmed Software

Because function points are a measure of functional size, there are aspects of various types of software that are important to the work effort estimation but are not reflected in the FP size. For example, scientific software can be complex, with algorithms that will affect the overall work effort; however, the algorithmic complexity is addressed with the nonfunctional components of many software estimating models.

Object-oriented software development does not change the functional user requirements of software; the requirements may be documented and articulated using different conventions and terminology — but they are functional user requirements all the same. As such, these functional requirements can be sized using function points. The same logic applies to software developed using rapid application development, iterative, and Extreme Programming approaches. The thing to remember is the three types of software requirements outlined at the beginning of the article: the functional user requirements, the nonfunctional requirements, and the technical and design constraints.

Function points were never designed to address the nonfunctional aspects of the software requirements (except for those addressed by the general systems characteristics when companies use VAF), nor the technical or design constraints (which pertain to how the software will be built). They were, and are, intended to address and size the functional user requirements for software. Period.

Conclusion

Hopefully, readers can now see that sizing Web-based and real-time software can be done quickly and objectively using FPA. Often, these types of software result in larger applications from an FP point of view, simply because they are more function-rich than older applications — older applications might have included the logical requirements for the same set of functionality, however.

About the Authors

Carol A. Dekkers is the president of Quality Plus Technologies, Inc., a progressive management consulting firm whose specialty is training companies to become proficient and successful with software measurement and function points. Ms. Dekkers is an acknowledged measurement expert, a project editor for the ISO Functional Size Measurement project, an author, consultant, instructor, and the host of the weekly *Quality Plus e-Talk* radio show. She is past president of the IFPUG board of directors and currently serves in leadership positions for the Project Management Institute's Metrics Specific Interest Group, the Quality Assurance Institute Conference Advisory Board, and the American Society for Quality Software Division. She can be reached at dekkers@qualityplustech.com.

Shari Cartwright is a senior consultant with Quality Plus Technologies, Inc. She has been involved in software development and function point-based software measurement for her entire career. She has counted thousands of function points as a certified function point specialist, spanning myriad application domains and industries.
Ms. Cartwright has been an active participant on the Center of Research in Metrics function point listserv and is emerging as one of Quality Plus Technologies' most requested FP experts. She can be reached at slcartwright@earthlink.net.

© 2001 Cutter Information Corp. For subscriptions call +1 800 492 1650 or +1 781 641 9876

Build In Quality

Continued from page 1.

Quality Is a Positive Concept

Quality is the positive side of the quality-defect continuum. Obviously, a product doesn't have quality if defects overwhelm it. Beyond the absence of defects, however, quality is getting the correct attributes into the product — not just those that users want, but those they actually need. It also restricts these attributes to those users can afford. The marketplace cannot sustain a high price for a large number of little-needed attributes. Just how large is "large" is often a matter of judgment by the stakeholders concerned with the product in question.

What we know is that we want to incorporate in our product planning such qualities as expandability, flexibility, integrity, interoperability, maintainability, portability, reusability, resilience, and usability. These qualities cannot, in general, be "inspected in" or "tested in." They must be "designed in," or more specifically, brought in during the early phases of development: requirements capture, analysis, architecting, and design. During this design in process, these qualities must be weighed against users' needs in an iterative feedback loop.

We know further that it takes time to think through these steps, to implement some of them in iterations, and to get stakeholder concurrence in incorporating the winners in the product. Therefore, it is important to provide this time, and that gets us back to metrics. We need to plan for adequate development time. We also need to plan to employ a development process that allows for these activities.

What we are trying to do is relate core metrics to the achievement of positive quality, something beyond the absence of defects.

Identifying Quality in the First Phase

The achievement of quality begins in the first phase: the feasibility study. A division of the lifecycle into four phases is shown in Figure 1. The ultimate absence of quality is a project that fails — a system that never gets built. It is in this phase that we establish the boundaries of the system and take the first stab at establishing the requirements within those boundaries. Setting these requirements is analogous to characterizing the quality attributes. So it's important that the main concepts underlying the requirements be established in this first phase. They can be filled out further in the next phase: functional design.

It follows logically that it is important to have a first phase, at least when the project is headed into a new area or new work. Project continuation or project activity in an established field does not require this first phase, or not much of one.

If we need a first phase, it follows further that management allocate enough time to it to establish the main concepts. There are basically two:

1. Identification of risks of magnitude great enough to bring the successful completion of the system into question. Risks of this magnitude are usually few, if any, but the first phase must explore them to the point at which the stakeholders see that they can be surmounted.
2. Categorization of requirements and architecture to the degree necessary to assure the stakeholders that the project falls within time and cost parameters they can afford. These initial parameters may vary by several hundred percent from the eventual "bid."

Metrics plays a role in the first phase in this sense. Someone has to provide some calendar time, experienced people, effort in the sense of person-months, and the funding to support this phase. These elements are metrics or depend on metrics. The funding source has to appreciate the role that metrics plays in the first phase. Moreover, aside from the funding the first phase takes, it also takes some time. That time delay is getting down to the business of producing main build code. The funding source has to be willing to accept this delay. The tradeoff is that the main build goes better after a good first phase.

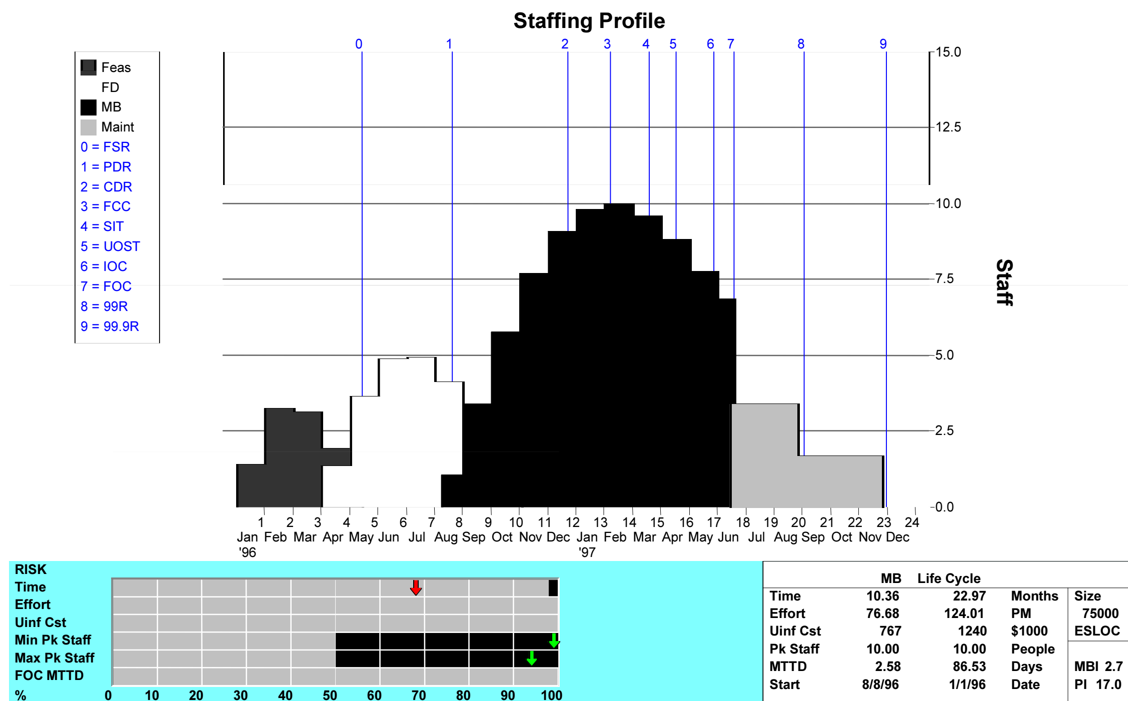

Figure 1 — The software lifecycle is typically divided into four phases, shown here as feasibility study (Feas), functional design (FD), main build construction (MB), and maintain and change (Maint). Software organizations name these phases differently, but they usually perform comparable functions.

That presents the problem of how much time, effort, and cost to allow the first phase. As soon as we can make a very rough estimate of the size of the proposed system, we can make an equally rough estimate — by employing macro-estimating tools — of the time and effort of the main build. Then the rules of thumb for estimating the feasibility study are:

- Schedule: about one-quarter of the main build schedule
- Effort: about 5%-10% of the main build effort

These estimates are uncertain to the same extent that the main build estimates are uncertain at this early point. They are also uncertain because the rules of thumb are rough. In each practical situation, they need to be modified by experienced judgment — for example, how much of a feasibility phase does this project need? But they are something to go on. They also undergird the reality that this work (establishing feasibility) needs to be done.

Continuing into the Second Phase

The identification of quality attributes continues in the second phase: functional design (also called high-level or architectural design). From a quality standpoint, we do three things in this phase:

1. Develop the requirements to the extent that it takes to establish the architecture baseline of the system. This means that we further refine the requirements, certainly all the novel ones and those necessary to establish the main features of the architecture design. We may defer a few detail requirements to the early part of the main build construction phase.
2. Identify the areas in which risks affect estimating and investigate these uncertainties to the extent necessary to prepare the estimate.
3. Carry the architecture design to the point of being able to project the project plan and estimate the cost of carrying it out to the precision level of a business bid.

Basically, to get quality, you have to: know what you are going to do; know that doing it

will not run you into unanticipated risks; have enough information to plan the project; and schedule and staff the plan to the level it takes to get quality.

Again, this second phase takes time and effort at a productivity level — and these are three of the core metrics. By the beginning of the functional design phase, we have reached a better estimate of the project size — the fourth core metric. (Defects are the fifth.) We estimate the time and effort of this phase with these rules of thumb:

- Schedule: about 30%-35% of the estimated main build schedule
- Effort: about 20% of the estimated main build effort

These estimates are more accurate than the first phase estimate because by the end of the first phase we have a better size estimate. Still, the longer you can delay making this estimate, the less the uncertainty of the size estimate and the better the functional design phase estimates will be.

Solidify Quality in the Third Phase

The next phase is the main build. It is where most of the time and effort is spent and where the developers build in the quality that the first two phases planned. It goes better if those involved in the first two phases have done their work — specified what is needed, identified the risks, and estimated and bid enough time and effort to do a quality main build. However, there is still plenty to do in this phase, including detailed design, code, criteria, reviews, test plan, unit test, defect correction, integration, and system test.

One aspect of quality is reliability: the avoiding of defects (or, all too often, finding and correcting them). Development organizations can minimize the occurrence of errors by providing enough staff (effort) with enough time to work carefully. They can find errors by inspecting design documents and code as work progresses. They can prepare test plans and test code early, so that when unit code becomes available, they can test it.

The other aspect of quality is the positive one — meeting the real needs of the users. These needs are expressed in one way in the specifications and related documents, but this has to be turned into criteria tied to each unit of work. In iterative development, for instance, before beginning each increment, project leaders set the criteria the increment is to achieve. A review at the conclusion of the iteration considers whether the criteria have been met.

The principle at work here, both in unit testing and in criteria reviews, is to catch deficiencies — whether defects or lack of some quality attribute — early in development. At these early points, a project still has schedule time ahead of it; it is still staffed to provide the effort needed to instill quality. At a slightly later point in development — integration testing — testers can find the defects that turn up when code units try to work together. The overriding point is this: don’t wait for a system test to find code defects and lack of needed quality. At that point, the schedule is near its end, effort is scarce, and funds have been exhausted. You’re headed down the river toward the falls, and you no longer have the two big paddles — time and effort.

The Fourth Phase — The Last Opportunity to Add Quality

There is one more phase: settling into operation. Here, the emphasis is on getting the software to work in the users’ environment. This environment may be somewhat different from the system-test environment. Users may subject the software to unusual operations not covered by the test plan. Project folks have one last opportunity to add quality to the current release.

The Quality Load Overwhelms Managers

“Of course, there is nothing new about the need to involve senior managers in quality discussions,” conclude C.K. Prahalad and M.S. Krishnan,[1] two business professors at the University of Michigan. In the body of their article, however, they impose a tremendous workload on managers. To give you a rough idea, we pored through the article a second time and counted 44 times where they insisted that management input was needed. We don’t disagree with the need; we just suspect that mere human beings,

[1] Prahalad, C.K., and M.S. Krishnan. “The New Meaning of Quality in the Information Age.” *Harvard Business Review*, September/October 1999.

short of a Napoleon or a Jack Welch (CEO of GE), couldn't meet these demands. What to do?

What we have to do is organize. That is the secret of the past 200 years following the Industrial Revolution, indeed, of the last two millenniums of military organization. What exceeds the grasp of one person can be grasped by an organization of many men and women, dividing the workload and executing it according to a process.

As applied to software development, what does that mean? It means a process, as implied by the four phases we have outlined in this article. It means a modeling language (the software equivalent of engineering drawings) so that people can build on each other's work. It means tools, so that the tasks computers can do better than people can be relegated to the tools. It means metrics, especially the core metrics, so people can effectively plan and execute projects. Bringing these four elements into play enables mere human beings to cope successfully with the complexities of software development.

Metrics Underlie Quality

As a matter of common sense, we observe that it takes time and effort at a productivity level to achieve product functionality at a quality level. Therefore, to achieve quality, we have to provide (at a minimum) this time and effort. We can do it still better if we can provide this time and effort at a high productivity level. In other words, we have to provide metrics appropriate to achieve the quality goal.

Posters on the wall saying "Quality First" don't provide this time, effort, and productivity. Managers making occasional speeches about quality don't provide these metrics. Only metrics can tell you when you are providing the right amount of time, effort, and productivity. If you are not providing them, your posters and speeches will ring hollow, as the daily experiences of Dilbert's pointy-haired boss reveal.

Spend Enough Time on the Front End

The software process consists of a series of overlapping and iterative phases. Developers can accomplish the tasks of each phase more successfully if management is in a position to provide the time and effort each phase properly takes. In particular, a software organization can estimate the time and effort the main build will take more accurately if the early phase team can reach a good estimate of the system size. Reaching that good estimate, in turn, takes time and effort, for which management has to make allowance — itself another estimate.

About the Authors

Ware Myers has a bachelor's degree in engineering from Case Institute of Technology and a master's in management from the University of Southern California. For a decade, he was an extension instructor in personnel management and training, and, for another decade, he was a lecturer in engineering organization and administration at the University of California at Los Angeles. Along the way, he put in stints as a safety engineer, sales engineer, Labor Department representative, naval officer, training officer, vocational advisor, production control manager, systems analyst, drafting instructor, personnel director, engineering writer, contributing editor, and writer of software books. As a production control manager, he learned that estimates often fail to work out as hoped. So when he first heard Larry Putnam explain his views of software estimating in 1977, he applauded their merit in the form of a long article in *Computer* magazine. That led to a collaboration on three books and many articles and columns. Mr. Myers can be reached at myersware@cs.com.

Lawrence H. Putnam has more than 25 years of research experience in software metrics, measurement, estimating, and control. He founded Quantitative Software Management in 1978 and continues today as its president. His most recent work employs statistical quality control to gain effective control of in-progress projects and measure the commercial benefits of process improvement. Mr. Putnam has a bachelor's degree from the US Military Academy at West Point and a master's degree in physics from the US Naval Postgraduate School. He can be reached at larry_putnam_sr@qsm.com.

© 2001 Cutter Information Corp. For subscriptions call +1 800 492 1650 or +1 781 641 9876

Case Study: Using Metrics to Ensure High Reliability of Software Releases

by Jim Greene

Software projects are notorious for being late, over budget, and delivered with poor reliability. In the past this has often been due to ambitious attempts to take on the construction of large amounts of functionality within very aggressive market deadlines. Increasingly, one solution involves partitioning large release projects into smaller subsystems, each developed by separate smaller teams. Then the subsystems are integrated and validated to provide the final product release.

An example is found in mobile telecommunication releases. Here, a "typical" release consists of at least three subsystems, all software intensive:

1. Transceiver
2. Transceiver control
3. Control center

The challenge is to estimate, plan, and manage the development of these distributed subsystems and the final full-system release. An overriding concern is to deliver the release with confidence in high software reliability. Customers for the release are motivated to monitor the software reliability as development progresses to safeguard their business interests.

These complex software-intensive developments are now common in defense, telecommunications, air traffic control, and space. They are recognizable because of their sheer scale. In these developments, a release is often developed over two to four years and costs in excess of 200-300 person-years of development effort with costs in the range of US $30-$45 million. Features define the content of a release. The features are realized by mapping the requirements across the subsystems. Normally this involves building on a large existing software base made up of the existing subsystems that are modified and extended to engineer the new release.

The volume of features in turn drives the resultant size of each major subsystem. At this point, a fundamental tension arises between the promised deadline for the release and the desired number of features. It's reasonable to say that within a fixed deadline there exists a largest possible size for the project, where the team is able to deliver within the time constraint and at a high level of quality. That being said, at one end of the spectrum lies a solution where the volume of features is modest enough that the project easily meets its deadline and perhaps has some slack. Because the promised scope is manageable and realistic, there is enough time for adequate testing of the system so that when deployed, the release has the high reliability that's needed when placed into service.

At the other end of the spectrum is a scenario where the amount of functionality attempted for the release had crossed the limits of what can be done in that time frame to meet a high level of quality and reliability. In this case, the natural tendency of the release is to move to the right on the time line against or past the desired delivery date. Furthermore, the volume of defects compared to the first scenario is likely much higher. More defects have to be found and fixed within the release deadline for the system to have a high likelihood of meeting its mission requirements when placed into service.

Therefore, managing the volume of promised features within each subsystem is a substantial objective of successful project negotiations. If these negotiations are done well, then the shared objective of both management and the developers in achieving high reliability will be met.

Because of their very nature, these complex software projects must achieve high reliability. Software defects arise in the final release validation, not only from the new software but also due to latent defects in the existing code. Substantial regression tests are run to check out the existing software base, which may amount to millions of statements. Requirement changes are common in these developments because of the long lead times involved. This adds to the complexity both

at the individual subsystem level and the release level.

These complex developments represent major management challenges to both developers and purchasers. Frequently, the purchaser's competitive position depends on delivery (time to market) of all the new release features by the scheduled date, within budget and with high reliability.

Figure 2 illustrates the specification and development of a release where major subsystems are developed separately and then integrated and validated.

Background to Case Study

The case study is from the point of view of the purchaser who requires software planning and progress metrics as specified under a formal commercial contract with the supplier. This contract requires the supplier to provide metrics for high-level development estimates and plans, ongoing progress metrics, and delivery acceptance criteria based on quantified mean-time-to-defect (MTTD) metrics.

The project estimate is first used to check that the schedules proposed for each subsystem and the release are realistic and that each development represents value for money in terms of volume of functionality within the target deadline and the budgeted cost. Once accepted, the baseline plan provides the basis to monitor progress.

The contract requires the supplier to provide progress metrics every two weeks. In the past, the monitoring of progress against a baseline plan in the traditional sense would have included regular metrics reporting on only two dimensions: cost and schedule. Latest research illustrates that these two taken alone are inadequate in that they only describe the time and effort resources that have been expended.

More sophisticated progress monitoring techniques employ metrics reporting on two additional dimensions that open the possibility of advanced forecasting techniques. These dimensions are the amount of functionality built over time and the number of defects reported once testing has begun.

Defect metrics consist of software defect numbers and classes. This defect data is supplied once individual modules within each subsystem are formally handed over by programmers for integration. Additional progress data relates to staff numbers, module status and code production, milestones, and integration and validation test cases.

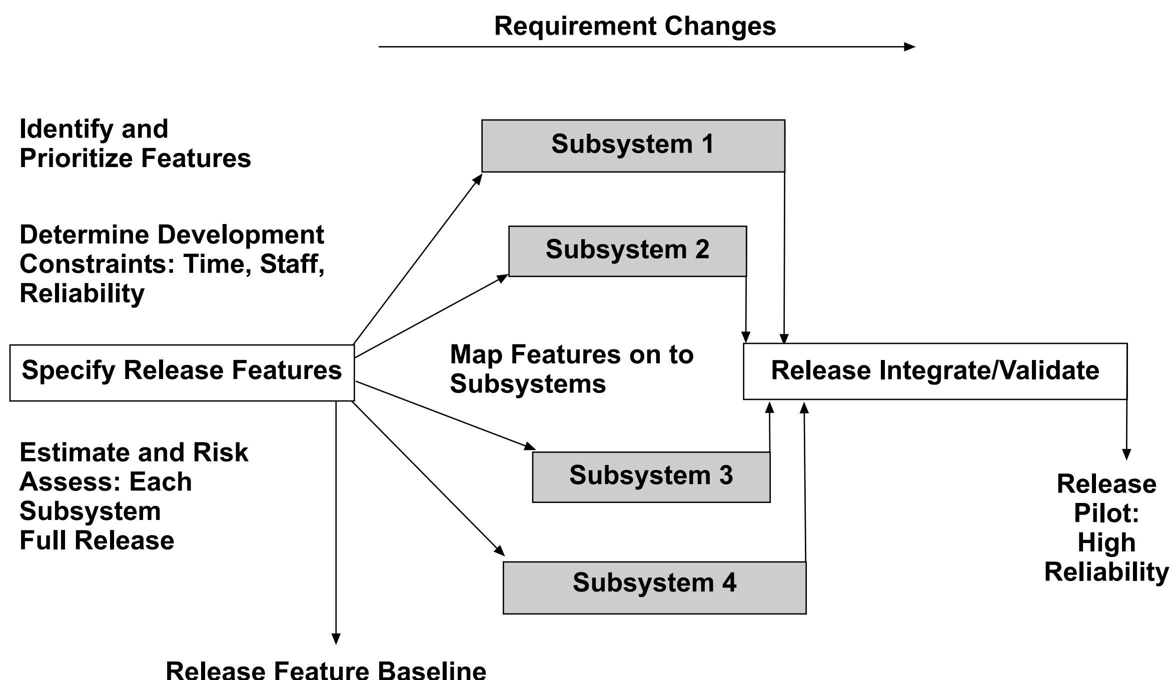

Figure 2 — Development of a release where major subsystems are developed separately.

© 2001 Cutter Information Corp. For subscriptions call +1 800 492 1650 or +1 781 641 9876

The supplier of these advanced telecommunications products develops subsystems at different locations. Each location provides progress metrics up to the point when the individual subsystem is passed over for the final product release integration and validation.

The case study shows how the defect metrics at the subsystem and release level are used to provide evidence of the reliability throughout development and to forecast the remaining defects in order to gauge whether the system is on track to meet the desired reliability when the release is placed into service.

Reliability Model and the Basic Data

Sophisticated defect monitoring in software lifecycles uses a Rayleigh[1] model to forecast the discovery rate of defects as a function of time throughout the software development process. Empirical evidence shows the Rayleigh model closely approximates the actual profile of defect data collected from software development efforts.

The generic form of the Rayleigh model is "tuned" using the actual defect data reported within each development. The model then forecasts the defects that remain and the key milestone dates when specific levels of reliability will be achieved. Where very high reliability is required, this is the point in time when 99.9% of the theoretical software defects have been discovered. Other projects that do not have as stringent reliability demands might suffice with 99% or 95% of the software defects discovered and fixed, which would correspond to placing the system into operation at earlier points in time.

Simple extensions of the model provide other useful information. For example, defect priority classes are specified as percentages of the total. This allows the model to predict defects by severity classes over time. The tuning for the defect classes is again made using the actual reported defects and adjusted as development progresses. In the case study, the reported software defect classes are:

1. Critical
2. Major
3. Minor

Individual Subsystem Progress Data: Software Defects

A monthly summary of the progress metrics from the three subsystems is shown in Figure 3. Subsystem 1 begins about one year

[1]Rayleigh refers to the mathematical curve function where the y-axis rises to a peak, then descends through a long tail over time. Research by the QSM organization shows that software projects exhibit defect patterns that follow the Rayleigh shape. Independent analysis by IBM confirms the Rayleigh function as a sound basis for defect modeling.

	SUBSYSTEM 1						SUBSYSTEM 2						SUBSYSTEM 3					
			Defects						Defects						Defects			
Month	Staff	Code	Total	Crit	Maj	Min	Staff	Code	Total	Crit	Maj	Min	Staff	Code	Total	Crit	Maj	Min
1	2																	
2	1	4500																
3	2	4641																
4	2	5522																
5	3	9802																
6	3	16760																
7	4	19833																
8	6	22682																
9	7	30831																
10	9	31642																
11	10	42242																
12	5.5	44402					1	1500										
13	8.5	50611	15		7	8	5	3300					4.5	721				
14	12	53589	38	2	13	23	8	7769					6.5	1537				
15	11	56211	55		26	299	15.5	16126					13	2592				
16	10.5	58193	78		33	45	20	19716	10	2	5	3	14.5	3879	23	1	15	7
17	10	60368	13		7	6	11	19716	35	10	18	7	18	5871	53	6	29	18
18	9.5	62520	18		3	15	12	19716	24	1	15	8	19.5	5871	36	4	24	8
19	8.5	62984	24		6	18	12.5	22334	54	3	21	30	18	6046	37	3	24	10
20	7.5	63322	6		2	4	10	23850	38	6	11	21	15.5	6492	64	6	36	22
21	6	63789	2			2	6.5	25468	63	1	12	40	11	6575	60	1	25	34

Figure 3 — Subsystem progress data.

ahead of the other two subsystems. This is often the case, due to a particular subsystem having more features and being more large and complex than other subsystems.

The defects are reported once the integration tests for each subsystem begin. Each month, the columns show the total defects and the breakout into the three software defect classes.

Release Consolidated Defects

The data in Figure 3 is consolidated for all subsystems and is used to track the overall defects for the product release. This continues for each subsystem until it completes and is delivered for final release integration, load tests, and validation.

As defects are found during the final release integration, these are identified by subsystems and communicated back to the developers for fixing. Each defect continues to be classified according to severity. The defects are used to tune the Rayleigh defect model and forecast the outstanding defects and the date when the release will be sufficiently reliable to meet the acceptance criteria defined in the contract.

Forecasting the Outstanding Defects

Figure 4 shows the current cumulative defects found by month 21. A total of 750 actual defects are reported. This data is used to tune the defect model and forecast the number of defects that remain to be found and fixed. The results show that an additional 600 defects of all classes remain to be found. High reliability for the full release is achieved around month 33.

Acceptance Milestones: MTTD

Purchasing sets the contractual delivery acceptance in terms of the software MTTD. MTTD is simply the average time between software failures. This is calculated as the reciprocal of the average number of defects within a given period. For instance, if 20 defects are found each working month of 20 days, then, on average, the MTTD is one day between each defect.

The contract sets MTTD objectives using this measure. No dates are set for delivery; acceptance (and payment) is linked to MTTD.

Defects are tracked (and forecast) throughout the development of each subsystem, as well as the entire release. Release acceptance begins when concrete evidence exists that the supplier meets the MTTD acceptance criteria. Quality continues to be improved after this first delivery.

In practice, three MTTD milestones are set (see Figure 5):

1. RFA (ready for acceptance) marks the first delivery to the purchaser to start a factory acceptance test (typically this is an MTTD of 5 days).
2. RFP (ready for production) is the point at which the release is introduced at a single location as a pilot (MTTD of 15 days).

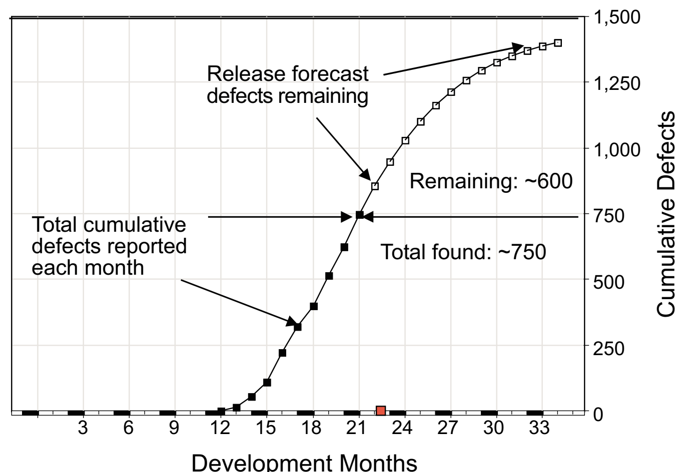

Figure 4 — Cumulative actual defects and forecast remaining defects.

© 2001 Cutter Information Corp. For subscriptions call +1 800 492 1650 or +1 781 641 9876

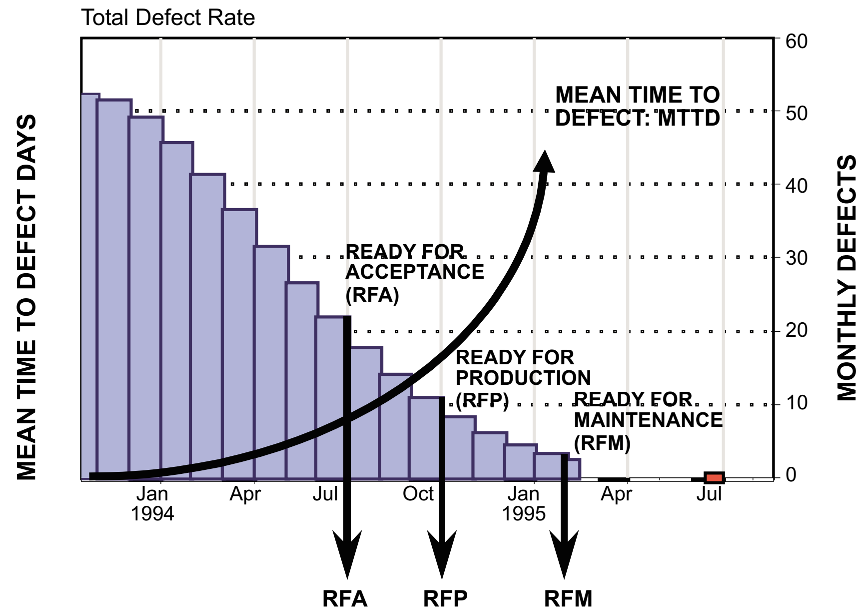

Figure 5 — MTTD acceptance milestones.

3. RFM (ready for maintenance) marks the start of rolling out the new, highly reliable release, after the final defects are found and fixed during the pilot (MTTD of 30 days).

Tracking the Course of the Defect Curves

As the project progresses, tracking the actual size and the defects against the release plan can be enhanced by employing statistical process control (SPC) techniques.

As described earlier, the Rayleigh model provides an expected shape of the planned curve, which defines the trajectory that the project size and defects should track for the release to be completed by the target date at the required reliability. The actuals should closely approximate this rate of progress. However, in real life, there is a normal variation at any given time from where the project actuals "should" be.

As long as the actuals fall within some reasonable bounds, the progress of the release can be said to be within normal limits. It is when performance deviates significantly from these bounds that attention should be drawn to the fact that things have veered off course. This is where SPC methods play a role, to alert the team as early as possible to this potential fact.

Figure 6 demonstrates an example of an SPC chart where defects are plotted against an expected trajectory, rising to a peak, and tailing off in the characteristic Rayleigh shape. The curve plotted in this figure represents a rate curve similar to the partial curve depicted in Figure 5, but with control bounds added. The center line of the curve depicts the expected values for the number of defects found by month. Two sets of upper and lower control bounds define what could be described as a close-in "green light" zone and a farther-out "yellow light" zone.

The actual defect metrics are plotted as connected dots. As you can see, at the second major milestone (shown as number 2, detailed design [DD]) the metrics demonstrated an upward spike that passed through the upper threshold of the green and into the yellow. At that point, a yellow traffic light warning would have been a visual indicator of this condition.

The team would be tasked to react accordingly and return the project to a green light condition if possible. This might be the result of deploying skilled resources to correct the problems. With that action taken, the project returned and stayed in a green

Join our free weekly e-mail service, The Cutter Edge: www.cutter.com/consortium/

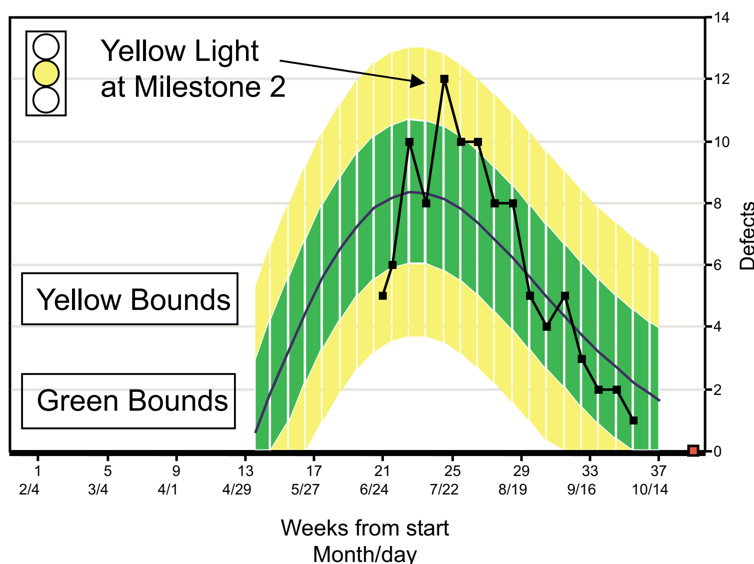

Figure 6 — Statistical process control chart for defects.

condition. This is reflected in the defects continuing their downward path, with the values being sufficiently low enough at the time that the release was successfully placed into service.

The purpose of these types of monitoring methods are to deploy corrective action as early as possible by the team. The truth of IT project lifecycles is that, at some point, they reach a point of no return. At or beyond this point, corrective actions can often be ineffective — it's already too late. The aim of defect monitoring is to capture a project that is veering off course at early stages, when there is still time to affect the end result.

Observations and Conclusions

The results shown here for a large-scale complex development demonstrate that it is practical to treat the individual subsystems as software developments in their own right. The subsystems are consolidated, and their final integration and validation are included as part of the full release development. The full release behaves as a single large development. Reliability depends on finding and fixing the software defects originating from the subsystems. Tracking the subsystem defect behavior reveals the contribution that factors such as time pressure and software size make to the defects found in each subsystem and later during the release integration and validation tests. These insights offer ways to improve reliability and avoid the premature delivery of the subsystems and the release. High reliability, measured by MTTD, determines when release delivery takes place.

In practice, the management techniques outlined here form part of a software control office. This office continuously evaluates and reports on all current developments. This ensures that completed releases are accepted and payments made only when specific reliability objectives are achieved. When subsystems and releases are completed, the software control (or project office) records the size, time, effort, and defect metrics into a historical projects database. This database provides a baseline of recent performance for the various classes of work. It serves as a storehouse of knowledge that can act as the corporate memory of the organization.

© 2001 Cutter Information Corp. For subscriptions call +1 800 492 1650 or +1 781 641 9876

Future release plans are evaluated against this historical projects database to determine whether these are realistic. Disasters are avoided by sizing the features to determine what can be developed within specific constraints and their level of risk. Deadline targets and promised functionality for a given release are tested for legitimacy and reasonableness against this internal benchmark. In this way, a realistic baseline-planning estimate is produced. Progress visibility during development is then ensured using the contractual progress data. This visibility results in the early detection of any variance against the baseline plan and initiates immediate corrective action.

These two aspects — a realistic estimate and continuous visibility during development — are fundamental to the successful management of software projects by development and purchasing organizations. Although true for all software development, this is even more important where large-scale, complex subsystems are involved.

About the Author

Jim Greene is managing director of Quantitative Software Management Europe in Paris, France. He has more than 35 years of experience in software engineering, with a particular interest in management methods used by development and purchasing organizations based on the quantification of software development. He can be reached at qsm.europe@pobox.com.

References

Greene, J.W.E. "Sizing and Controlling Incremental Development." *Managing System Development,* November 1996.

Greene, J.W.E. "Software Process Improvement — Management Commitment, Measures and Motivation." *Managing System Development,* February 1998.

Greene, J.W.E. "The Software Control Office." *EC2 Software Engineering Conference*, Toulouse, France, 1991.

Greene, J.W.E. "Getting a Runaway Software Development under Control." *EC2 Software Engineering Conference*, Toulouse, France, 1990.

Humphrey, Watts S. "Three Dimensions of Process Improvement — Part 1: Process Maturity." *Crosstalk: The Journal of Defense Software Engineering,* February 1998.

Kempff, Geerhard W. "Managing Software Acquisition." *Managing System Development,* July 1998.

Putnam, Lawrence H. "The Economic Value of Moving up the SEI Scale." *Managing System Development,* July 1994.

Putnam, Lawrence H. *Measures For Excellence: Reliable Software, On Time, Within Budget.* Prentice Hall, 1992.

Putnam, Lawrence H. and Ware Myers. *Industrial Strength Software: Effective Management Using Measurement.* IEEE Computer Society Press, 1997.

❑ Please start my subscription to ***IT Metrics Strategies®*** for one year at $485, or US $545 outside North America.
Phone +1 781 648 8700 or +1 800 964 5118; Fax +1 781 648 1950 or +1 800 888 1816; or E-mail sales@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, or Diners Club

220*5ITS

Card no.
Expiration Date
Signature

Web site: www.cutter.com/itms/
Cutter Information Corp.
37 Broadway, Suite 1
Arlington, MA 02474-5552 USA