Using Metrics to Manage Software Projects

Edward F. Weller

Bull HN Information Systems*

In 1989, Bull's Arizona facility launched a project management program that required additional software metrics and inspections. Today, the company enjoys improvements in quality, productivity, and cost.

Five years ago, Bull's Enterprise Servers Operation in Phoenix, Arizona, used a software process that, although understandable, was unpredictable in terms of product quality and delivery schedule. The process generated products with unsatisfactory quality levels and required significant extra effort to avoid major schedule slips.

All but the smallest software projects require metrics for effective project management. Hence, as part of a program designed to improve the quality, productivity, and predictability of software development projects, the Phoenix operation launched a series of improvements in 1989. One improvement based software project management on additional software measures. Another introduced an inspection program,[1] since inspection data was essential to project management improvements. Project sizes varied from several thousand lines of code (KLOC) to more than 300 KLOC.

The improvement projects enhanced quality and productivity. In essence, Bull now has a process that is repeatable and manageable, and that delivers higher quality products at lower cost. In this article, I describe the metrics we selected and implemented, illustrating with examples drawn from several development projects.

Project management levels

There are three levels of project management capability based on software-metrics visibility. (These three levels shouldn't be equated with the five levels in the Software Engineering Institute's Capability Maturity Model.) Describing them will put the Bull examples in perspective and show how we enhanced our process through gathering, analyzing, and using data to manage current projects and plan future ones.

First level. In the simplest terms, software development can be modeled as shown in Figure 1. Effort, in terms of people and computer resources, is put into a process that yields a product. All too often, unfortunately, the process can only be described

* Since writing this article, the author has joined Motorola.

0018-9162/94/ $4.00 © 1994 IEEE

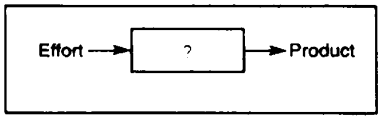


Figure 1. Software development level 1: no control of the development process. Some amount of effort goes into the process, and a product of indeterminant size and quality is developed early or (usually) late, compared to the plan.

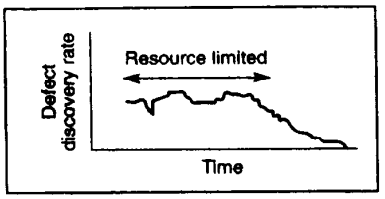


Figure 2. Defect discovery profile for lower development levels. The number of defects in the product exceeds the ability of limited resources to discover and fix defects. Once the defect number has been reduced sufficiently, the discovery rate declines toward zero. Predicting when the knee will occur is the challenge.

by the question mark in Figure 1. Project managers and development staff do not plan the activities or collect the metrics that would allow them to control their project.

Second level. The process depicted in Figure 1 rarely works for an organization developing operating-system software or large applications. There is usually some form of control in the process. We collected test defect data in integration and system test for many years for several large system releases, and we developed profiles for defect removal that allowed us to predict the number of weeks remaining before test-cycle completion. The profile was typically flat for many weeks (or months, for larger system releases in the 100- to 300-KLOC range) until we reached a "knee" in the profile where the defect discovery rate dropped toward zero (see Figure 2).

Several factors limit the defect discovery rate:

- Defects have a higher probability of being "blocking defects," which prevent other test execution early in the integration- and system-test cycle.
- The defect discovery rate exceeds the development staff's capacity to analyze and fix problems, as test progresses and more test scenarios can be run in parallel.

Although this process gave us a fairly predictable delivery date once the knee was reached, we could not predict when the knee would occur. There were still too many variables (as represented by the first two boxes in Figure 3). There was no instrumentation on the requirements or design stages (the "?" in Figure 3).

Our attempts to count or measure the size of the coding effort were, in a sense, counterproductive. The focus of the development effort was on coding because code completed and into test was a countable, measurable element. This led to a syndrome we called WISCY for "Why isn't Sam coding yet?" We didn't know how to measure requirements analysis or design output, other than by document size.

We also didn't know how to estimate the number of defects entering into test. Hence, there was no way to tell how many weeks we would spend on the flat part of the defect-removal profile. Predicting delivery dates for large product releases with 200 to 300 KLOC of new and changed source code was difficult, at best.

A list of measures is available in the second-level model (Figure 3):

- effort in person-months,
- computer resources used,
- the product size when shipped, and
- the number of defects found in the integration and system tests.

Figure 3. Software development at level 2: measurement of the code and test phases begins.

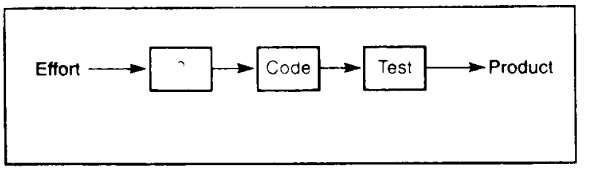
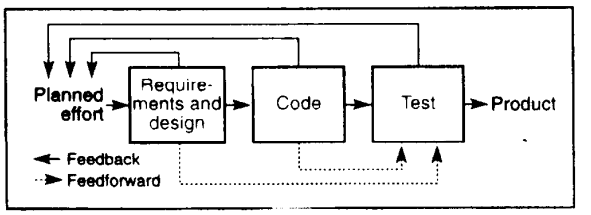


Figure 4. Software development level 3: control of the entire development process. You measure the requirements and design process to provide feedforward to the rest of the development as well as feedback to future planning activities.



Although these measures are "available," we found them difficult to use in project planning — there was little correlation among the data, and the data was not available at the right time (for instance, code size wasn't known until the work was completed). Project managers needed a way to predict and measure what they were planning.

Third level. The key element of the initiative was to be able to predict development effort and duration. We chose two measures to add to those we were already using: (1) size estimates and (2) defect removal for the entire development cycle.

Because the inspection program had been in place since early 1990, we knew we would have significant amounts of data on defect removal. Size estimating was more difficult because we had to move from an effort-based estimating system (sometimes biased by available resources) to one based on quantitative measures that were unfamiliar to most of the staff. The size measures were necessary to derive expected numbers of defects, which then could be used to predict test schedules with greater accuracy. This data also provided feedback to the planning organization for future projects.

To meet the needs of the model shown in Figure 4, we needed the following measures (italics designate changes from the prior list):

- effort in person-months,
- computer resources used,
- *estimated product size at each development stage,*
- product size *after coding,*
- *product size after each test stage,*
- number of defects *found in all development stages from inspections* (in this article, inspection defects refer to major defects), *unit test,* integration test, and system test, and
- *estimated completion date for each phase.*

The sidebar "Data collection sheet" shows a sample form used to compile data.

Project planning

Once the project team develops the first size estimate, the project manager begins to use the data — as well as historical data from our metrics database — for effort and schedule estimating. Several examples from actual projects illustrate these points.

Using defect data to plan test activities. We use the inspection and test defect databases as the primary defect-estimation source. The inspection data provides defect detection rates for design and code by product identifier (PID). Our test database can be searched by the same PID, so a defect depletion curve[2] for the project can be constructed by summarizing all the project's PIDs. (Several interesting examples in Humphrey[2] provided a template for constructing a simple spreadsheet application that we used to plan and track defect injection and removal rates.) Figure 5 shows such a curve for one project. The size and defect density estimates were based on experience from a prior project. The project manager estimated the unit and integration test effort from the defect estimates and the known cost to find and fix defects in test. The estimates and actual amounts are compared in the "Project tracking and analysis" section below.

Data collection sheet

This "Data collection sheet," developed by Kathy Griffith, Software Engineering Process Group project manager at Bull, compiles effort, size, defect, and completion data. Although the sheet is somewhat busy, only six data elements are estimated or collected at each development-cycle phase. The cells with XX in them indicate data collected at the end of high-level design; the cells with YY are derived from the XX data.

DATA COLLECTION SHEET

Project Name

Build

Product or Feature Group Identifier(s) (PIDs, IDs, etc.)

Date of Initial Estimates

	REQ	HLD	LLD	CODE	LEV1	LEV2	LEV3	LEV4	GS	TOTAL
N&C Original KLOC Est										
N&C Revised KLOC Est		XX								
N&C KLOC Actuals										
Effort - Estimate (PM)										
Effort - Revised (PM)		XX								
Effort - Actual (PM)		XX								
# Defects - Estimate		YY								
# Defects - Actual		YY								
Est Phase End Dates		XX								

GS = General Ship
HLD = High-Level Design
LLD = Low-Level Design

LEV1 = Unit, or Level 1, Test
LEV2 = Integration, or Level 2, Test
LEV3 = System, or Level 3, Test
LEV4 = Beta, or Level 4, Test

N&C = New and Changed
PID = Product IDentifier
PM = Person Months
REQ = Requirements Analysis

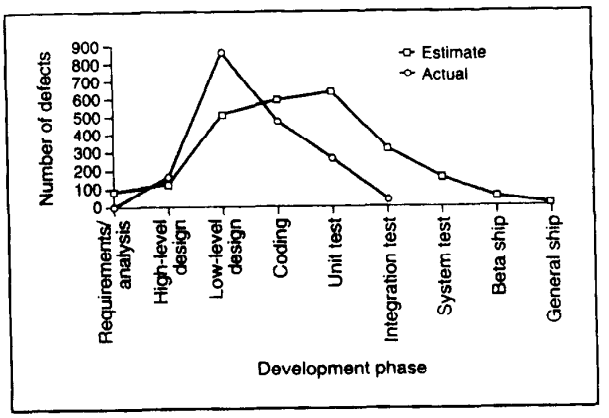


Figure 5. Estimated versus actual defect depletion curves.

Multiple data views. The data in Figure 6 helped the project manager analyze results from the integration test. The project team had little experience with the type of product to be developed, so a large number of defects were predicted. The team also decided to spend more effort on the unit test. After the unit test, the results seemed within plan, as shown in Figure 6a. During the integration test, some concern was raised that the number of defects found was too high. Once the data was normalized against the project size and compared to projections for the number of defects expected by the development team, the level of concern was lowered.

However, this project had a serious requirement error that was discovered in the later stages of the system test; this demonstrated why it's important to look at more than the total number of defects or the defect density, even when the number of defects is below expectations. A closer look at the early development stages shows that very few requirements or high-level design defects were found in the inspections. The low-level design inspections also found fewer defects than expected. What the project members missed in the data analysis during the unit and integration tests was the large number of design errors being detected (see Figure 6b). This example demonstrates the value of independent data collection and analysis as soon as it is available.

An objective analysis, or at least an analysis that looked at the project from a different viewpoint, might have spotted the anomaly. Unit test data accuracy might have been questioned as follows:

- Are some of the errors caused by high-level design defects?
- Why weren't any design defects found in the integration test?

When the data was charted with the defect source added, the design-defect data discovery rate in the unit test was obvious. The inaccuracy of the integration test data also became apparent. A closer look at the project revealed the source of the defect data had not been collected.

We also questioned members of the design inspection teams; we found that key people were not available for the high-level design inspection. As a result, we changed the entry criteria for low-level design to delay the inspection for two to three weeks, if necessary, to let a key system architect participate in the inspection. Part of the change required a risk analysis of the potential for scrubbing the low-level design work started before the inspection.

Using test cost. On one large project, the measured cost in the integration test was much higher than expected. Even though you know the cost of defects in test is high, an accurate cost tally can surprise you. If you haven't gathered the data for a project, the following example may convince you that the effort is worthwhile.

On this large project, it took

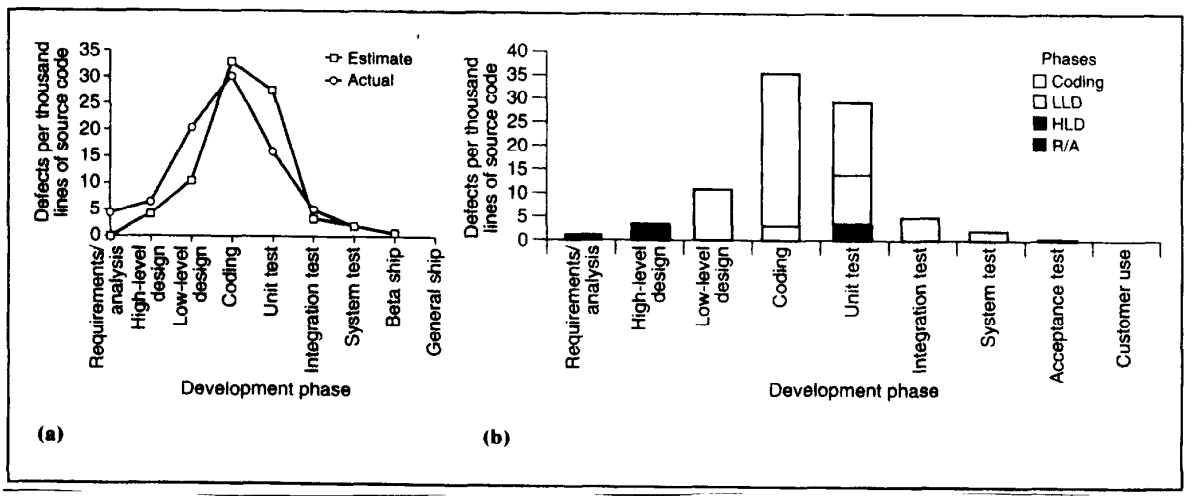


Figure 6. Projected versus actual number of defects found per thousand lines of code from inspections and test (a), and additional information when the defect source is included (b).

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

- 80 hours to find and fix the typical defect,
- 3.5 person-months to rebuild the product and rerun the test suite, and
- 8-10 calendar days to complete the retest cycle.

Three months' effort represents the fixed cost of test for this product area.

This analysis reemphasizes the need to spend more time and effort on inspecting design and code work products. How many additional hours should be spent inspecting the work products (design, code, and so forth) versus the months of effort expended in test?

Table 1. Defect density inferences.

Defect Density Observation	Inferences
Lower than expected	Size estimate is high (good). Inspection defect detection is low (bad). Work product quality is high (good). Insufficient level of detail in work product (bad).
Higher than expected	Size estimate is low (bad). Work product quality is poor (bad). Inspection defect detection is high (good). Too much detail in work product (good or bad).

Sanity check. Size estimates and software development attribute ratings are used as input to the Cocomo (Constructive Cost Model) estimating model.[3] (Joe Wiechec of Bull's Release Management group developed a Cocomo spreadsheet application based on Boehm[3] for schedule and effort sanity checks.) The accuracy of the effort estimate produced by Cocomo depends on accurate size estimate and software development attribute ratings (analyst capability, programmer capability, and so forth).

We compare the assumed defect rates and cost to remove these defects with the Cocomo output as a sanity check for the estimating process. Since project managers often assign attribute ratings optimistically, the defect data based on project and product history provides a crosscheck with the cost for unit and integration test derived from the Cocomo estimate. Reasonable agreement between Cocomo test-effort estimates and estimates derived from defect density and cost to find and fix per-defect figures confirm that attribute ratings have been reasonably revised. This sanity check works only for the attribute ratings, since both the Cocomo effort estimates and test cost estimates depend on the size estimate.

Project tracking

The keys to good project tracking are defining measurable and countable entities and having a repeatable process for gathering and counting. During the design phase, the metrics available to the project manager are

- effort spent in person-months,
- design-document pages, and
- defects found via work product review, inspection, or use in subsequent development stages.

Interpreting effort variance. When effort expenditures are below plan, the project will typically be behind schedule because the work simply isn't getting done. An alternative explanation might be that the design has been completed, but without the detail level necessary to progress to the next development phase. This merely sets the stage for disaster later.

We use inspection defect data from design inspection to guard against such damaging situations. If the density falls below a lower limit of 0.1 to 0.2 defects per page versus an expected 0.5 to 1.0 defects per page, the possibility increases that the document is incomplete. When defect detection rates are below 0.5 defects per page, the preparation and inspection rates are examined to verify that sufficient time was spent in document inspection. We also calculate the inspection defect density using the number of major defects and the estimated KLOC for the project. If the defect density is lower than expected, either the KLOC estimate is high or the detail level in the work product is insufficient (see Table 1).

When trying to determine which of the eight possible outcomes reflect the project status, project managers must draw on their experience and their team and product knowledge. I believe the project manager's ability to evaluate the team's inspection effectiveness will be better than the team's ability to estimate the code size. In particular, a 2-to-1 increase in detection effectiveness is far less likely than a 2-to-1 error in size estimating.

When effort expenditures are above the plan and work product deliverables are on or behind schedule, the size estimate was clearly lower than it should have been. This also implies later stages will require more effort than was planned.

In both cases, we found that the process instrumentation provided by inspections was very useful in validating successful design completion. The familiar "90 percent done" statements have disappeared. Inspections are a visible, measurable gate that must be passed.

Project tracking and analysis. Inspection defect data adds several dimensions to the project manager's ability to evaluate project progress. Glass[4] and Graham[5] claim that defects will always be a part of the initial development effort. (Glass says design defects are the result of the cognitive/creative design process, and Graham says errors are "inevitable, not sinful, and unintentional.") Humphrey presents data indicating that injection rates of 100 to 200 defects per KLOC of delivered code are not unusual.[6] Although we have seen a 7-to-1 difference in defect injection rates between projects, the variance is much less for similar projects.

For the same team doing similar work, the defect injection rate is nearly the same. The project analyzed in Figure 5 involved a second-generation product developed by many of the people who worked on the first-generation product. At the end of the low-level design phase, Steve Magee, the project manager, noticed a significant difference in the estimated and actual defect-depletion counts for low-level design defects. The estimate was derived from the earlier project, which featured many similar characteristics. There was a significant increase in the actual defect data. We

Table 2. Measures and possible inferences during requirements and design phases.

Measure	Value	Inference
Effort	Above plan	Project is larger than planned, if not ahead of schedule; or project is more complex than planned, if on schedule.
	Below plan	Project is smaller than estimated, if on schedule; or project is behind schedule; or design detail is insufficient, if on or ahead of schedule.
Defects Detected	Above plan	Size of project is larger than planned; or quality is suspect; or inspection detection effectiveness is better than expected.
	Below plan	Inspections are not working as well as expected; or design lacks sufficient content; or size of project is smaller than planned; or quality is better than expected.
Size	Above plan	Marketing changes or project complexity are growing — more resource or time will be needed to complete later stages.
	Below plan	Project is smaller than estimated; or something has been forgotten.

also had some numbers from the first project that suggested inspection effectiveness (defects found by inspection divided by the total number of defects in the work product) was in the 75 percent range for this team.

Again, we were able to use our inspection data to infer several theories that explained the differences. The defect shift from code to low-level design could be attributed to finding defects in the low-level design inspections on the second project, rather than during code inspections in the first project. A closer look at the first project defect descriptions from code inspections revealed that a third of the defects were missing functionality that could be traced to the low-level design, even though many defects had been incorrectly tagged as coding errors. Reevaluating the detailed defect descriptions brought out an important fact: The percentage of design defects detected in code inspection on the first project was higher than we thought.

We were also concerned by the number of defects, which exceeded the total we had estimated even after accounting for the earlier detection. It seemed more likely that the size estimate was low rather than that there was a significant increase in defect detection effectiveness. In fact, the size estimate was about 50 percent low at the beginning of low-level design. The project manager adjusted his size estimates and consequently was better able to predict unit test defects when code inspections were in progress, and time for both unit and integration test.

Using defect data helped the project manager determine that design defects were being discovered earlier and project size was larger than expected. Hence, more coding effort and unit and integration test time would be needed.

Figure 7 shows the actual data as this project entered system test. Comparing the data in Figures 5 and 7 indicates a shift in defect detection to earlier stages in the development cycle; hence, the project team is working more effectively.

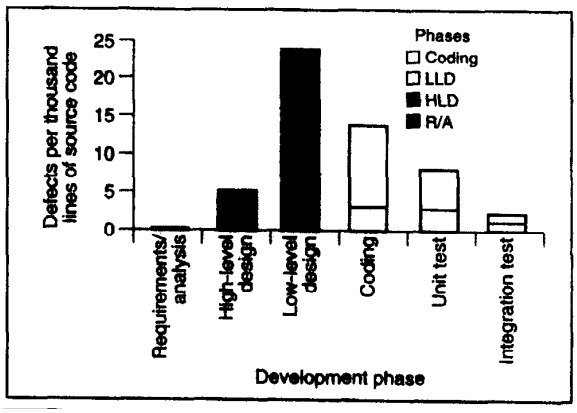


Figure 7. Actual defect density data for the project depicted in Figure 5.

Defect data can be used as a key element to improve project planning. Once a size estimate is available, historical data can be used to estimate the number of defects expected in a project, the development phase where defects will be found, and the cost to remove the defects.

Once the defect-depletion curve for the project is developed, variances from the predictions provide indicators that project managers can examine for potential trouble spots. Table 2 summarizes these measures, their value (above or below plan), and the possible troubles indicated. These measures and those listed in Table 1 answer many of the questions in the design box in Figure 3.

One difficulty project managers must overcome is the unwillingness of the development staff to provide defect data. Grady[7] mentions the concept of public versus private data, particularly regarding inspection-data usage. Unless the project team is comfortable with making this data available to the project manager, it is difficult to gather and analyze the data in time for effective use.

I believe that continuing education on the pervasiveness of defects, and recog-

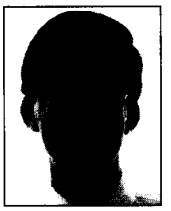
nition that defects are a normal occurrence in software development, is a critical first step in using defect data more effectively to measure development progress and product quality. Only through collecting and using defect data can we better understand the nature and cause of defects and ultimately improve product quality. ■

Acknowledgments

I thank John T. Harding and Ron Radice for their many hours of discussion on defect removal as a project management tool; Steve Magee, Fred Kuhlman, and Ann Holladay for their willingness to use the methods described in this article on their projects; Jean-Yves LeGoic for his excellent critique; Barbara Ahlstrand, Robin Fulford, and George Mann for inspecting the manuscript; and the anonymous referees for their helpful recommendations.

References

1. E. Weller, "Lessons from Three Years of Inspection Data," *IEEE Software*, Vol. 10, No. 5, Sept. 1993, pp. 38-45.
2. W. Humphrey, *Managing the Software Process*, 1990, Addison-Wesley, Reading, Mass., pp. 352-355.
3. B.W. Boehm, *Software Engineering Economics*, Prentice Hall, Englewood Cliffs, N.J., 1981.
4. R. Glass, "Persistent Software Errors: 10 Years Later," *Proc. First Int'l Software Test, Analysis, and Rev. Conf.*, Software Quality Engineering, Jacksonville, Fla., 1992.
5. D. Graham, "Test Is a Four Letter Word: The Psychology of Defects and Detection," *Proc. First Int'l Software Testing, Analysis, and Rev. Conf.*, 1992, Software Quality Engineering, Jacksonville, Fla.
6. W. Humphrey, "The Personal Software Process Paradigm," *Sixth Software Eng. Process Group Nat'l Meeting*, Software Eng. Inst., Carnegie Mellon Univ., Pittsburgh, 1994.
7. R. Grady, *Practical Software Metrics For Project Management and Process Improvement*, Prentice Hall, Englewood Cliffs, N.J., 1992, pp. 104-107.



Edward F. Weller is a technical staff engineer at Motorola's Satellite Communications Division, where he is responsible for developing software subcontract management processes. Previously, he was the software process architect for the Bull HN Information System's Mainframe Operating Systems Group.

Weller received a BSEE from the University of Michigan and an MSEE from the Florida Institute of Technology. He was awarded Article of the Year honors in 1993 by *IEEE Software* for authoring "Lessons from Three Years of Inspection Data." He is a member of the Software Engineering Institute's Software Measurements Steering Committee and is co-chair of the Software Inspection and Review Organization, a special-interest group promoting inspection process usage. He is a senior member of the IEEE and a member of the IEEE Computer Society.

Readers can contact Weller at Motorola Government Systems and Technology, 2501 S. Price Rd., MS G1137, Chandler, AZ 85248-2899, e-mail ed_weller-p26708@email.mot.com.