Successfully Applying Software Metrics

Robert B. Grady, Hewlett-Packard

What do you need to measure and analyze to make your project a success? These examples from many projects and HP divisions may help you chart your course.

The word *success* is very powerful. It creates strong, but widely varied, images that may range from the final seconds of an athletic contest to a graduation ceremony to the loss of 10 pounds. Success makes us feel good; it's cause for celebration.

All these examples of success are marked by a measurable end point, whether externally or self-created. Most of us who create software approach projects with some similar idea of success. Our feelings from project start to end are often strongly influenced by whether we spent any early time describing this success and how we might measure progress.

Software metrics measure specific attributes of a software product or a software-development process. In other words, they are measures of success. It's convenient to group the ways that we apply metrics to measure success into the four areas shown in Figure 1. This article contains four major sections highlighting examples of these areas.

Figure 1 also shows two arrows that represent conflicting pressures for data. For example, on one of the first software projects I managed, the finance department wanted me to use their system to track project costs, arguing that this would help me. I shortly learned that their system didn't give me the kind of information I needed to be successful. The reports weren't timely or accurate, and they didn't usefully measure progress. This was one of my first experiences with the opposing desires for information that can arise between a project manager and the division's management team. They wanted summary data across many diverse functions; I wanted data that would help me track day-to-day progress.

I soon realized that projects stand the best chance of success when the goals driving the use of different measures can be stated and mutually pursued. This article's examples are all from real projects, and they were chosen to show both viewpoints illustrated by the arrows in Figure 1. The examples also show how the possibly con-

0018-9162/94/$4.00 © 1994 IEEE

flicting goals of a project team and of an organization can effectively complement each other. Finally, they are examples of things that *you* can measure to be more successful.

Project estimation and progress monitoring

Today there are dozens of software-estimation tools. Figure 1 suggests that such tools can be quite useful to project managers. These tools are now very sophisticated because they account for many possible project variables. Unfortunately, most of us are not much better at guessing the right values for these variables than we are at guessing total project schedules.

The basis for estimates. Most estimating tools are based on limited measurements. For example, the first three columns of Table 1 show measurements for my early 25,000-engineering-hour project, with and without nonengineering activities. (I finally tracked these measurements without using our normal accounting system.) Some of the data is useful for future estimates. For example, the percentages for supervision and administrative support would be reasonably accurate for other projects, particularly since they can be controlled. Even the time spent in different activities doesn't differ much from the averages for 132 more-recent Hewlett-Packard projects, although my team didn't collect the data in exactly the way that HP currently does.

Should you collect data like this for your projects? Since estimation models are based on such data, informally collecting it will help you track the validity of your inputs into any model. This data can give you useful insights into the accuracy of your estimates. The earlier you find differences, the more likely it is that management might accept schedule changes.

The bottom line. Higher level managers are usually not interested in as much detail as Table 1 presents. They want the bottom line: Is the project on schedule? Figure 2 shows how one HP lab tracked this across many projects.[1,2] Two ideas went into this graph. First, a schedule slip is the amount of time that a project schedule is moved to a later date. Second, average project progress for a

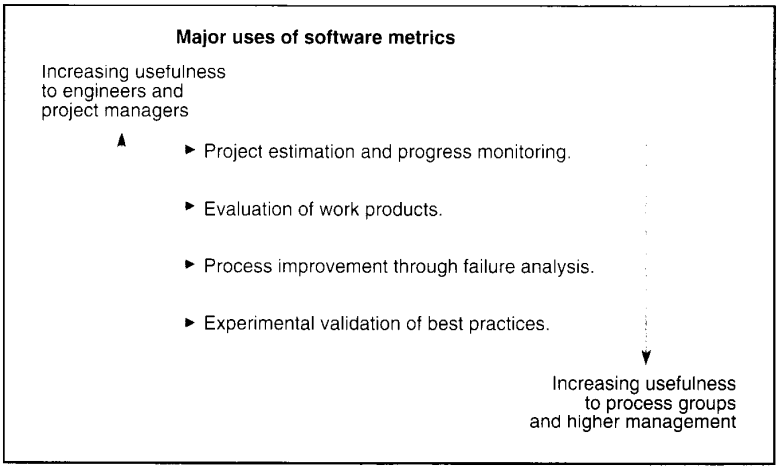

Figure 1. Major uses of software metrics and the conflicting pressures for data.

Table 1. Task breakdowns for a 25,000-engineering-hour software project over 2.5 calendar-years.

Tracked Times	Project %	Eng. Only %	Categories Currently Tracked	Approx. Project %	HP Average
Investigation	20	26	Reqs./Specs.	19	18
External/Internal			Design	16	19
Reference Specs	2	2			
Coding	19	23	Implement	32	34
Debugging	19	24			
Integration	11	14	Test	33	29
Quality Assurance	8	11			
Manuals	7				
Supervision	9			Not included	
Support	5				

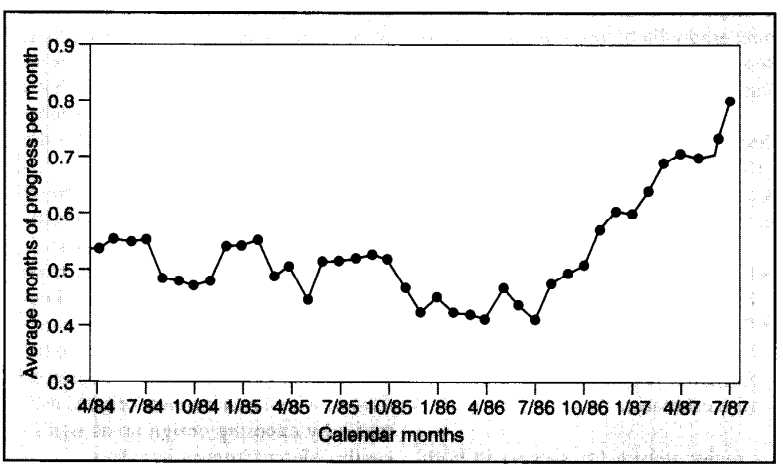

Figure 2. Development project progress for all software projects in one HP division.

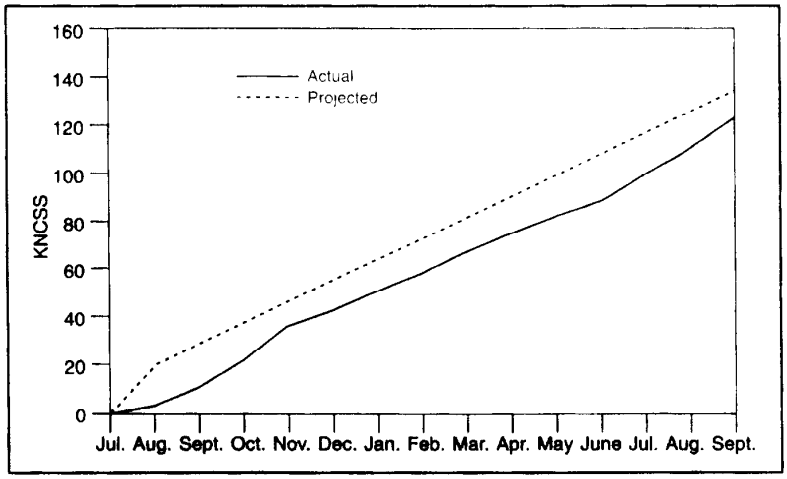

Figure 3. Plot of thousands of noncomment source statements (KNCSS) against time for project summarized in Table 1 (© 1987 Prentice-Hall, used with permission).

time period is defined as one minus the ratio of the sum of all project slips divided by the sum of project elapsed times.

For example, suppose you have a small lab with three projects. In a one-month period, the first project's manager believes that it is on schedule (its schedule doesn't change). Its slip is zero. The second project had a bad month. Its slip is one. The third project's manager expects the project will slip one week, for a slip of about one-quarter. The sum of the slips is 1.25. The sum of elapsed times is 3. Average project progress is therefore $1 - (1.25/3) = 0.58$.

The graph uses a moving average to smooth month-to-month swings. The lab first started plotting the graph around October 1985. They had enough historical data to show that they had only averaged about a half month of progress for every elapsed month. After monitoring the graph for a few months, the project teams gradually focused on more accurate schedules, and they improved their accuracy to an enviable point.[1,2]

Successful usage. The examples shown in Table 1 and Figure 2 were both successes. They show two things that you will see repeated in other examples:

- Lab management wanted limited, high-level summary data.
- Project-management data provided both confidence-building tracking information and a basis for better future estimates.

A major reason for success in both cases was that their end points — their goals — were measurable. Figure 2 graphically shows that the way the division estimated schedules hadn't changed for at least 1.5 years before they defined a way to measure progress. In the other example, Table 1, the data influenced dozens of my decisions. They included resource balancing, intermediate and final schedule commitments, test schedules, and technical writing schedules. Furthermore, I could confidently show and explain progress on a very large project in ways that few high-level HP managers had seen before.

Monitoring progress against estimates. There are two time-proven ways to track progress on a software project. The first is to track completed functionality (the features or aspects of the software product).

Tracking functionality. Despite often-expressed concerns about the usefulness of counting lines of code, I have found tracking code size against time to be very useful for managing projects. Figure 3 is a plot of thousands of noncomment source statements (KNCSS) against time for the project summarized in Table 1.[3]

KNCSS represents completed functionality here. It is reasonable to use during the coding and testing phases, particularly if you track coded NCSS separately from tested NCSS (not shown in Figure 3). I updated this graph every week to make sure the project was on track. I also tracked the status of modules designed for this project. This provided a link to our original estimates by exposing design areas significantly different from earlier plans.

More recently, HP has measured FURPS criteria (functionality, usability, reliability, performance, and supportability) to complement simpler size-tracking metrics. (Grady[2] describes FURPS more completely.)

Finally, function points, another popular functionality measure, are computed from a combination of inputs, outputs, file communications, and other factors. They can be computed independently of source code, so some people find their added difficulty of use offset by this earlier availability. (Capers Jones' *Applied Software Measurement*, McGraw-Hill, New York, 1991, is a useful function-point reference.)

Found-and-fixed defects. Tracking functionality doesn't attract high-level management attention like the trends of found-and-fixed defects. This second method is very useful in monitoring progress in later development phases. These trends are also among the most important aids you have in deciding when to release a product successfully. Methods for analyzing such trends vary. Variations include simple trend plotting, sophisticated customer-environment modeling, and accurate recording of testing or test-creation times.

Using defect trends to make larger system-release decisions gives valuable confidence to higher level managers. They feel more comfortable when major release decisions are backed by data and graphs. Figure 4 shows the system test defects for a project involving over 30 engineers. It shows this project's status about one month before completion. The team derived the goal from past project experiences. One release criteria was for the defects/1,000-test-hours rate to drop and stay below the goal line for at least two weeks before release. The alternative projections were simple hand-drawn extrapolations using several of the past weeks' slopes. The team updated the graph every week. The weekly test hours were much fewer than 1,000, so the weekly ratios may give an impression that there were more defects than there really were.

HP has learned that the critical downward trend that Figure 4 displays is necessary to avoid costly postrelease crises. This project's downward slope continued, and the project released successfully. An opposite example was an HP software system released despite an absence of a clear downward defect trend. The result was a multimillion-dollar update shortly after release and a product with a bad quality reputation. This kind of mistake can cause

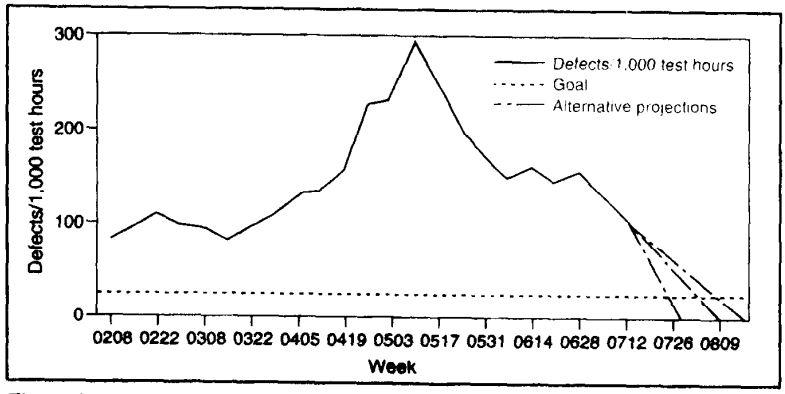

Figure 4. System-test defect trend for project involving over 30 engineers.

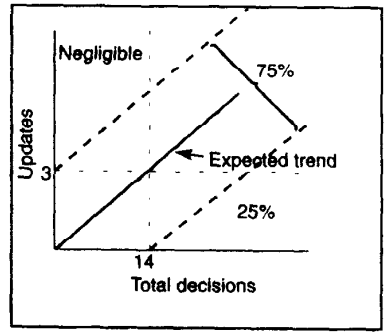

Figure 5. Trend analysis of number of source updates for system-level testing versus the number of decision statements per subroutine.

an entire product line's downfall. A recent article describes how one company learned this lesson the hard way.[4]

Plotting defect trends is one method where both project and high-level management have similar interests. While exact completion points may vary based on differing project goals, better decisions are possible when trends are visible. Those decisions will help to ensure project success.

Evaluation of work products

A work product is an intermediate or final output that describes the design, operation, manufacture, or test of some portion of a deliverable or salable product. It is not the final product. All software development finally results in a work product of code. While this section's brief examples center on code, the idea of extracting useful metrics from virtually any work product is the point to remember.

Because code can be analyzed automatically, it has been a convenient research vehicle for sophisticated statistical analysis. Unfortunately, this emphasis has created a strong bias in perceptions of metric applications. Many managers believe that useful metrics require time-consuming techniques outside of their normal decision-making processes. Even recently, one metrics expert told me that the minimum number of code metrics a project manager should monitor is around 20.

Cyclomatic complexity. Fortunately, HP has had good results when measuring just one code metric: cyclomatic complexity, which is based on a program's decision count. (The decision count includes all programmatic conditional statements, so if a high-level-language statement contains multiple conditions, each condition is counted once.) One HP division especially saw this when they combined the metric with a visual image that graphically showed large complexity.[5] Graphs, and their source code cross references, help engineers understand problem locations and may provide insights for fixing them. The graphs excite managers because their availability encourages engineers to produce more maintainable software. This doesn't mean that the tool's numeric values are ignored: Complexity metrics give managers and engineers simple numerical figures of merit.

When considering engineering tool value, high-level managers want to know whether using such a tool yields better end products in less time. Project managers may have to look at other data like that shown in Figure 5 to build a strong case for tools.[6] This study concerned a project of 830,000 lines of executable Fortran code.

Those doing the study plotted the relationship between program-decision counts and the number of updates reflected by their source code control system. Seventy-five percent of the updates fell within the dashed lines. For their system, the number of updates was proportional to the number of decision statements. From their analysis, they drew a trend line. By knowing the cost and schedule effects of modules with more than three updates, they concluded that 14 was the maximum decision count to allow in a program. (Tom McCabe originally suggested 10, based on testing difficulty.[7])

You can do a similar analysis or you can accept these and similarly documented results and assume they apply equally well to your project. Then estimate how using complexity tools can make your project more effective. Measure normal defect rates and both the engineering time and the calendar time to do fixes. Estimate how long it would take to run complexity tools. Finally, calculate your savings when you reduce complexity *before* your people start finding the defects in test. Grady[2] provides an economic justification for the purchase of complexity tools like these.

But the metrics expert who set a minimum of 20 code metrics was not totally wrong. Because cyclomatic complexity is a measure of control complexity, it is more valuable for control-oriented applications than for data-oriented ones. It works for both, but the characteristics of data-oriented applications suggest that you must consider other dimensions as well. Unfortunately, reported data-oriented results haven't been as thoroughly tested as those I've mentioned.

Design complexity. A promising metric for data-oriented complexity is fanout squared. The fanout of a module is the number of calls from that module. At least three studies have concluded that fanout squared is one component of a design metric that correlates well to probability of defects.[2,8,9] More importantly, fanout squared can be determined before code is created. Figure 6 on the next page shows the top-level structure chart of the most defect-prone module in a system. This module was the source of 50 percent of the system test defects, even though it had only 8 percent of the code. Its fanout squared was also the largest among the system's 13 top-level modules. In fact, postrelease defect densities were highly correlated to the fanout squared of the system's modules.

The figure shows a large number of

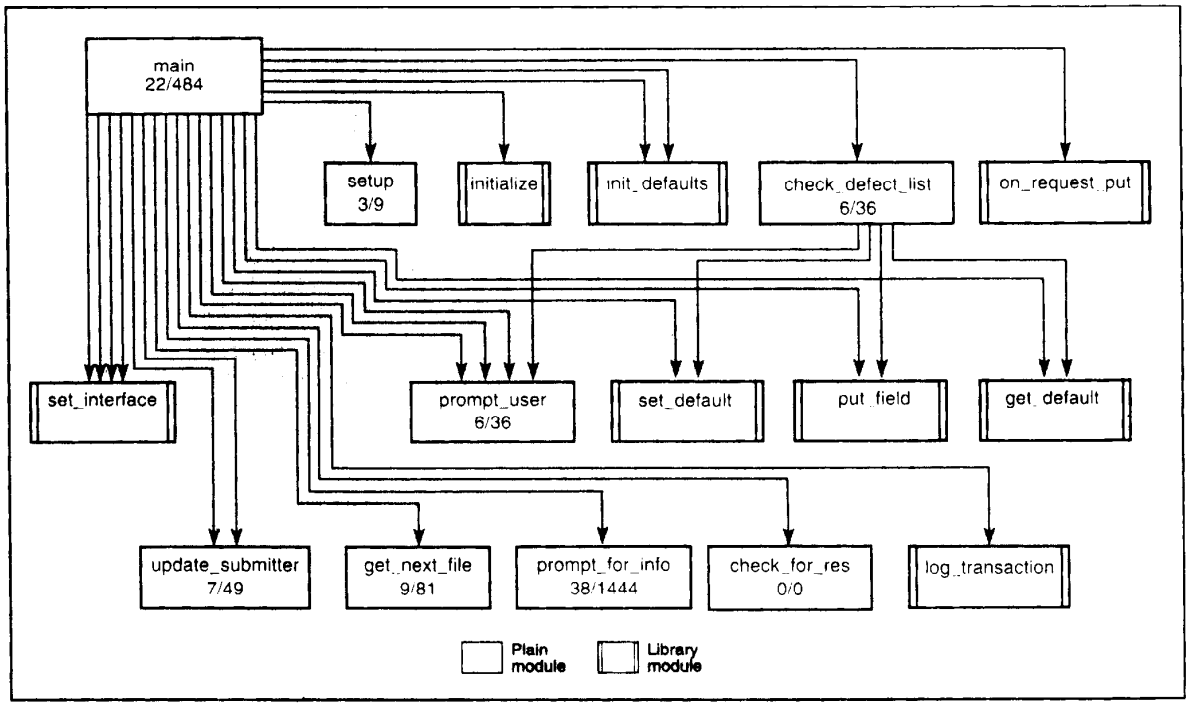

Figure 6. Structure chart showing fanout/fanout squared for each module. Diagram is simplified to show a complete set of connections for the main module only. Calls to system or standard utility routines are not shown or counted.

connections between *main* and the 15 other modules. The library modules don't call other modules, so no fanout is given for them. Note the fanout for *prompt_for_info*. With a fanout of 38 (and a fanout squared of 1,444), you can imagine its structure chart's complexity. Such large fanouts suggest that there is a missing design layer. Structure charts combine with fanout squared to give the same type of results as cyclomatic complexity and graphical views of control flow, only earlier.

Based on limited past experimental results, design complexity metrics may not be justified yet during design. However, if information like fanout squared were readily available as a byproduct of normal project tools, progress toward understanding design complexity would be faster. Meanwhile, measuring code complexity is desirable from both project and higher management viewpoints. Also, measuring design complexity in designs provides an important research opportunity.

More on successful usage. Cyclomatic complexity and fanout squared are just two types of work-product analysis. Automation recently introduced by the CASE (computer-aided software engineering) boom continues to expand engineers' comprehension of their work. You can take advantage of complexity data to help your project in several ways. Like the Figure 5 example suggests, you can enforce a standard by computing complexity for all modules and not accepting any above some value. Another approach is to limit the number of modules above a given complexity level. Then make sure those modules are inspected and tested carefully. Another way is to require more documentation for such modules. Whichever way works best for you, it is certain that complexity information will help both you and your engineers to be more successful by providing information for better, more timely decisions.

Process improvement through failure analysis

I believe this third area from Figure 1 is the most promising for improving development processes. Failure analysis, and finding and removing major defect sources, offers the best short-term potential for guiding improvements.

Project defect patterns. There are several valuable approaches. One simple approach for project managers is to monitor the number of defects found during system test, code inspections, or design inspections. This data can be sorted by module, and special actions can be taken as soon as potential problem areas appear. For example, Figure 7 shows both prerelease system-test defects and defects for a product during the first six months after its release.[3]

Unfortunately, the project manager in this case didn't do anything different until after collection of postrelease data. The project manager then focused team effort on thorough inspections and better testing of the three most defect-prone modules. (While these modules were only 24 percent of the code, they accounted for 76 percent of the defects.) As a result, the team succeeded in greatly reducing the product's incoming defect rate by focusing on those three modules.

Software process defect patterns. Another type of failure analysis examines defect patterns related to development processes. This analysis affects enough people to generally require lab-level management sponsorship. Many HP divisions today start this analysis by cate-

U-M-I
DUE TO LACK OF CONTRAST, GRAPHS DID NOT REPRODUCE WELL.
GRAPHS FOLLOW SAME SEQUENCE AS LEGEND

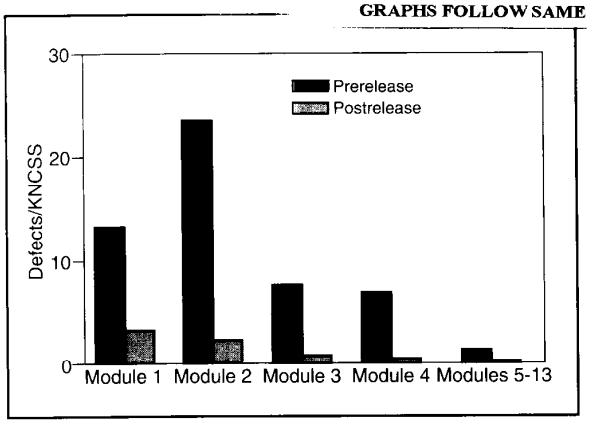

Figure 7. Analysis by code module for prerelease system-test defects and for those occurring during the first six months after the product's release (© 1987, Prentice-Hall, used with permission).

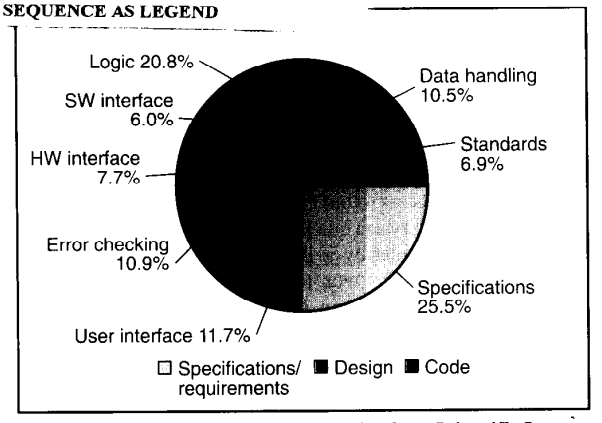

Figure 8. Top eight causes of defects for four Scientific Instruments Division projects at Hewlett-Packard.

gorizing their defects according to a three-level HP model used since 1987. The three levels are origin, type, and mode. Grady[2] describes several significant improvements achieved in divisions using this model. Figure 8 shows recent data for two levels of the model for yet another HP division (Scientific Instruments Division). The shading represents defect origin information, and the pie wedges are defect types. The data reflects the eight most frequently recorded causes of defects for four projects.

The division performed three post-project reviews to brainstorm potential solutions to their top four defect types. Several initiatives were launched. The first of these that yielded data for a full project life cycle recently concluded. The project team had decided to focus on user-interface defects. They had over 20 percent of that defect type on their previous project (even though the division-wide average was lower). They brainstormed the Figure 9 fishbone diagram and decided to create guidelines for user-interface designs that addressed many of the fishbone-diagram branches.

Their results were impressive. They reduced the percentage of user-interface defects in test for their new year-long project to roughly 5 percent. Even though the project produced 34 percent *more* code, they spent 27 percent *less* time in test. Of course, other improvement efforts also contributed to their success. But the clear user-interface defect reduction showed them that their new guidelines and the attention they paid to

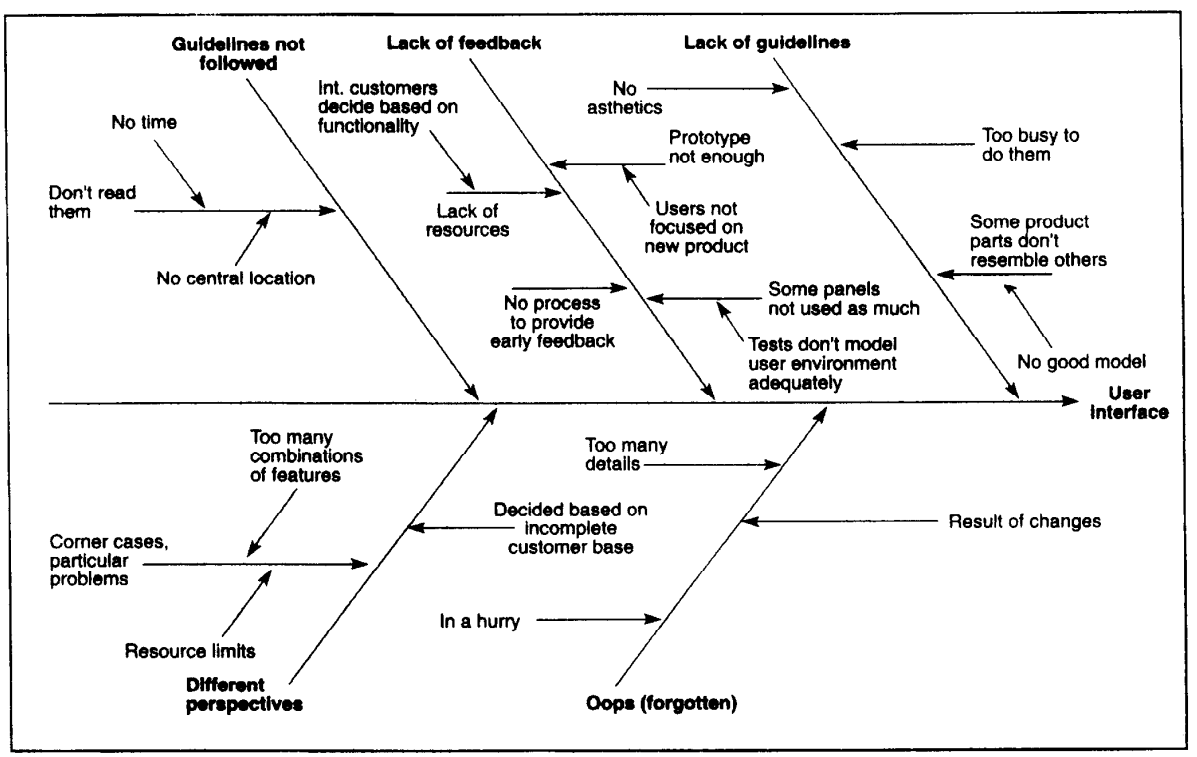

Figure 9. Fishbone diagram showing the causes of user-interface defects.

their interfaces were major contributors.

The examples you've seen in Figures 7, 8, and 9 show how a small investment in failure analysis can reap practical short-term gains. Ironically, the main limiter to failure-analysis success is that many managers still believe they can quickly reduce *total* effort or schedules by 50 percent or more. As a result, they won't invest in more modest process improvements. This prevents them from gaining 50 percent improvements through a series of smaller gains. Because it takes time to get any improvement adopted *organization-wide*, these managers will continue to be disappointed.

Table 2. Comparison of testing efficiencies.*

Testing Type	Efficiency (Defects found/hour)
Regular use	0.210
Black box	0.282
White box	0.322
Reading/ Inspections	1.057

*Defect-tracking system lumped code reading and inspections into one category. About 80 percent of the defects so logged were from inspections.

Experimental validation of best practices

This software metric use has been the most successful of the four listed in Figure 1. People have validated the success of important engineering practices (for example, prototyping,[10] reducing coupling, increasing cohesion,[9] limiting complexity,[6] inspections and testing techniques,[11] and reliability models[12]). This validation should lead to quicker, widespread acceptance of these "best" practices.

Of the four Figure 1 metric uses, project managers are least motivated to validate best practices because normal project demands have higher priority. On the other hand, these metrics have probably brought project managers the greatest benefits. The first example here is what high-level managers want to see. One HP division measured the data in Table 2 for different test and inspection techniques. The average efficiency of code reading/code inspections was 4.4 times better than other test techniques yield.[2,13] This data helps project managers to plan inspections for their projects and to convince their engineers of the merits of inspections by showing the benefits.

However, experience has shown that it takes many years to widely apply even proven best practices. It often takes local proof to convince engineers to change their practices. For example, Henry and Kafura first showed the fanout squared metric discussed earlier to be useful over 10 years ago (as a part of their information-flow metric).[8] Even then, they pointed out how such an early design metric would be useful during design inspections. Unfortunately, most software-developing organizations don't have standard design practices yet. Also, people haven't been convinced that it's worth the effort to do high-level design with the detail necessary to compute such measures.

Project managers might find Figure 10's graph useful for their projects. It shows all the fanout squared values for an HP product. If you had this information early in your project, you could focus inspections and evaluations on the high-value modules. Like cyclomatic complexity, fanout squared appears to have several very desirable properties:

- It is easy to compute.
- Graphical views (like Figure 6) do reflect high complexity.
- The metric exposes a small percentage of a system's modules as potential problems.

Although this may be a significant future metric, it illustrates a dilemma. How much time can project managers or organizations spend proving such practices? As positive evidence grows and competitive pressures for higher quality grow, the motivation to apply promising new practices also increases. Not all validations of beneficial practices are as easy to measure as inspections. However, this is the road to progress. My advice to project managers is to invest some of your team's effort on improvements, but track and validate the benefits. My advice to high-level managers is to reserve some funds and encouragement to support such validations.

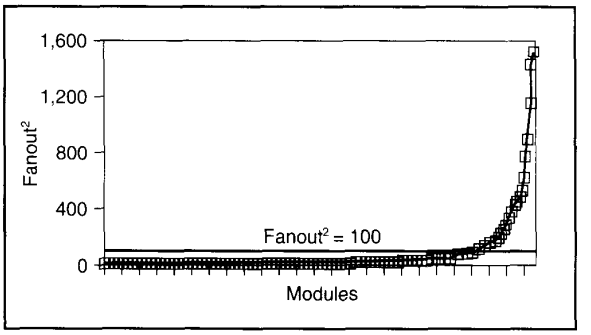

Figure 10. Fanout squared for a 250-module product, sorted by ascending fanout squared (which is more than 100 for only 10 percent of the modules).

How do you apply software metrics to be successful? Review the four major uses of metrics, studying the project-level and management-level examples from successful projects. These examples lead to three recommendations for project managers:

- Define your measures of success early in your project and track your progress toward them.
- Use defect data trends to help you decide when to release a product.
- Measure complexity to help you optimize design decisions and create a more maintainable product.

Don't forget that other aspects contribute to successful metrics usage and project management beyond this article's examples. They include linking metrics to project goals, measuring product-related metrics, and ensuring reasonable collection and interpretation of data.

Consider two more recommendations for strategic purposes:

- Categorize defects to identify product and process weaknesses. Use this data to focus process-improvement decisions on high-return fixes.
- Collect data that quantifies the success of best practices.

This is all useful advice, *but what do you need to measure to be successful*? It is difficult to reduce this answer to a small set of measures for high-level managers. Chapter 15 of Grady[2] discusses nine

useful management graphs. Finally, I suggest that project managers collect the following data:

- engineering effort by activity,
- size data (for example, noncomment source statements or function points),
- defects counted and classified in multiple ways,
- relevant product metrics (for example, selected measurable FURPS),
- complexity, and
- testing code coverage (an automated way of measuring which code has been tested).[2]

Understand how each of these relates to *your* success, and perform timely analyses to optimize your future. ■

Acknowledgments

I especially thank Jan Grady and Debbie Caswell for their early reviews and suggestions when this article was just a way of organizing my ideas for a talk. I also thank other Hewlett-Packard and *Computer* reviewers for their helpful comments and suggestions. Finally, thanks to Brad Yackle, Mark Tischler, and the others at SID for sharing their failure-analysis results.

References

1. D. Levitt, "Process Measures to Improve R&D Scheduling Accuracy," *Hewlett-Packard J.*, Vol. 39, No. 2, Apr. 1988, pp. 61-65.
2. R. Grady, *Practical Software Metrics for Project Management and Process Improvement*, Prentice-Hall, Englewood Cliffs, N.J., 1992.
3. R. Grady and D. Caswell, *Software Metrics: Establishing a Company-Wide Program*, Prentice-Hall, Englewood Cliffs, N.J., 1987, pp. 31 and 110.
4. D. Clark, "Change of Heart at Oracle Corp.," *San Francisco Chronicle*, July 2, 1992, pp. B1 and B4.
5. W. Ward, "Software Defect Prevention Using McCabe's Complexity Metric," *Hewlett-Packard J.*, Vol. 40, No. 2, Apr. 1989, pp. 64-69.
6. R. Rambo, P. Buckley, and E. Branyan, "Establishment and Validation of Software Metric Factors," *Proc. Int'l Soc. Parametric Analysts Seventh Conf.*, 1985, pp. 406-417.
7. T. McCabe, "A Complexity Measure," *IEEE Trans. Software Eng.*, Vol. SE-2, No. 4, Dec. 1976, pp. 308-320.
8. S. Henry and D. Kafura, "Software Structure Metrics Based on Information Flow," *IEEE Trans. Software Eng.*, Vol. SE-7, No. 5, Sept. 1981, pp. 510-518.
9. D. Card with R. Glass, *Measuring Software Design Quality*, Prentice-Hall, Englewood Cliffs, N.J., 1990.
10. B.W. Boehm, T. Gray, and T. Seewaldt, "Prototyping vs. Specifying: A Multi-Project Experiment," *Proc. Seventh Int'l Conf. Software Eng.*, IEEE Press, Piscataway, N.J., Order No. M528 (microfiche), 1984, pp. 473-484.
11. L. Lauterbach and W. Randell, "Six Test Techniques Compared: The Test Process and Product," *Proc. Fourth Int'l Conf. Computer Assurances*, Nat'l Inst. Standards and Technology, Gaithersburg, Md., 1989.
12. M. Ohba, "Software Quality = Test Accuracy × Text Coverage," *Proc. Sixth Int'l Conf. Software Eng.*, IEEE Press, Piscataway, N.J., 1982, pp. 287-293.
13. T. Tillson and J. Walicki, "Testing HP SoftBench: A Distributed CASE Environment: Lessons Learned," *HP SEPC Proc.*, Aug. 1990, pp. 441-460 (internal use only).

Robert B. Grady is the software-metrics program manager for Hewlett-Packard's Corporate Engineering Software Initiatives. His research interests include software process improvement, failure analysis, management methods, and metrics. Grady has written and coauthored numerous papers and articles on software subjects, and the books *Software Metrics: Establishing a Company-Wide Program* (Prentice-Hall, 1987) and *Practical Software Metrics for Project Management and Process Improvement* (Prentice-Hall, 1992).

Grady received a BS in electrical engineering from the Massachusetts Institute of Technology and an MS in electrical engineering from Stanford University. He is a member of the IEEE Computer Society.

Readers can contact Grady at Hewlett-Packard, Bldg. 5M, 1501 Page Mill Rd., Palo Alto, CA 94304.